Programowanie obiektowe C++ w środowisku Windows

dr inż. Sławomir Koczubiej

Politechnika Świętokrzyska
Wydział Zarządzania i Modelowania Komputerowego
Katedra Technologii Informatycznych

(12 października 2024)

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 czasem warto zajrzeć.

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.

Treść wykładów

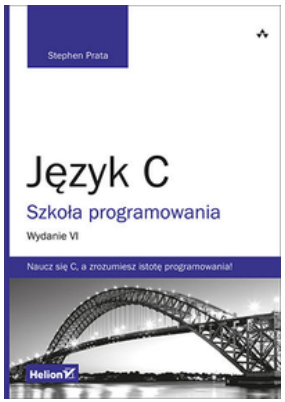
- Wstęp do programowania w języku C. Instrukcje, zmienne i ich typy, tablice w języku C.
- Funkcje, obsługa wejścia i wyjścia w języku C, obsługa plików w języku C.
- Wprowadzenie do programowania w języku C++.
- Obiekty i klasy. Ochrona i kapsułkowanie.
- Dziedziczenie, dziedziczenie wielobazowe. Polimorfizm.
- Wyjątki i ich obsługa.
- Obiekty i zarządzanie pamięcią. Tworzenie i niszczenie obiektów.
- Operatory przeciążone.
- Obsługa plików.

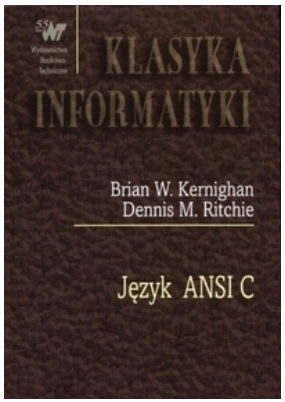
Treść laboratoriów

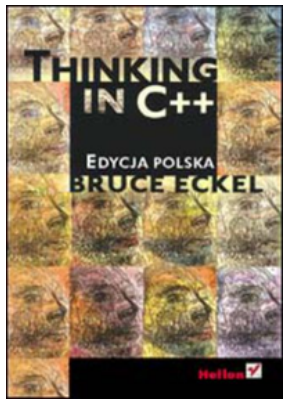
- Struktura programu w języku C. Rola plików nagłówkowych.
- Operacje wejścia-wyjścia z wykorzystaniem biblioteki języka C.
- Wybrane operatory. Własności i priorytety operatorów.
- Instrukcje warunkowa przełączające, pętle. Algorytmy przetwarzania iteracyjnego. Tablice i instrukcje pętli.
- Definiowane funkcji. Przekazywanie parametrów.
- Rodzaje błędów i ich diagnozowanie. Testowanie programu.
- 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.
- Przeciążanie operatorów.
- Obsługa plików.

Literatura

- S. Prata. *Szkoła programowania. Język C.* Wydawnictwo Helion, Gliwice 2006.
- B. Kernighan, P. Ritchie. *Język ANSI C.* Wydawnictwo Naukowo Techniczne, Warszawa 2004.
- S. Prata. *Szkoła programowania. Język C++.* Wydawnictwo Helion, Gliwice 2006.
- J. Grębosz. *Symfonia C++ Standard.* Wydawnictwo Edition 2000, Kraków 2009.
- B. Eckel. *Thinking in C++. Edycja polska.* Wydawnictwo Helion, Gliwice 2002.







Po co mi programowanie?

Linus Torvalds

Computer programming is not for everyone. I think it's reasonably specialized, and nobody really expects most people to have to do it.

Steve Jobs

Everybody in this country should learn how to program a computer, because it teaches you how to think.

Celem nauki programowania jest wykształcenie umiejętności **myślenia komputacyjnego** (*computational thinking*), które obejmuje myślenie algorytmiczne w rozwiązywaniu problemów oraz umiejętność programowania rozszerzone na wszystkie obszary działalności ludzkiej. Takie podejście przede wszystkim pozwala zwiększyć efektywność i ułatwić pracę.

Nawet życiowe problemy i decyzje można potraktować jako problem algorytmiczny.

Języki programowania lub **makropolecenia** udostępniają szereg narzędzi pozwalających na przyśpieszenie i zwiększenie efektywności pracy, zrobienie czegoś w łatwiejszy sposób, a nawet utworzenie kompletnie nowych narzędzi.

Rozwój języków programowania znacznie obniżył próg umiejętności jakie należy posiadać, żeby zacząć naukę. Nie trzeba posiadać żadnych informacji na temat budowy komputera, nie trzeba mieć solidnych podstaw matematycznych (choć te są przydatne), nie trzeba mieć superkomputera ani kupować dodatkowego drogiego oprogramowania.

Przypomnienie podstawowych terminów

Program komputerowy (aplikacja) – sekwencja symboli (zrozumiałych dla komputera rozkazów) przeznaczonych do przetworzenia zgodnie z pewnymi regułami, zwanymi **językiem programowania**.

Program w postaci języka *zrozumiałego* dla człowieka nazywany jest **kodem źródłowym**, podczas gdy program wyrażony w postaci zrozumiałej dla maszyny (to jest za pomocą ciągu liczb, a bardziej precyzyjnie zer i jedynek) nazywany jest kodem maszynowym bądź postacią binarną (wykonywalną).

Program jest zazwyczaj wykonywany przez komputer, bezpośrednio – jeśli wyrażony jest w języku zrozumiałym dla danej maszyny lub pośrednio – gdy jest interpretowany przez inny program (interpreter).

Programy komputerowe można zaklasyfikować według ich zastosowań. Wyróżnia się zatem aplikacje użytkowe, systemy operacyjne, gry, kompilatory i inne. Programy wbudowane wewnątrz urządzeń określa się jako **firmware**.

W najprostszym modelu wykonanie programu (zapisanego w postaci zrozumiałej dla maszyny) polega na umieszczeniu go w pamięci operacyjnej komputera i wskazaniu procesorowi adresu pierwszej instrukcji.

Po tych czynnościach procesor będzie wykonywał kolejne instrukcje programu, aż do jego zakończenia.

Program komputerowy będący w trakcie wykonania nazywany jest **procesem** lub **zadaniem**.

Tworzenie programu komputerowego można podzielić na dwa etapy:

- Po zrodzeniu się **pomysłu** powinien powstać **algorytm**. Algorytm wymusza stosowanie podziału programu na funkcje. zmienne, obiekty, na których program będzie operował, jak również wprowadzenie procedur, które opisują wykonywane operacje.
- Algorytm należy zapisać w języku programowania, stosując dostępne struktury danych i funkcje – tworzenie **kodu źródłowego**. W trakcie tworzenia programu kod jest poddawany **debugowaniu** – wyszukiwanie błędów.

Językiem programowania nazywamy zestaw zasad tekstowego lub graficznego opisu algorytmu za pomocą przyjętych elementów języka.

Podobnie jak języki naturalne, język programowania składa się ze zbiorów reguł syntaktycznych oraz semantyki, które opisują, jak należy budować poprawne wyrażenia oraz jak komputer ma je rozumieć

Język programowania pozwala na precyzyjny zapis algorytmów oraz innych zadań, jakie komputer ma wykonać.

Podział języków programowania

Kod maszynowy (język maszynowy) – język programowania, w którym zapis programu wymaga **instrukcji** bezpośrednio jako liczb, które są rozkazami i danymi bezpośrednio pobieranymi przez procesor wykonujący ten program.

Jest dopasowany do konkretnego typu procesora i przeznaczony do bezpośredniego wykonania przez procesor. Analiza kodu maszynowego jest praktycznie niemożliwa przez człowieka.

Języki niskopoziomowe – przedstawiają one instrukcje udostępniane przez system komputerowy w postaci prostych oznaczeń (o ograniczonej liczbie, zakodowane w procesorze). Do języków niskopoziomowych należą **assemblery**.

Języki wysokopoziomowe – składnia i słowa kluczowe mają maksymalnie ułatwić rozumienie kodu programu dla człowieka, tym samym zwiększając poziom abstrakcji i dystansując się od budowy sprzętu komputerowego.

Kod napisany w języku wysokiego poziomu nie jest bezpośrednio *zrozumiały* dla komputera – większość kodu stanowią tak naprawdę normalne słowa (najczęściej w języku angielskim).

Języki wysokopoziomowe dzielimy na dwie grupy:

- interpretowane,
- kompilowane.

Języki interpretowane nie wymagają kompilacji tylko **interpretera**. Są przechowywane w postaci kodu źródłowego i dopiero podczas uruchomienia wczytywane, analizowane i wykonywane przez interpreter języka – **PHP**, **JavaScript**, **Python**, **PERL**. Programy przeznaczone do interpretacji często nazywane są **skryptami**.

Języki kompilowane wymagają procesu kompilacji kodu źródłowego do postaci kodu maszynowego (postaci binarnej). Robi to specjalny program zwany **kompilatorem**, dzięki czemu możliwe staje się jego późniejsze uruchomienie. Języki kompilowane: **Pascal**, **C**, **C++**, **Fortran**, **Java**.

Do utworzenia programu w danym języku niezbędne są **edytor** tekstu, **debugger** i **kompilator**. Programy te mogą tworzyć integralne środowisko pracy, udostępniające kombinacje tych funkcji – mówimy wówczas o **środowisku programistycznym**.

Aby przyśpieszyć tworzenie aplikacji, szczególnie z interfejsem graficznym, tworzy się narzędzia **RAD** (*Rapid Application Development*), które umożliwiają tworzenie programów z gotowych komponentów (w sensie elementów interfejsu i funkcji z implementacją typowych algorytmów).

Syntaktyka i semantyka

Aby ciąg znaków mógł być rozpoznany jako program napisany w danym języku, musi spełniać reguły **syntaktyki** (składni). Składnia opisuje:

- rodzaje dostępnych symboli,
- zasady, według których symbole mogą być łączone w większe struktury.

Należy zauważyć, że na etapie przetwarzania składni w ogóle nie jest brane pod uwagę znaczenie poszczególnych symboli. W praktyce kod poprawny składniowo nie musi być poprawny semantycznie. Występuje tu analogia do języków naturalnych. Zdanie „Bźdźiągwy się muciosząl" jest poprawne pod względem gramatycznym, lecz nie posiada żadnego znaczenia, ponieważ zostały w nim użyte nieistniejące słowa.

Semantyka języka programowania definiuje precyzyjnie znaczenie poszczególnych symboli oraz ich funkcję w programie. Semantykę najczęściej definiuje się słownie, ponieważ większość z jej zagadnień jest trudna lub wręcz niemożliwa do ujęcia w jakikolwiek formalizm.

Część błędów semantycznych można wychwycić już w momencie wstępnego przetwarzania kodu programu, np. próbę odwołania się do nieistniejącej funkcji, lecz inne mogą ujawnić się dopiero w trakcie wykonywania.

Błędy

W trakcie pisania kodu nie da się uniknąć błędów. Błędy mogą wynikać z niepoprawnego wpisania instrukcji, pominięcia kropek, przecinków, nawiasów, itp. Takie błędy noszą nazwę **syntaktycznych** (składniowych).

Oprócz błędów syntaktycznych, można spotkać jeszcze błędy **semantyczne** (znaczeniowe, wykonania) i **logiczne**.

Błędy wykonania występują w chwili odtwarzania procedur (programu) i mogą wynikać z próby uruchomienia nieistniejącej procedury, otwarcia nieistniejącego pliku lub złego typowania zmiennych.

Błędy logiczne zwykle nie wywołują żadnych komunikatów błędu. Chociaż program jest poprawny pod względem syntaktycznym i semantycznym, nie powoduje żadnych problemów kompilacji i uruchamiania to sam rezultat działania może być błędny.

Błędy logiczne powodują otrzymanie wyników innych niż się spodziewano. Są to zwykle błędy bardzo trudne do odnalezienia.

Dla początkujących, problemem jest mnogość dostępnych obecnie języków programowania. Najbardziej klasyczne języki programowania to **C** i **C++**, popularne są **Java** i **C#**, modny jest **Python**, **JavaScript** i **Ruby**.

Analitycy często używają jżyków pozwalających na szybkie pisanie aplikacji np. **Visual Basic** i eksploracji baz danych: **SQL**, czy dedykowane do analizy danych **R** lub do obliczeń numerycznych (analizy danych też) **MATLAB**.

- https://www.tiobe.com/tiobe-index/ – Wskaźnik popularności języków programowania.
- https://insights.stackoverflow.com/survey/ – Ankiety serwisu społecznościowego programistów.

Supported Features	Visual Studio Community Free download	Visual Studio Professional Buy	Visual Studio Enterprise Buy
⊕ Supported Usage Scenarios	●●●○	●●●●	●●●●
Development Platform Support [2]	●●●●	●●●●	●●●●
⊕ Integrated Development Environment	●●●○	●●●○	●●●●
⊕ Advanced Debugging and Diagnostics	●●○○	●●○○	●●●●
⊕ Testing Tools	●○○○	●○○○	●●●●
⊕ Cross-platform Development	●●○○	●●○○	●●●●
⊕ Collaboration Tools and Features	●●●●	●●●●	●●●●

Microsoft Azure (dawniej: *Microsoft Imagine Premium*, *Microsoft DreamSpark* oraz *MSDN Academic Alliance*) – program przedsiębiorstwa Microsoft skierowany do uczniów, studentów i pracowników naukowych. Uczestnicy mogą pozyskać darmowe kopie pewnej części oprogramowania firmy Microsoft (systemów operacyjnych, serwerów i środowisk programowania) pod warunkiem, że będą korzystać z otrzymanego oprogramowania jedynie w celach edukacyjnych (w ramach licencji EDU).

Visual Studio Community to uproszczona, ale jednocześnie darmowa wersja środowiska programistycznego Visual Studio. Program kierowany jest do studentów, twórców oprogramowania w duchu otwartego kodu oraz indywidualnych programistów (ograniczenia dotyczące dochodów firmy, liczby komputerów, liczby instalacji).

Visual Studio Enterprise to wyposażona w zaawansowane narzędzia do projektowania, testowania i wdrażania dużych aplikacji, przeznaczona dla dużych zespołów ($\sim$15 000 zł).

Paradygmat programowania – wzorzec programowania komputerów, który definiuje sposób patrzenia programisty na przepływ sterowania i wykonywanie programu komputerowego.

Przykładowo, w programowaniu **obiektowym** jest on traktowany jako zbiór współpracujących ze sobą obiektów, podczas gdy w programowaniu **funkcyjnym** definiujemy, co trzeba wykonać, a nie w jaki sposób.

Różne języki programowania mogą wspierać różne paradygmaty programowania. Przykładowo, **Smalltalk** i **Java** są ściśle zaprojektowane dla potrzeb programowania obiektowego, natomiast **Haskell** jest językiem funkcyjnym. Istnieją także języki wspierające kilka paradygmatów, np. **Python**.

Wiele paradygmatów jest dobrze znanych z tego, jakie praktyki są w nich zakazane, a jakie dozwolone.

Na przykład, ścisłe programowanie funkcyjne nie pozwala na tworzenie skutków ubocznych (dowolny efekt wyrażenia, lub wywołania funkcji, który wykracza poza zwrócenie wartości). W programowaniu strukturalnym nie korzysta się z instrukcji skoku.

Zależności między paradygmatami programowania mogą przybierać skomplikowane formy, ponieważ jeden język może wspierać wiele różnych paradygmatów. Na przykład, **C++** posiada elementy programowania proceduralnego, obiektowego oraz uogólnionego, co stanowi o nim, że jest hybrydowym językiem. To projektanci i programiści decydują, jak zbudować z nich w pełni działający program.

Przykłady paradygmatów programowania:

- programowanie **imperatywne**,
- programowanie **deklaratywne**,
- programowanie **proceduralne**,
- programowanie **strukturalne**,
- programowanie funkcyjne,
- programowanie **obiektowe**,
- programowanie uogólnione,
- programowanie **sterowane zdarzeniami**,
- programowanie logiczne,
- programowanie aspektowe,
- programowanie agentowe,
- programowanie modularne.

Programowanie imperatywne – paradygmat programowania, który opisuje proces wykonywania jako sekwencję instrukcji zmieniających stan programu, podobnie jak tryb rozkazujący, wyraża żądania jakichś czynności do wykonania.

Programy imperatywne składają się z ciągu komend do wykonania przez komputer. Rozszerzeniem (w sensie wbudowanych funkcji) i rodzajem (w sensie paradygmatu) programowania imperatywnego jest programowanie proceduralne.

Programowanie deklaratywne – rodzina paradygmatów programowania, które nie są z natury imperatywne. W przeciwieństwie do programów napisanych imperatywnie, programista opisuje warunki, jakie musi spełniać końcowe rozwiązanie (co chcemy osiągnąć), a nie szczegółową sekwencję kroków, które do niego prowadzą (jak to zrobić).

Przykłady języków: **XSLT**, **Prolog**, **HTML**.

Programowanie proceduralne to paradygmat programowania zalecający dzielenie kodu na procedury, czyli fragmenty wykonujące ściśle określone operacje.

Procedury nie powinny korzystać ze zmiennych globalnych (w miarę możliwości), lecz pobierać i przekazywać wszystkie dane (czy też wskaźniki do nich) jako parametry wywołania.

Programowanie strukturalne to paradygmat programowania zalecający hierarchiczne dzielenie kodu na bloki, z jednym punktem wejścia i jednym lub wieloma punktami wyjścia.

Chodzi przede wszystkim o nieużywanie instrukcji skoku. Dobrymi strukturami są np. instrukcje: warunkowe, pętle, wyboru.

Strukturalność zakłócają instrukcje typu: `break`, `continue`, `switch`, które jednak w niektórych przypadkach znacząco podnoszą czytelność kodu.

Praktycznie w każdym języku można programować strukturalnie, jednakże w niektórych jest to styl naturalny (np. **Pascal**).

Programowanie obiektowe (*Object-Oriented Programming*) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących stan (czyli dane, nazywane polami lub właściwościami) i zachowanie (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

Podejście to różni się od tradycyjnego programowania proceduralnego, gdzie dane i procedury nie są ze sobą bezpośrednio związane. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów.

Największym atutem programowania, projektowania oraz analizy obiektowej jest zgodność takiego podejścia z rzeczywistością – mózg ludzki jest w naturalny sposób najlepiej przystosowany do takiego podejścia przy przetwarzaniu informacji. Przykłady języków: **C++**, **JAVA**.

Programowanie sterowane zdarzeniami – metodologia tworzenia programów komputerowych, która określa sposób ich pisania z punktu widzenia procesu przekazywania sterowania między poszczególnymi modułami tej samej aplikacji.

Programowanie sterowane zdarzeniami jest mocno powiązane ze środowiskami wieloprocesowymi, z graficznymi środowiskami systemów operacyjnych oraz z programowaniem obiektowym.

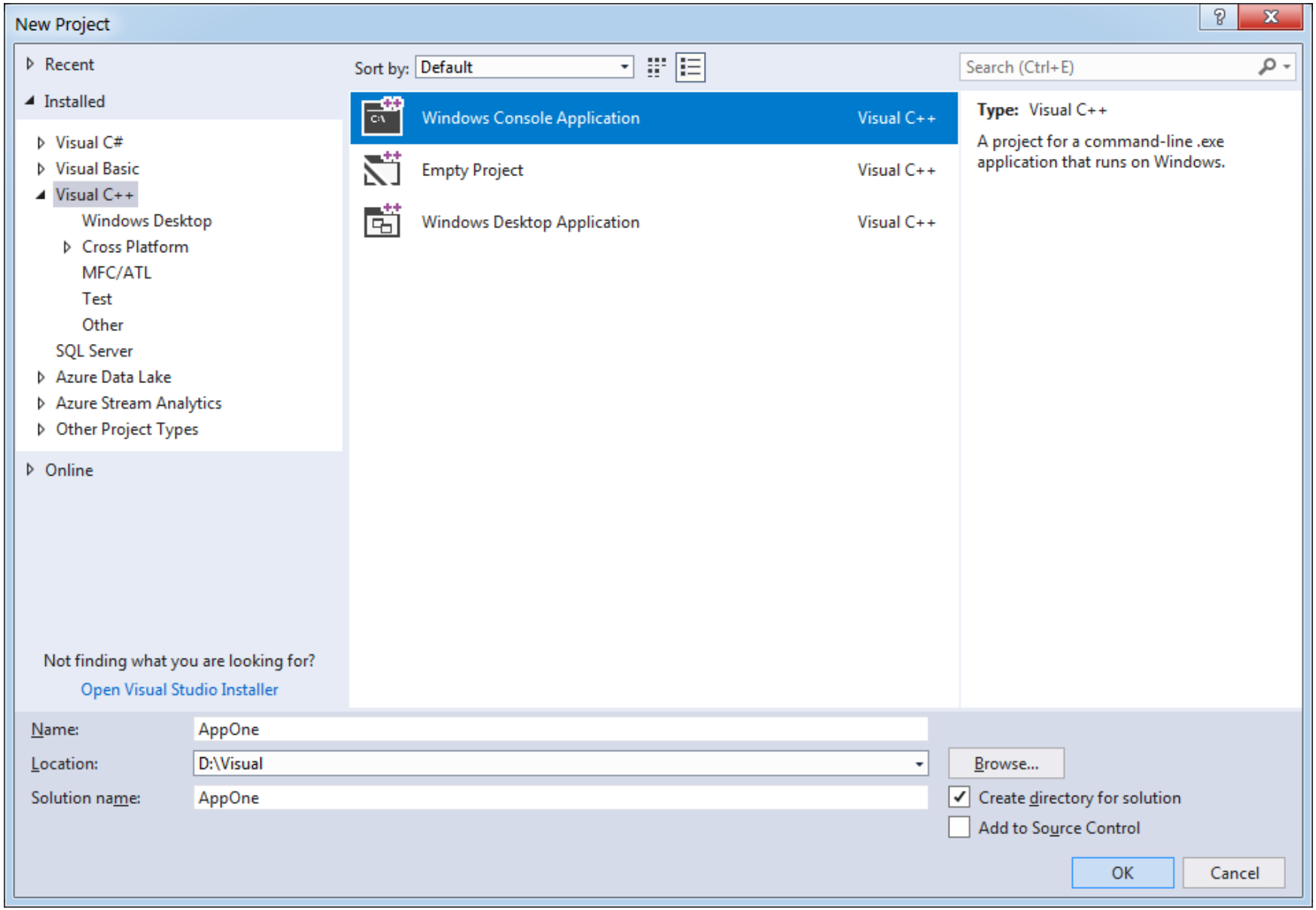
Paradygmat zakłada, że program jest cały czas bombardowany zdarzeniami, na które musi odpowiedzieć, i że przepływ sterowania w programie jest całkowicie niemożliwy do przewidzenia z góry.

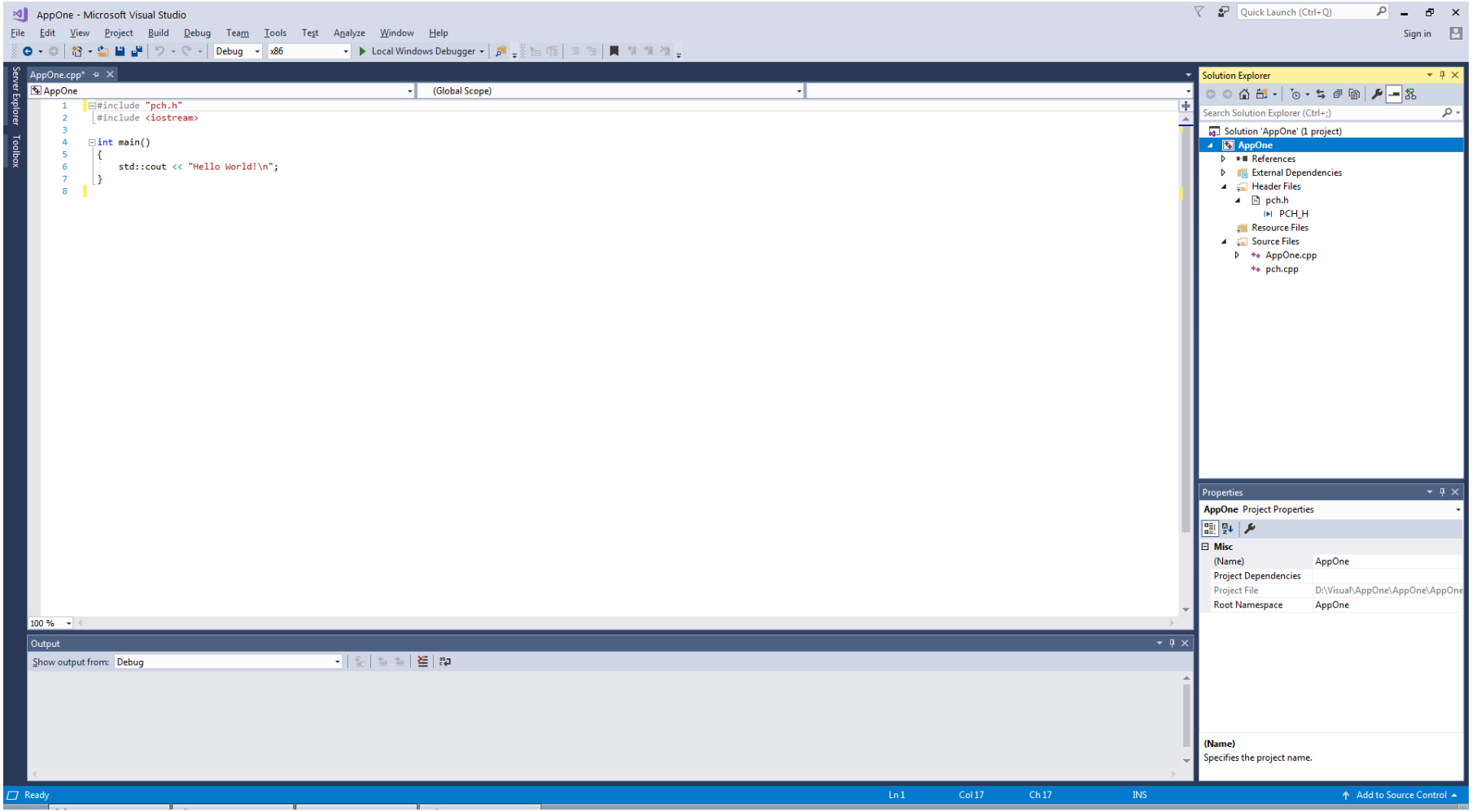
Programowanie zdarzeniowe jest dominującym typem programowania związanego z graficznym interfejsem użytkownika (*Graphical User Interface*) – zdarzenia to naciśnięcia myszy, klawiszy, żądania odświeżenia przez system okienkowy, różne zdarzenia sieciowe i inne.

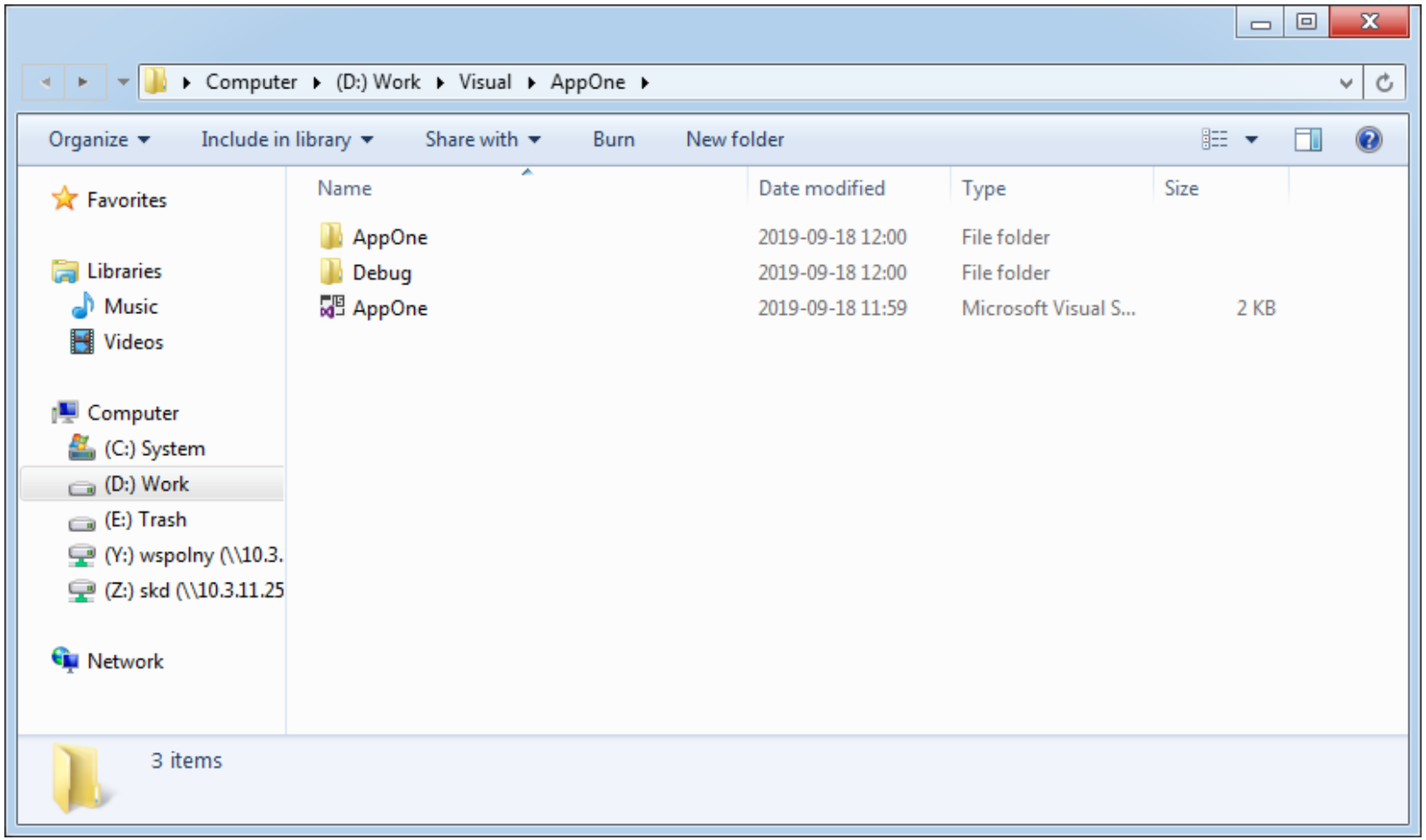
W programowaniu zdarzeniowym ważne jest żeby nie obsługiwać zbyt długo danego zdarzenia, bo blokuje się w ten sposób obsługę innych. W przypadku serwerów obniżyło by to znacznie wydajność, w przypadku GUI program zbyt wolno odpowiadałby na akcje użytkownika.

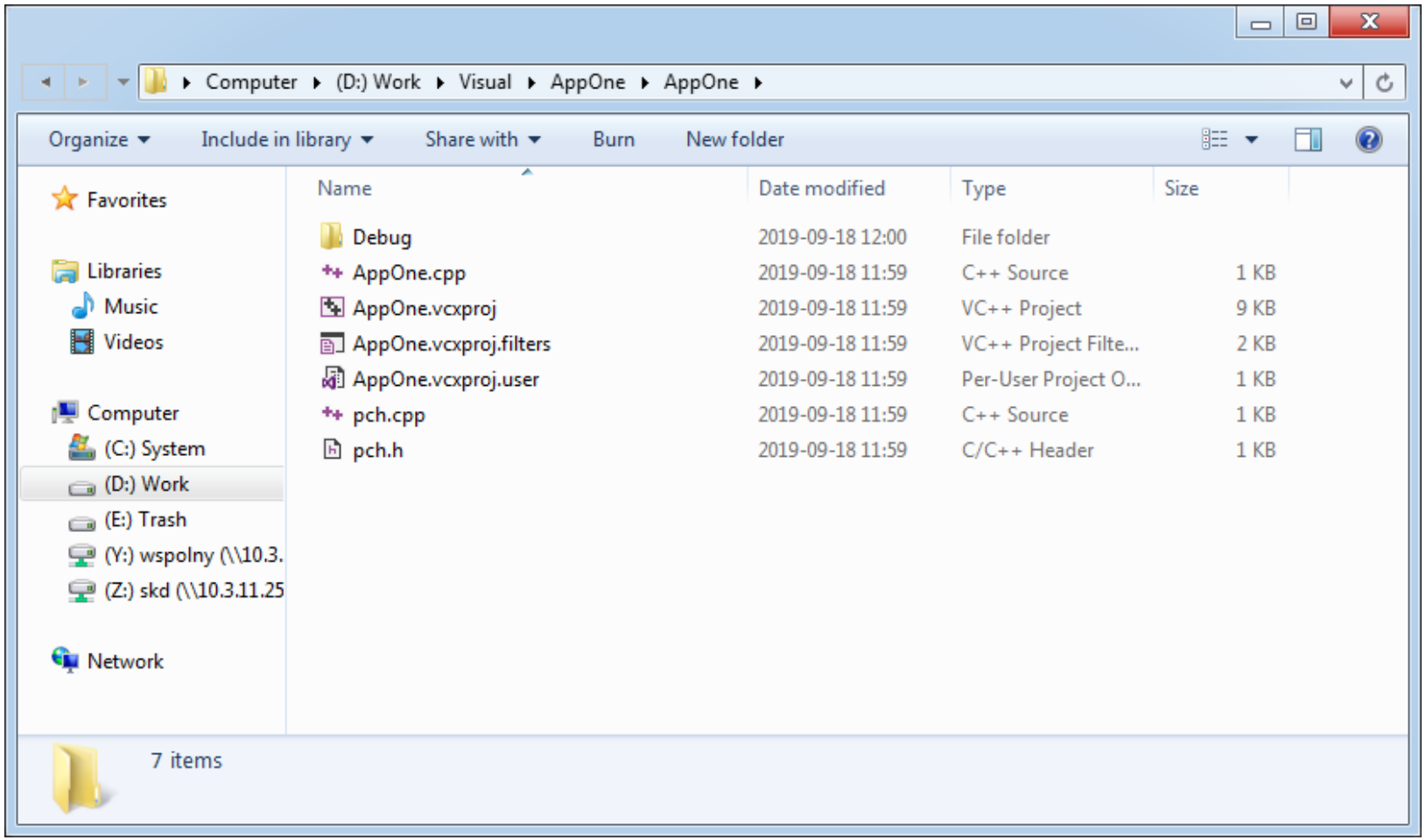
Można to osiągnąć za pomocą asynchronicznego I/O, wielowątkowości, rozbijania zdarzenia na podzdarzenia i wielu innych mechanizmów.

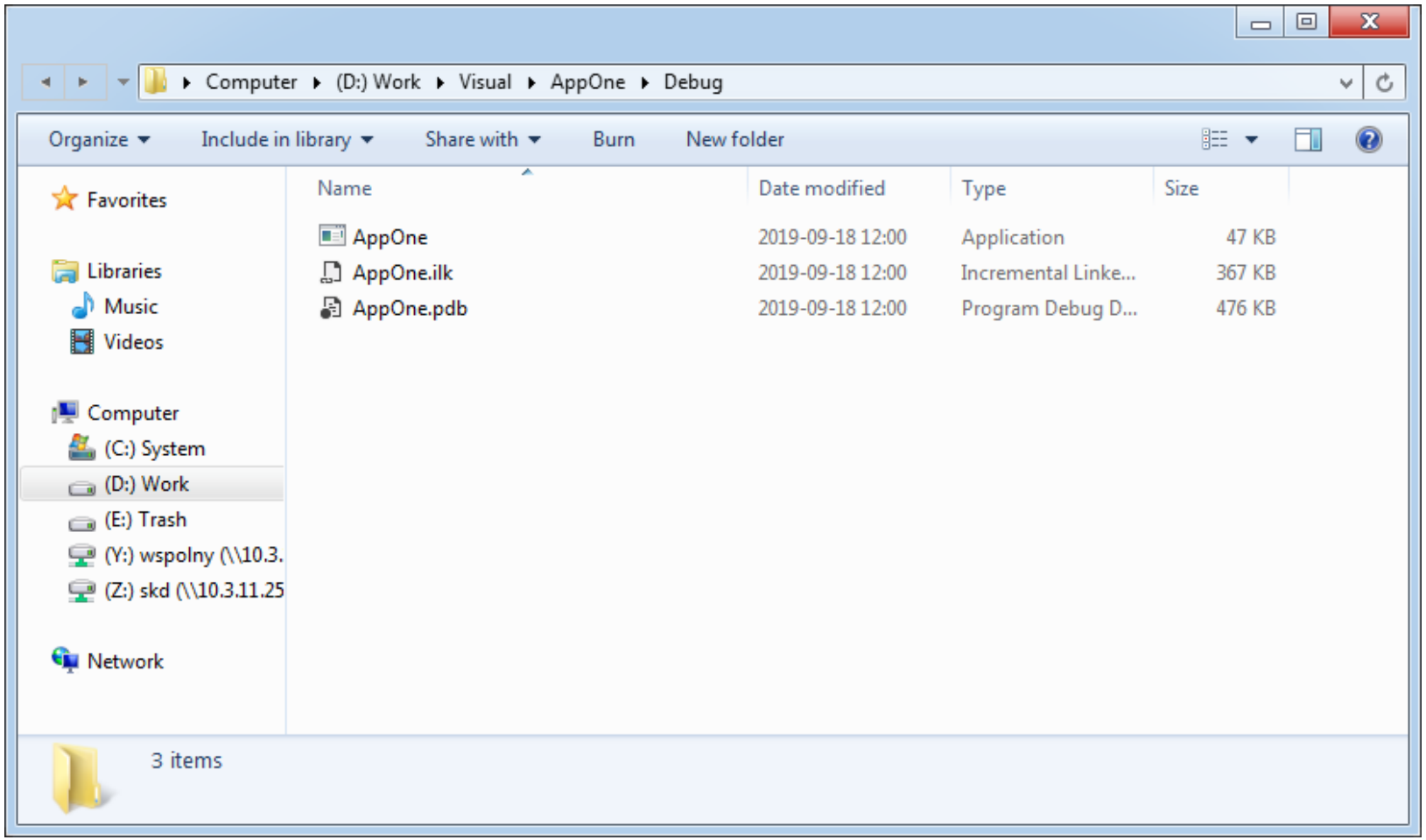
Środowisko programowania - Windows




AppOne - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug x86 Local Windows Debugger
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.

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

File Edit Options Encoding Help 100 %

```
Microsoft Visual Studio Solution File, Format Version 12.00
# Visual Studio 15
VisualStudioVersion = 15.0.28307.852
MinimumVisualStudioVersion = 10.0.40219.1
Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "AppOne", "AppOne\AppOne.vcxproj", "{44E04467-93A1-4015-8CC6-70A619274
EndProject
Global
        GlobalSection(SolutionConfigurationPlatforms) = preSolution
                Debug|x64 = Debug|x64
                Debug|x86 = Debug|x86
                Release|x64 = Release|x64
                Release|x86 = Release|x86
        EndGlobalSection
        GlobalSection(ProjectConfigurationPlatforms) = postSolution
                {44E04467-93A1-4015-8CC6-70A61927474D}.Debug|x64.ActiveCfg = Debug|x64
                {44E04467-93A1-4015-8CC6-70A61927474D}.Debug|x64.Build.0 = Debug|x64
                {44E04467-93A1-4015-8CC6-70A61927474D}.Debug|x86.ActiveCfg = Debug|Win32
                {44E04467-93A1-4015-8CC6-70A61927474D}.Debug|x86.Build.0 = Debug|Win32
                {44E04467-93A1-4015-8CC6-70A61927474D}.Release|x64.ActiveCfg = Release|x64
                {44E04467-93A1-4015-8CC6-70A61927474D}.Release|x64.Build.0 = Release|x64
                {44E04467-93A1-4015-8CC6-70A61927474D}.Release|x86.ActiveCfg = Release|Win32
                {44E04467-93A1-4015-8CC6-70A61927474D}.Release|x86.Build.0 = Release|Win32
        EndGlobalSection
        GlobalSection(SolutionProperties) = preSolution
                HideSolutionNode = FALSE
        EndGlobalSection
        GlobalSection(ExtensibilityGlobals) = postSolution
                SolutionGuid = {7FEB9663-D918-44FD-9D70-3C2B4DDFF7FD}
        EndGlobalSection
EndGlobal
```

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>
```

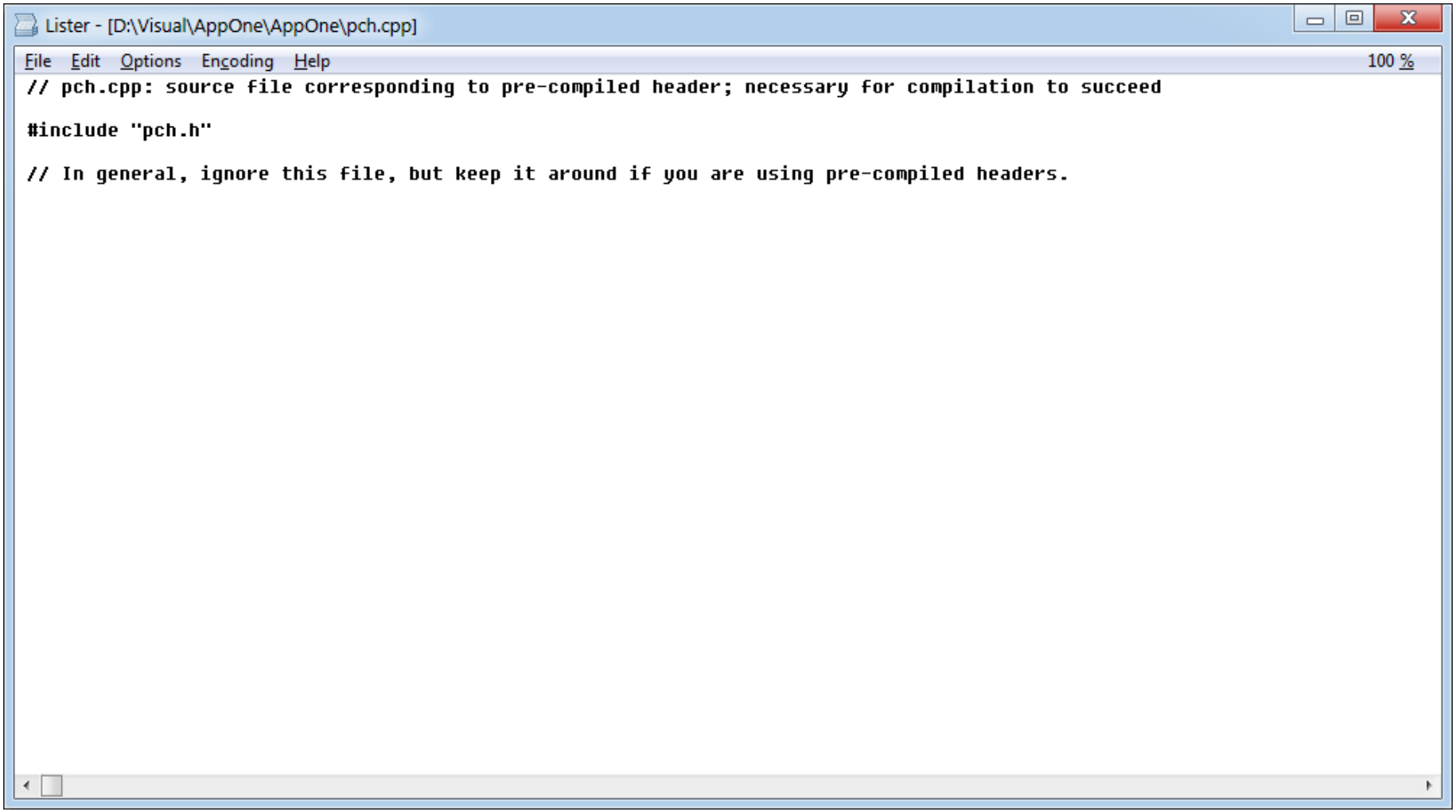
Plik `*.vcxproj.user` – plik *filtrów*, określa miejsce umieszczenia pliku, który jest dodawany do rozwiązania.

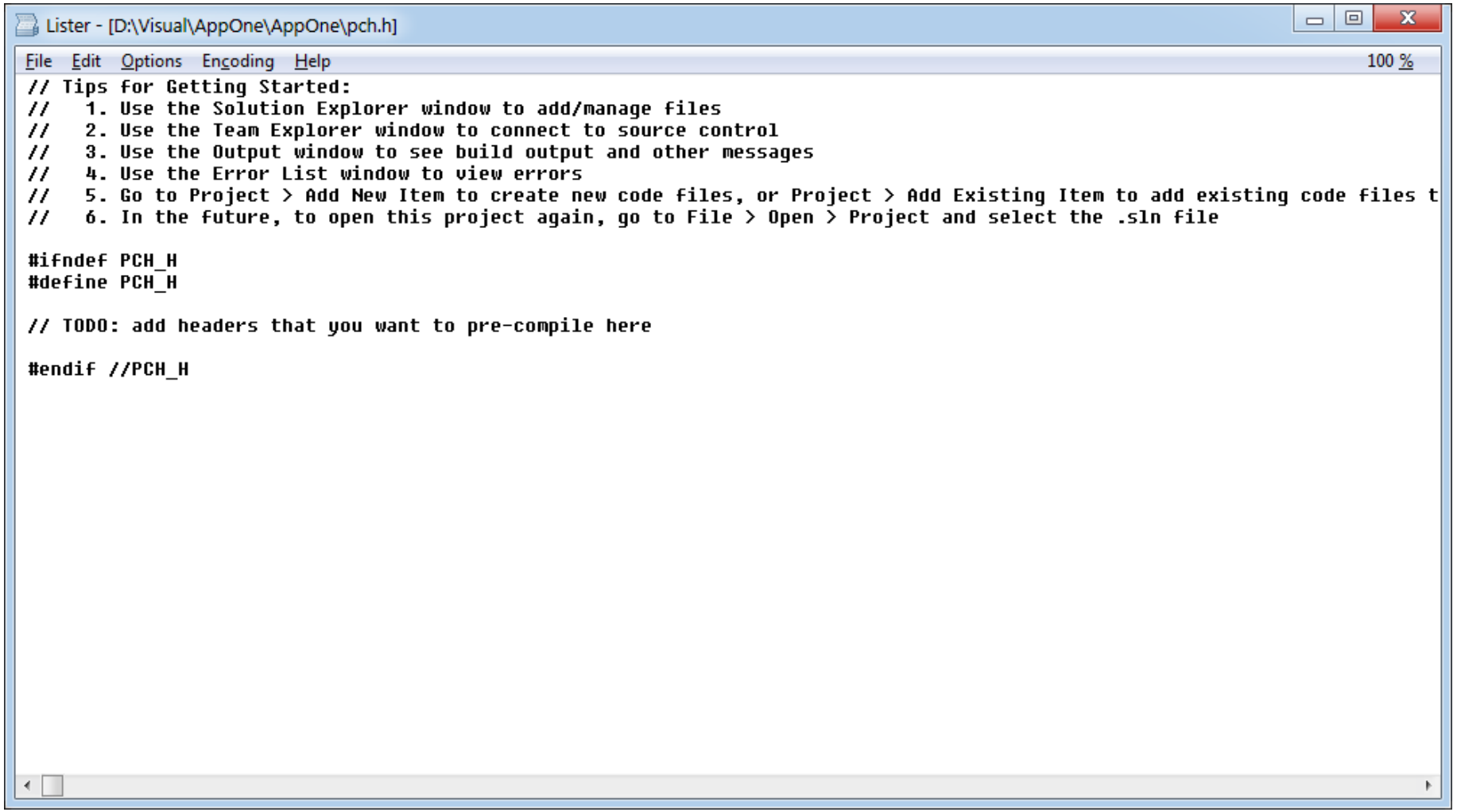
Lister - [D:\Visual\AppOne\AppOne\AppOne.vcxproj.filters]

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

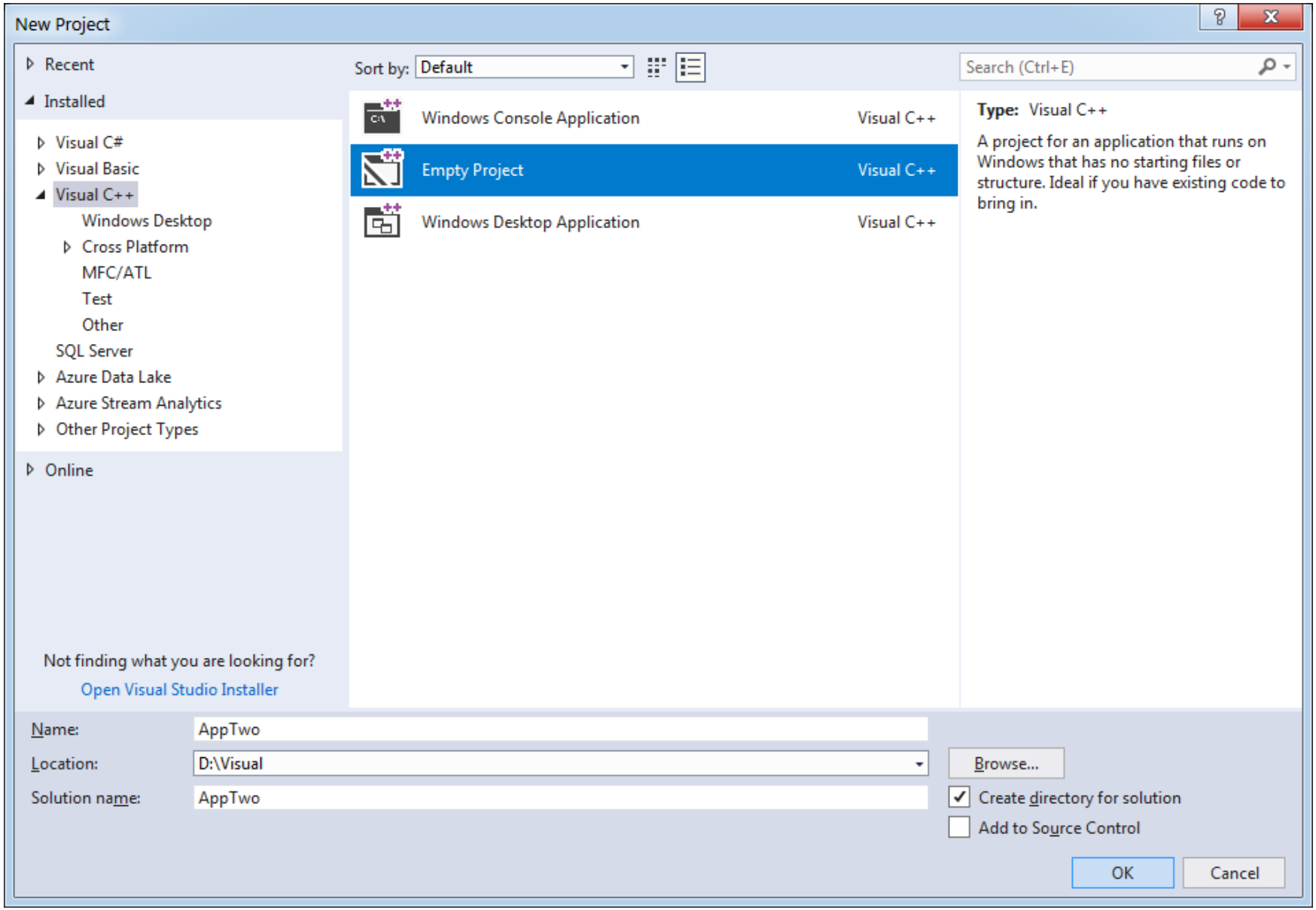
Plik *.cpp – plik *źródłowy*, zawiera kod źródłowy programu.

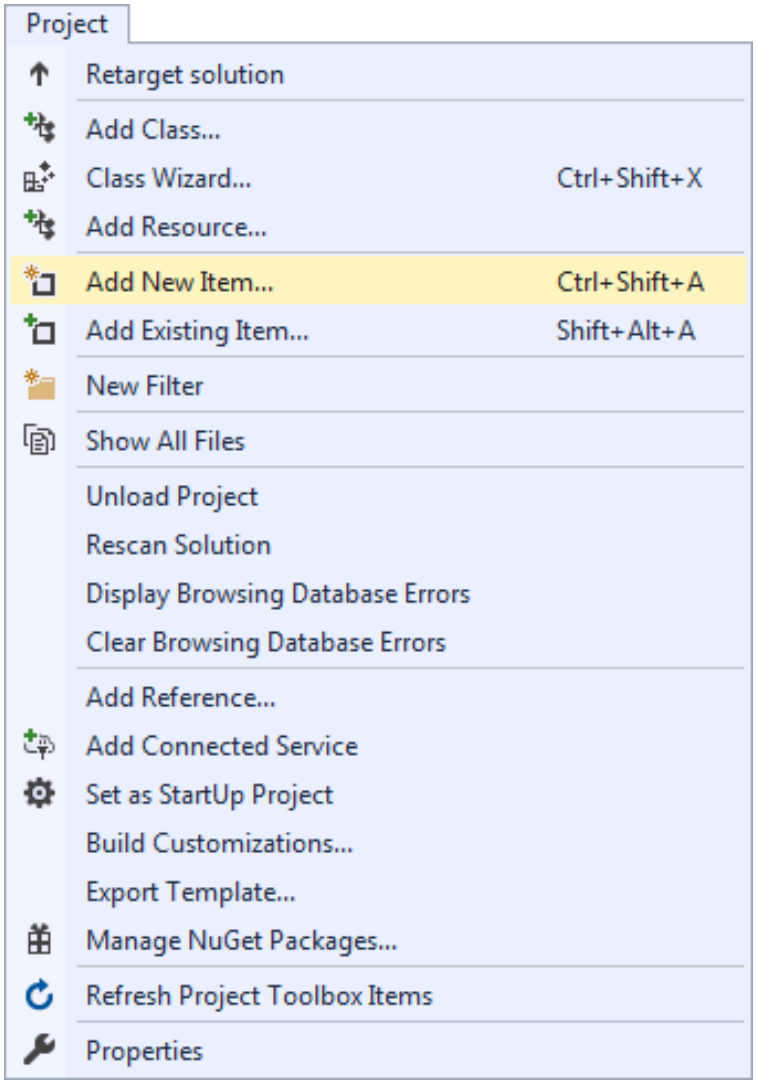
Lister - [D:\Visual\AppOne\AppOne\AppOne.cpp]

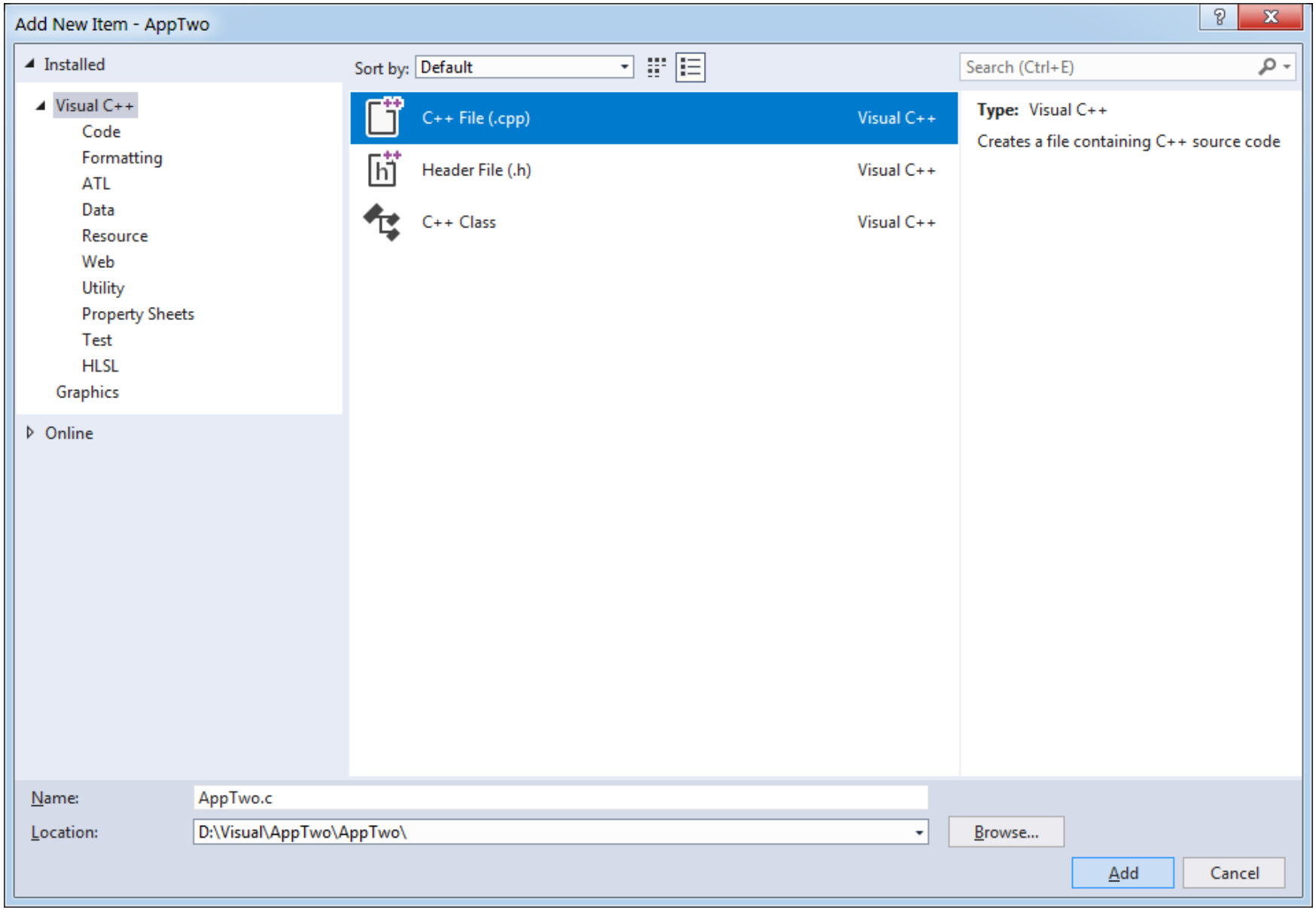
File Edit Options Encoding Help 100 %

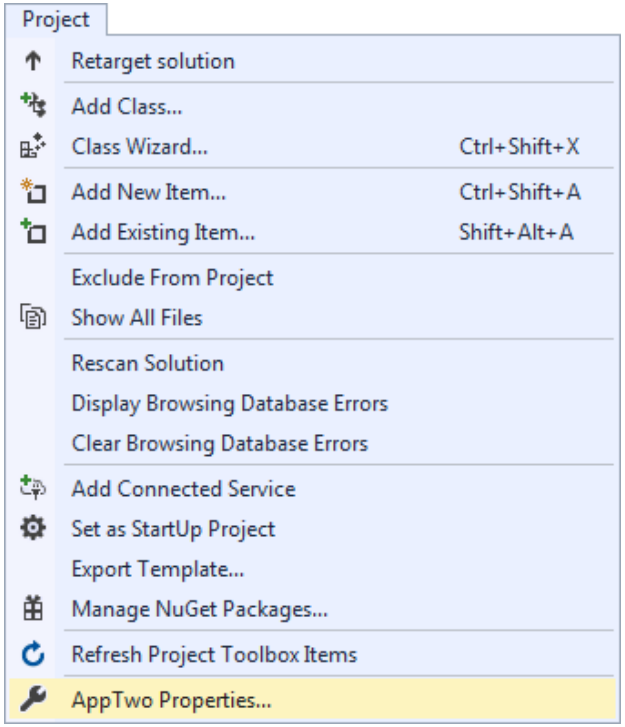
```
// AppOne.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

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

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// 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
```

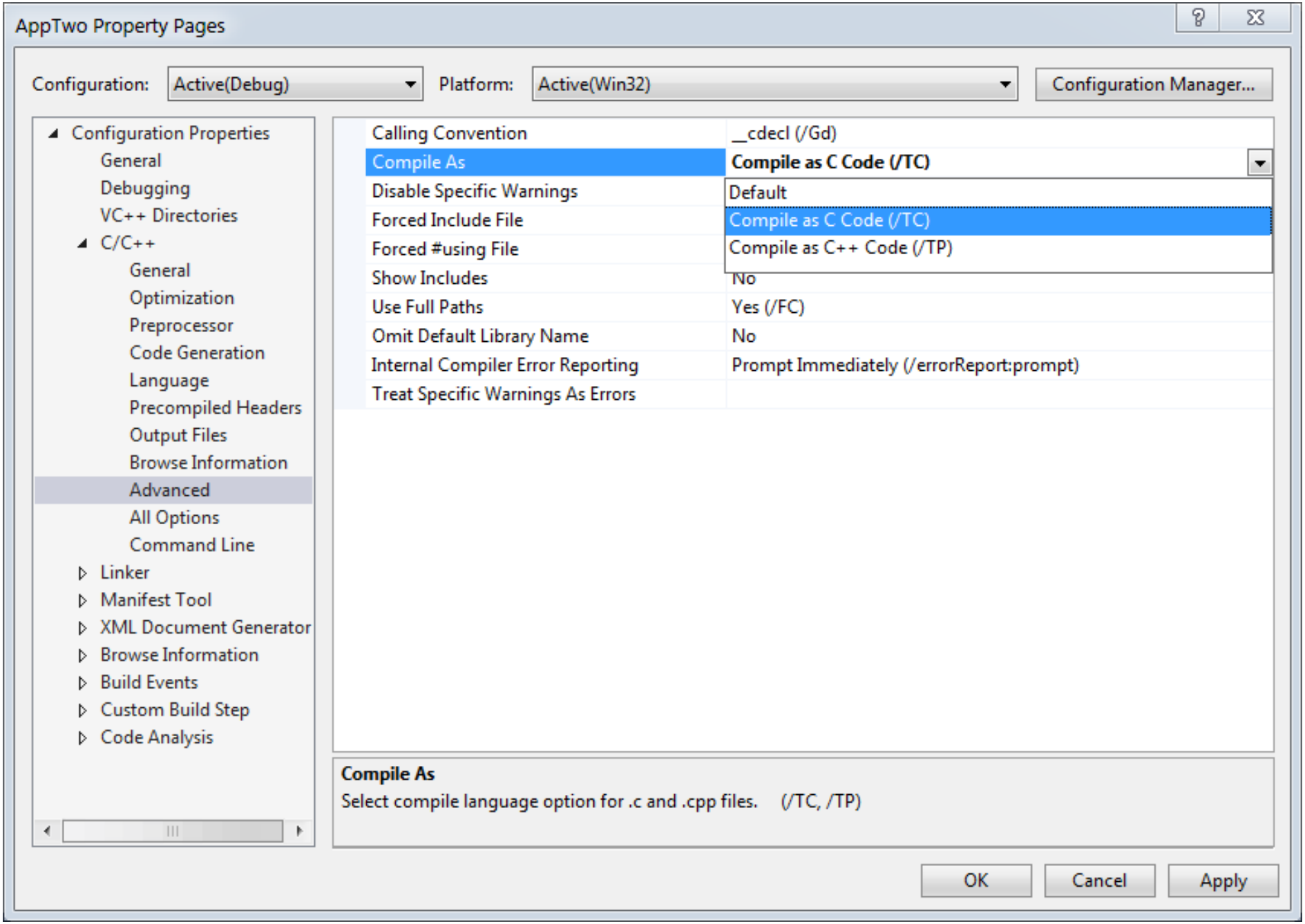
Projekt w języku C

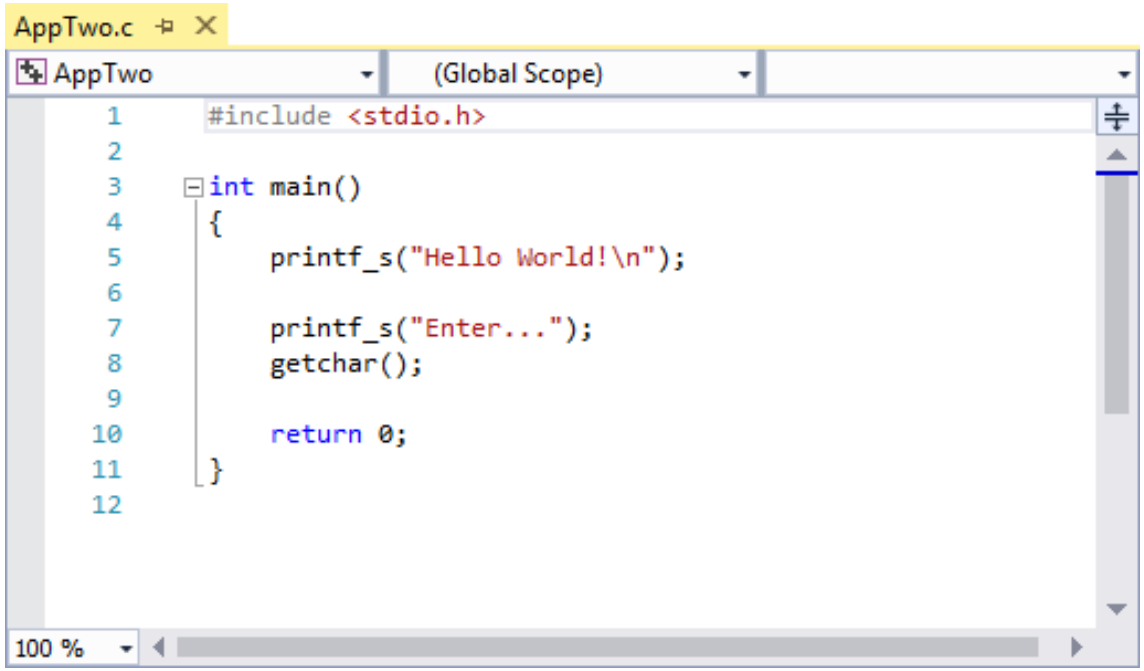



Project
Retarget solution
Add Class...
Class Wizard... Ctrl+Shift+X
Add Resource...
Add New Item... Ctrl+Shift+A
Add Existing Item... Shift+Alt+A
New Filter
Show All Files
Unload Project
Rescan Solution
Display Browsing Database Errors
Clear Browsing Database Errors
Add Reference...
Add Connected Service
Set as StartUp Project
Build Customizations...
Export Template...
Manage NuGet Packages...
Refresh Project Toolbox Items
Properties


Add New Item - AppTwo
Installed
Visual C++
Code
Formatting
ATL
Data
Resource
Web
Utility
Property Sheets
Test
HLSL
Graphics
Online
Sort by: Default
Search (Ctrl+E)
C++ File (.cpp) Visual C++
Header File (.h) Visual C++
C++ Class Visual C++
Type: Visual C++
Creates a file containing C++ source code
Name: AppTwo.c
Location: D:\Visual\AppTwo\AppTwo\
Browse...
Add
Cancel

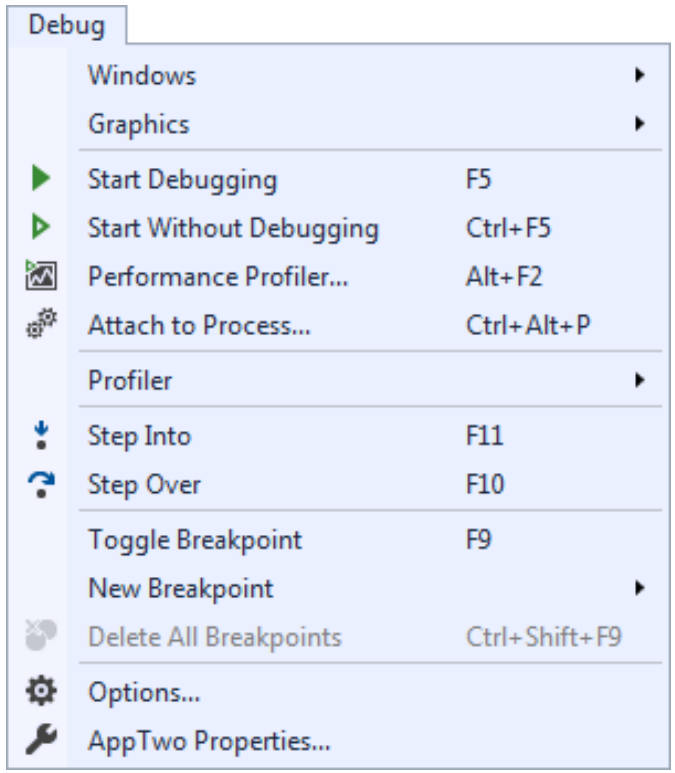

Project
Retarget solution
Add Class...
Class Wizard... Ctrl+Shift+X
Add New Item... Ctrl+Shift+A
Add Existing Item... Shift+Alt+A
Exclude From Project
Show All Files
Rescan Solution
Display Browsing Database Errors
Clear Browsing Database Errors
Add Connected Service
Set as StartUp Project
Export Template...
Manage NuGet Packages...
Refresh Project Toolbox Items
AppTwo Properties...

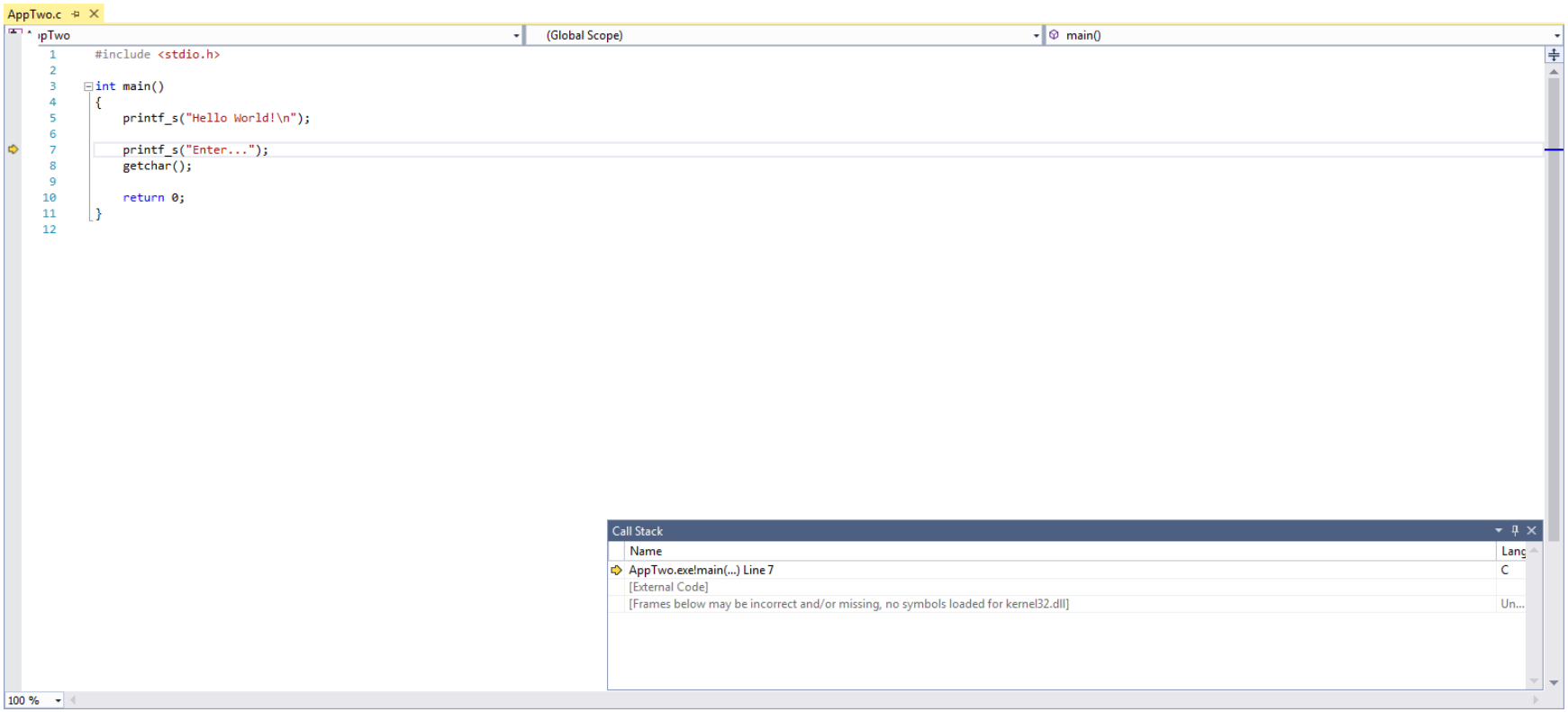

AppTwo Property Pages
Configuration: Active(Debug)
Platform: Active(Win32)
Configuration Manager...
Configuration Properties
General
Debugging
VC++ Directories
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
All Options
Command Line
Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Code Analysis
Calling Convention
__cdecl (/Gd)
Compile As
Compile as C Code (/TC)
Disable Specific Warnings
Default
Forced Include File
Compile as C Code (/TC)
Forced #using File
Compile as C++ Code (/TP)
Show Includes
No
Use Full Paths
Yes (/FC)
Omit Default Library Name
No
Internal Compiler Error Reporting
Prompt Immediately (/errorReport:prompt)
Treat Specific Warnings As Errors
Compile As
Select compile language option for .c and .cpp files. (/TC, /TP)
OK
Cancel
Apply

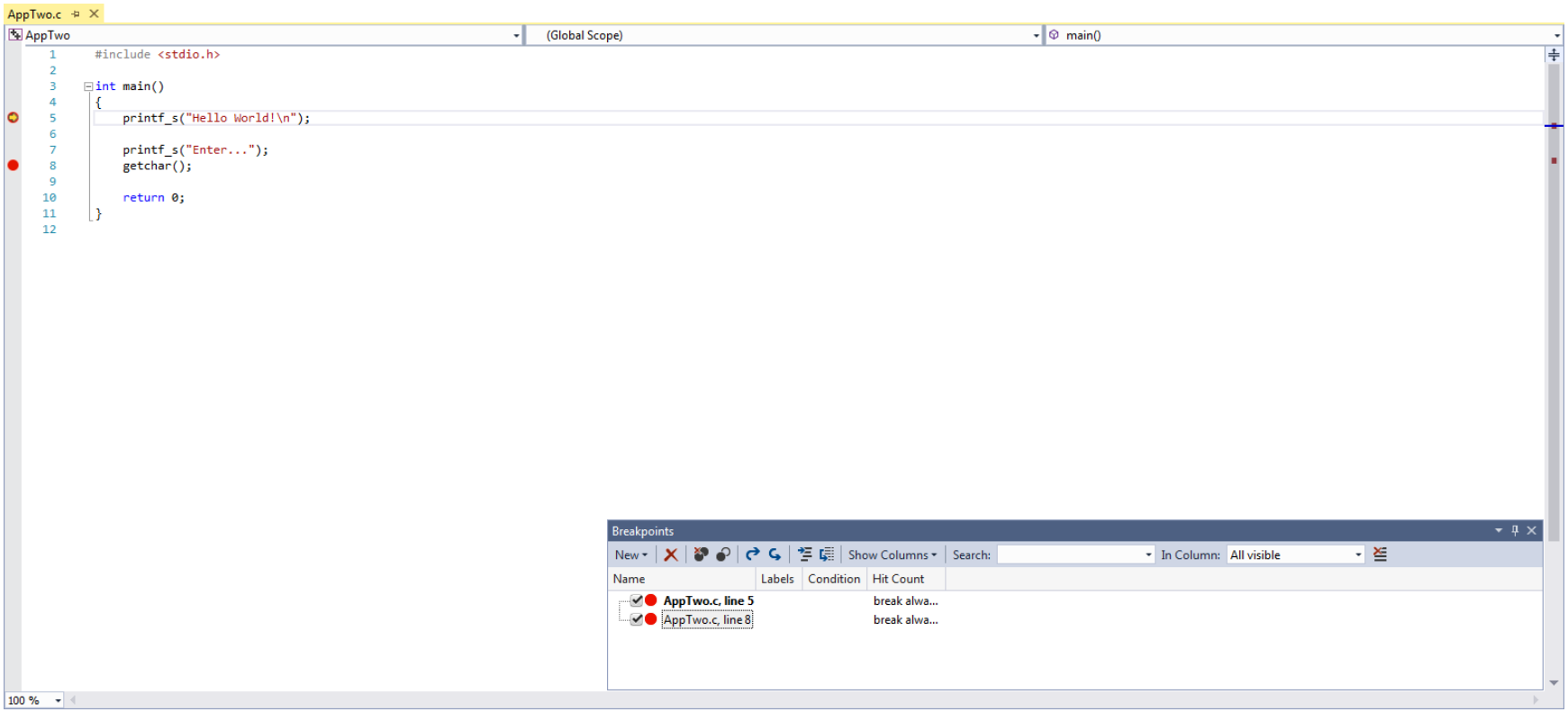

AppTwo.c
AppTwo
(Global Scope)
#include <stdio.h>
int main()
{
printf_s("Hello World!\n");
printf_s("Enter...");
getchar();
return 0;
}
100 %

Kompilowanie, debugowanie, śledzenie

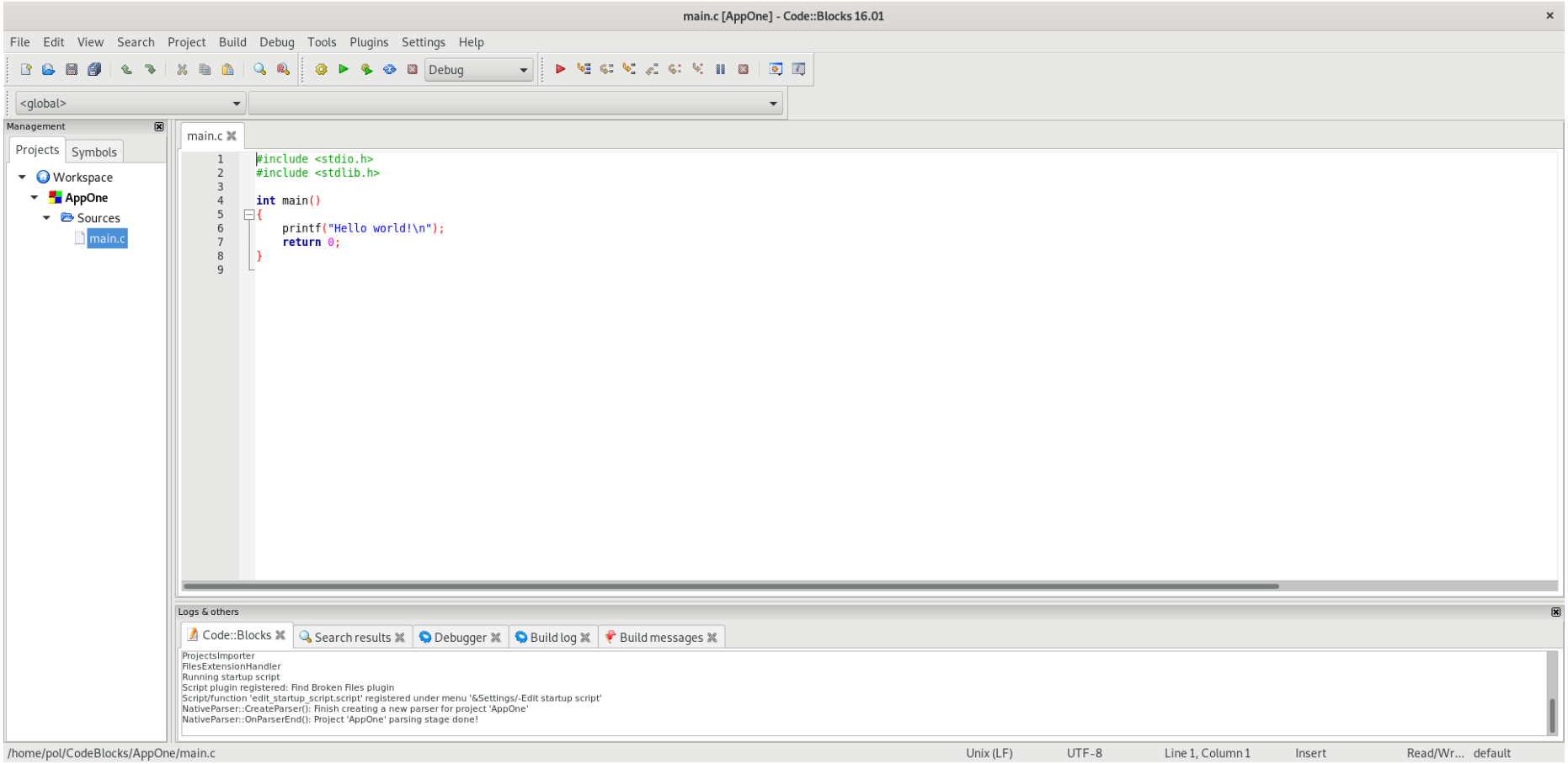
skróty klawiaturowe	
[F5]	start z nadzorowaniem (debugger) i kontynuacja wykonywania programu
[CTRL F5]	start bez debuggera
[F11]	krokowe wykonywanie programu
[F10]	krokowe wykonywanie programu, procedury i funkcje w jednym kroku
[F9]	wstawianie i usuwanie punktów przerwania (*Break Point*)



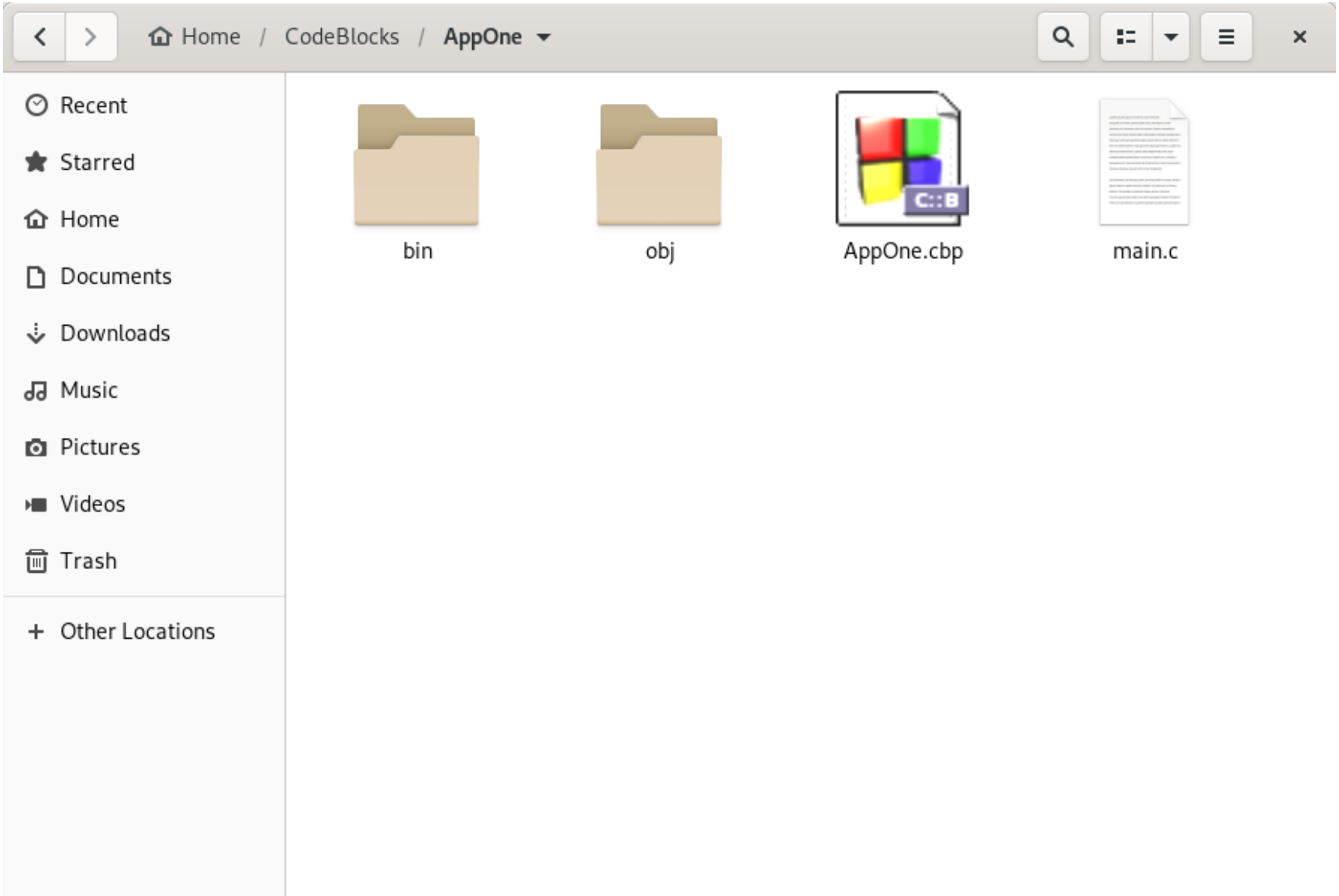

AppTwo.c
pTwo
(Global Scope)
main()
#include <stdio.h>
int main()
{
printf_s("Hello World!\n");
printf_s("Enter...");
getchar();
return 0;
}
Call Stack
Name
Lang
AppTwo.exe!main(...) Line 7
C
[External Code]
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dll]
Un...
100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

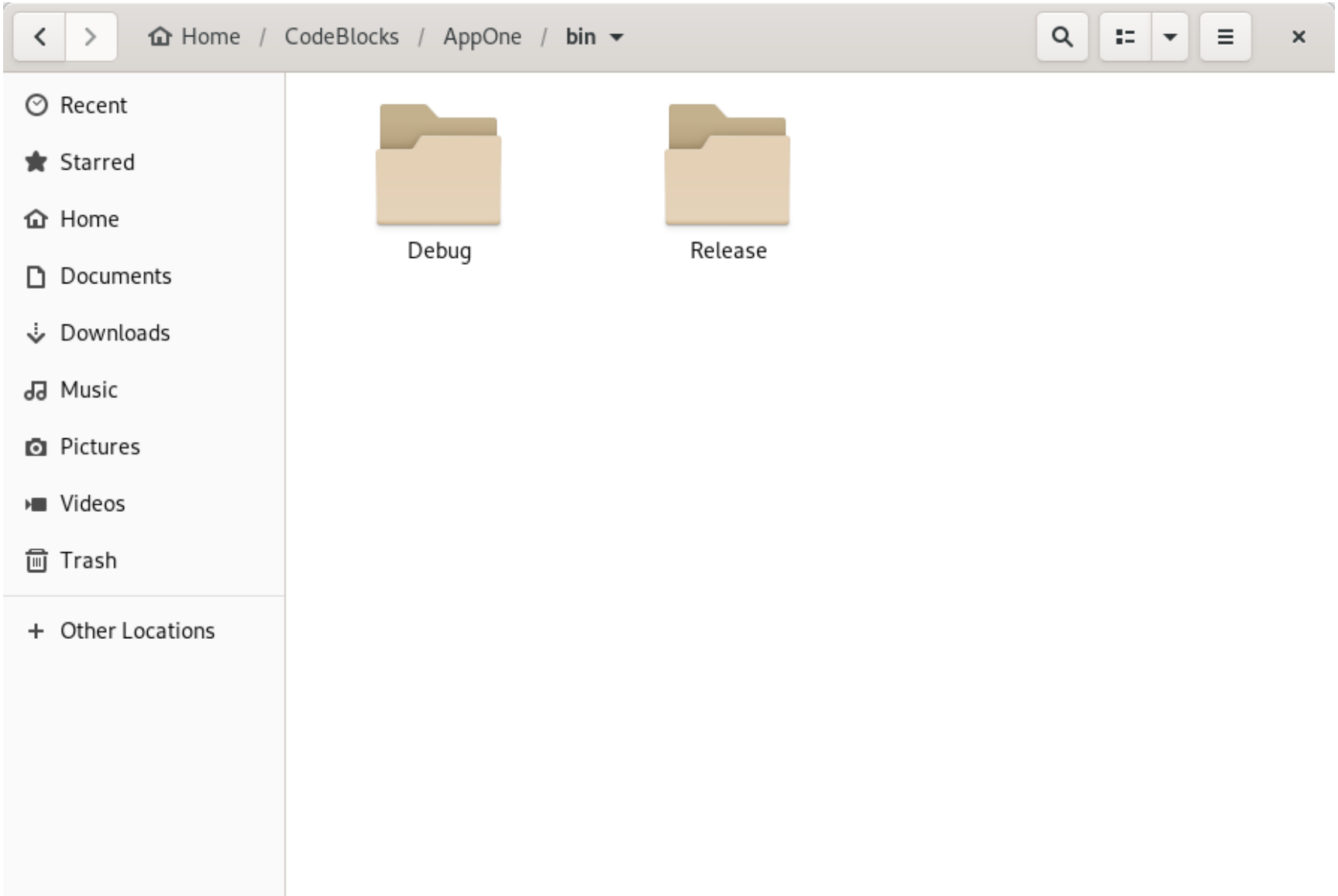

AppTwo.c
AppTwo
(Global Scope)
main()
#include <stdio.h>
int main()
{
printf_s("Hello World!\n");
printf_s("Enter...");
getchar();
return 0;
}
Breakpoints
New
Show Columns
Search:
In Column:
All visible
Name
Labels
Condition
Hit Count
AppTwo.c, line 5
break alwa...
AppTwo.c, line 8
break alwa...
100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

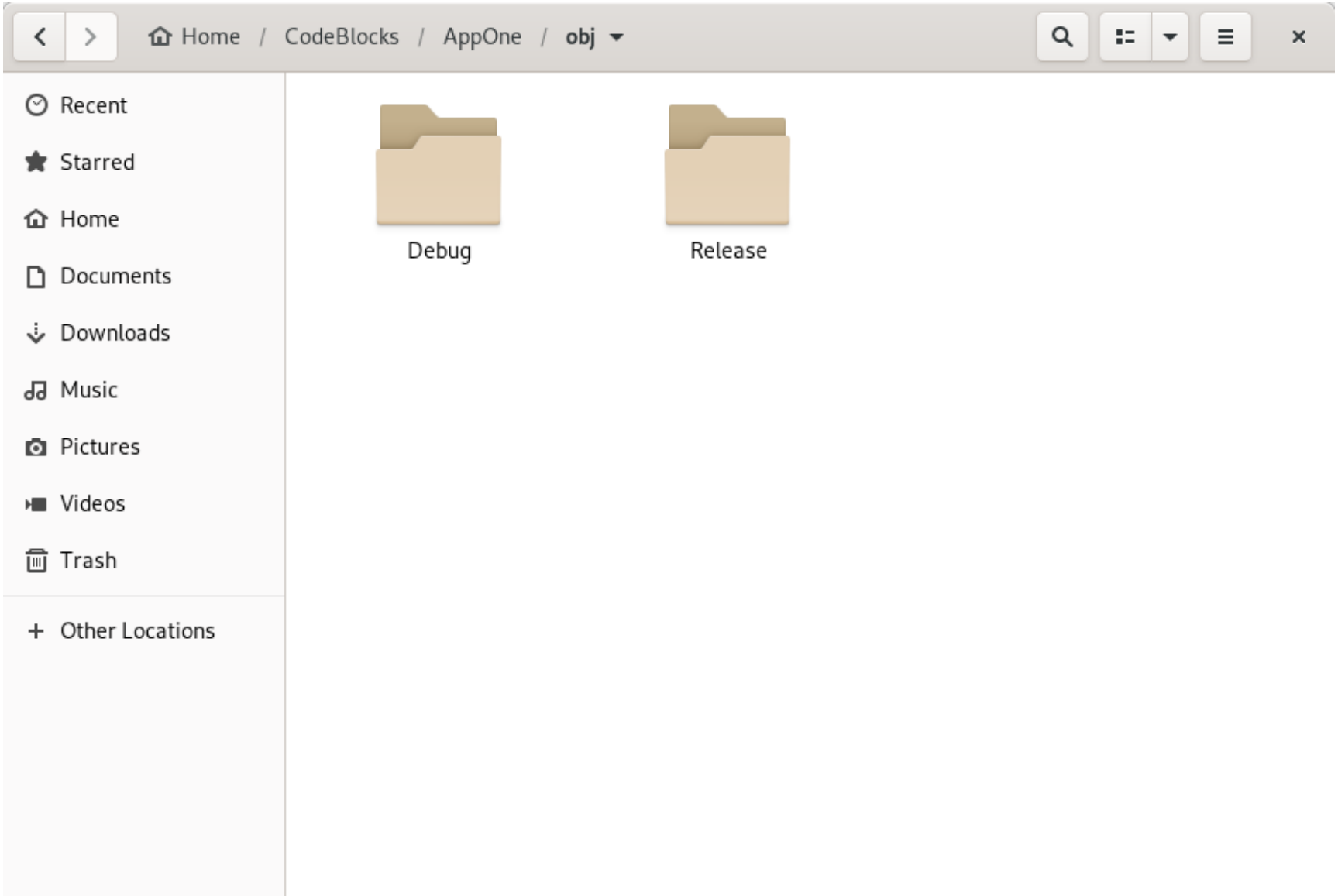
Środowisko programowania - Linux

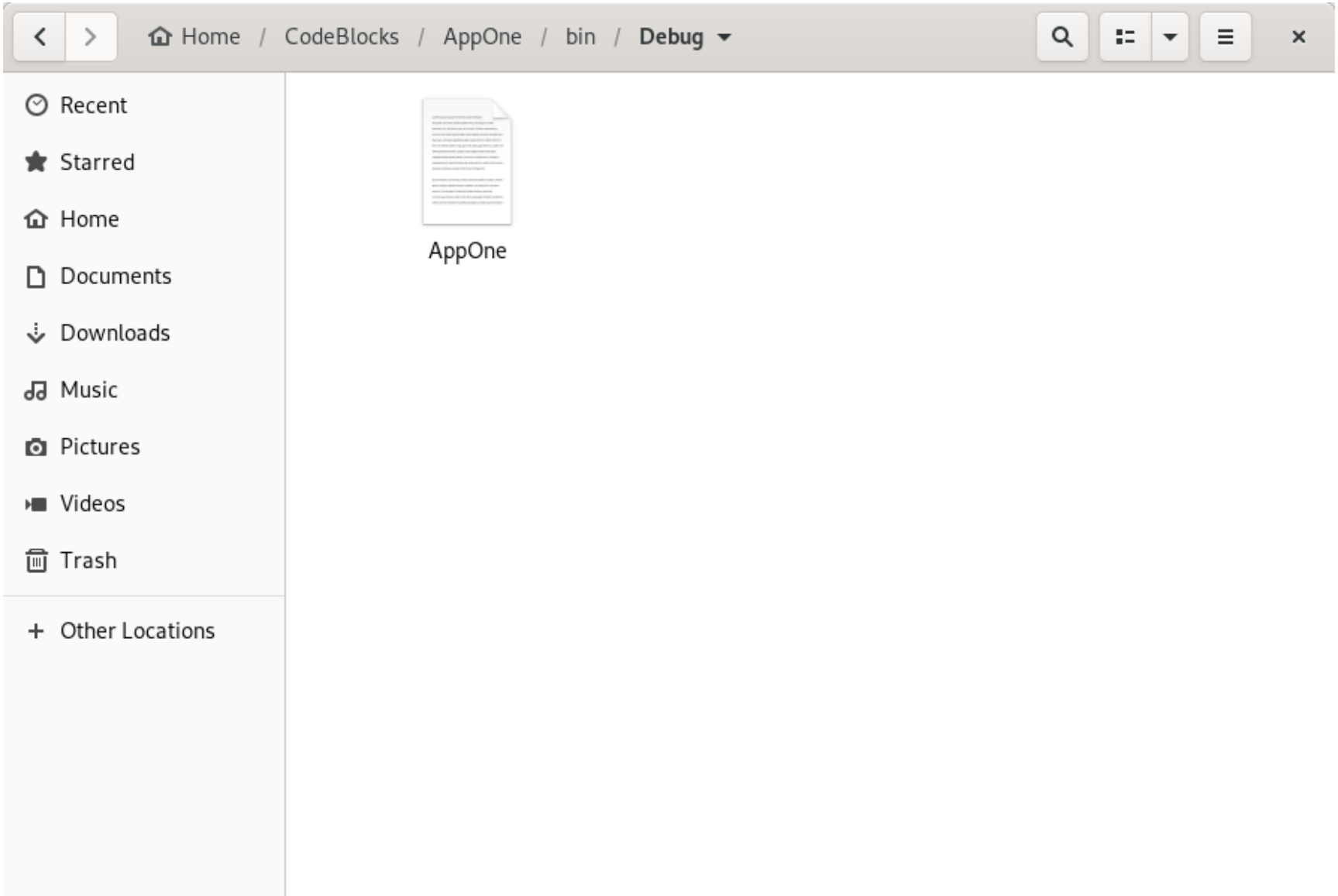


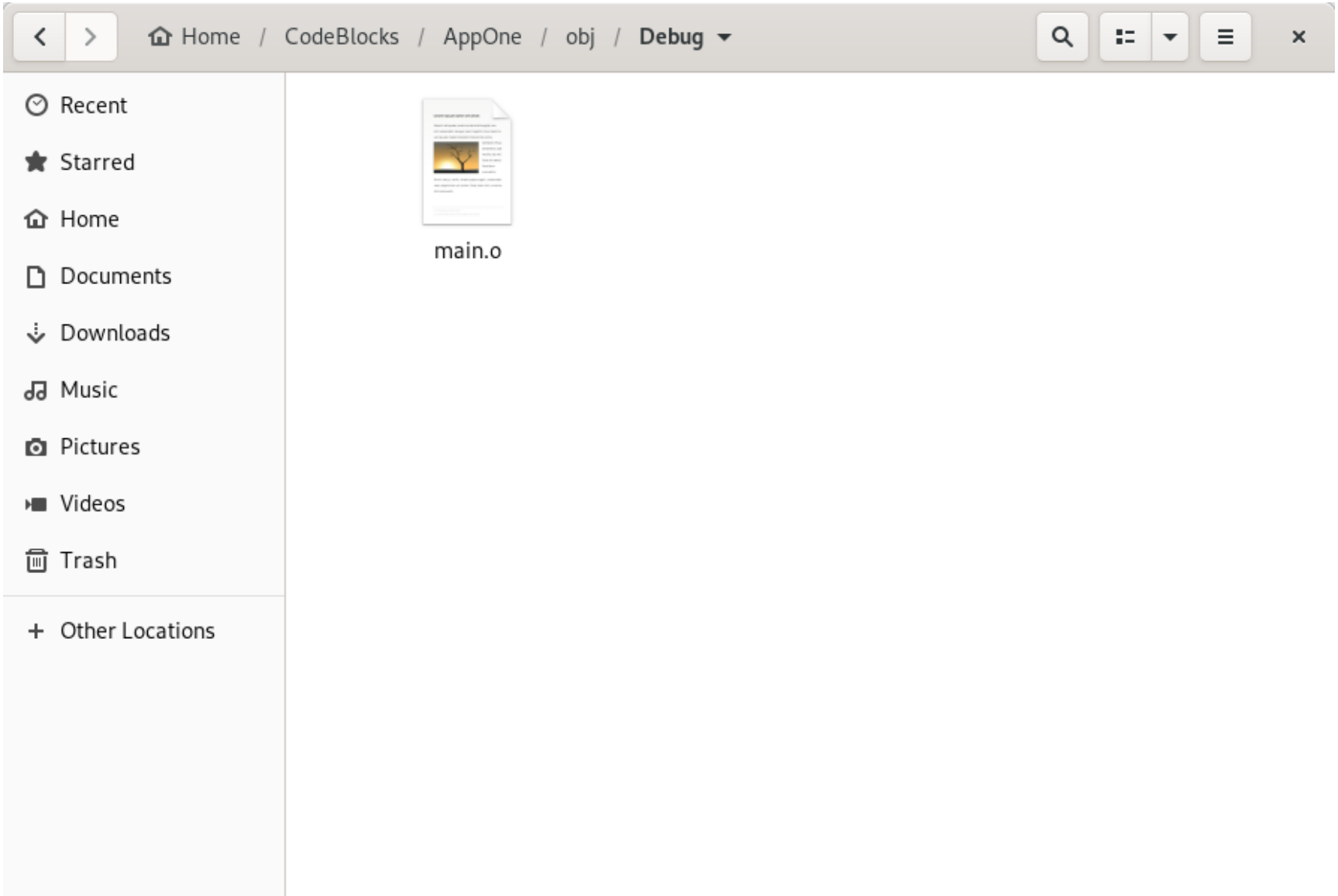
Pliki projektu




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.

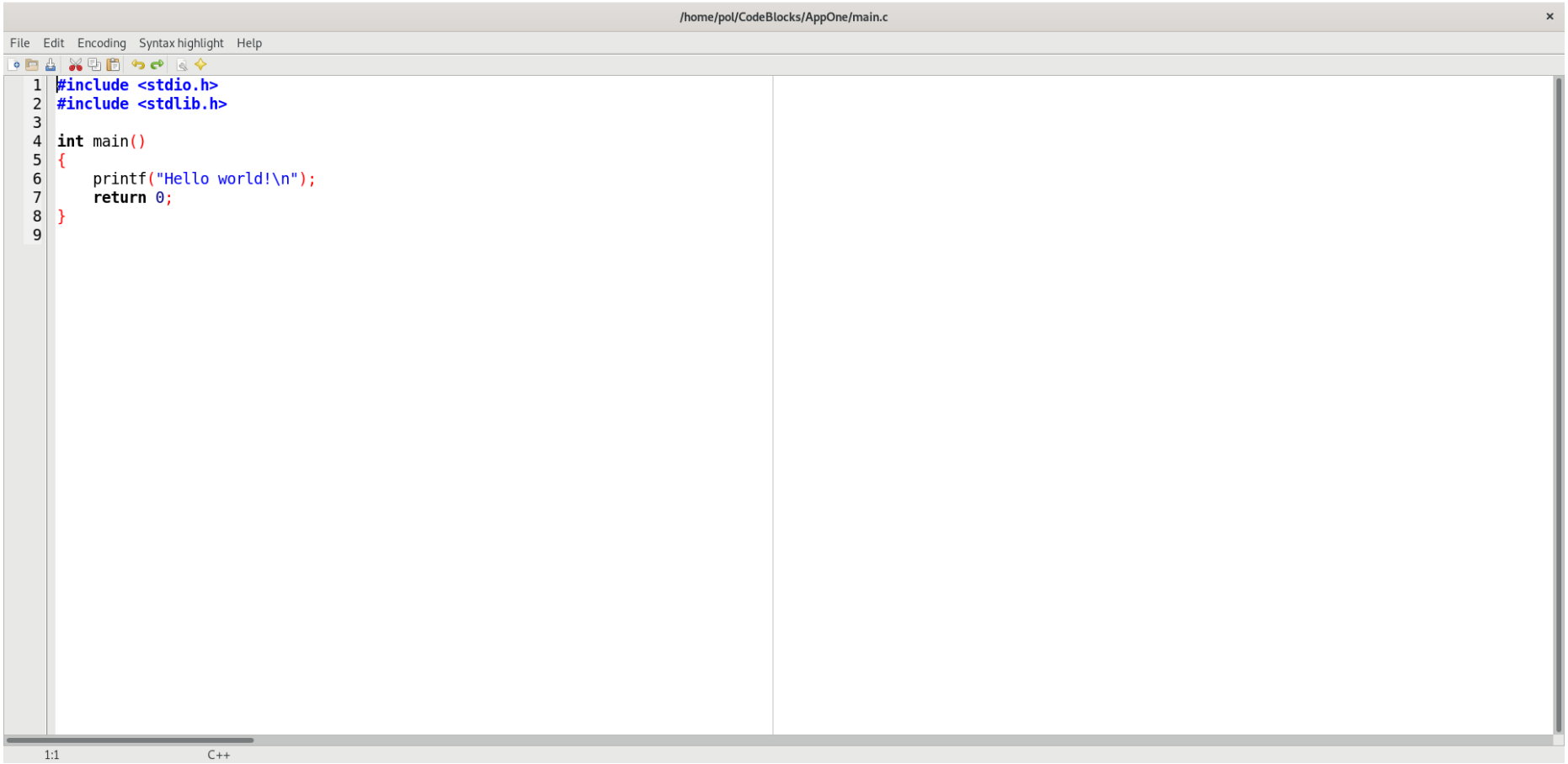
/home/pol/CodeBlocks/AppOne/AppOne.cbp

File Edit Encoding Syntax highlight Help

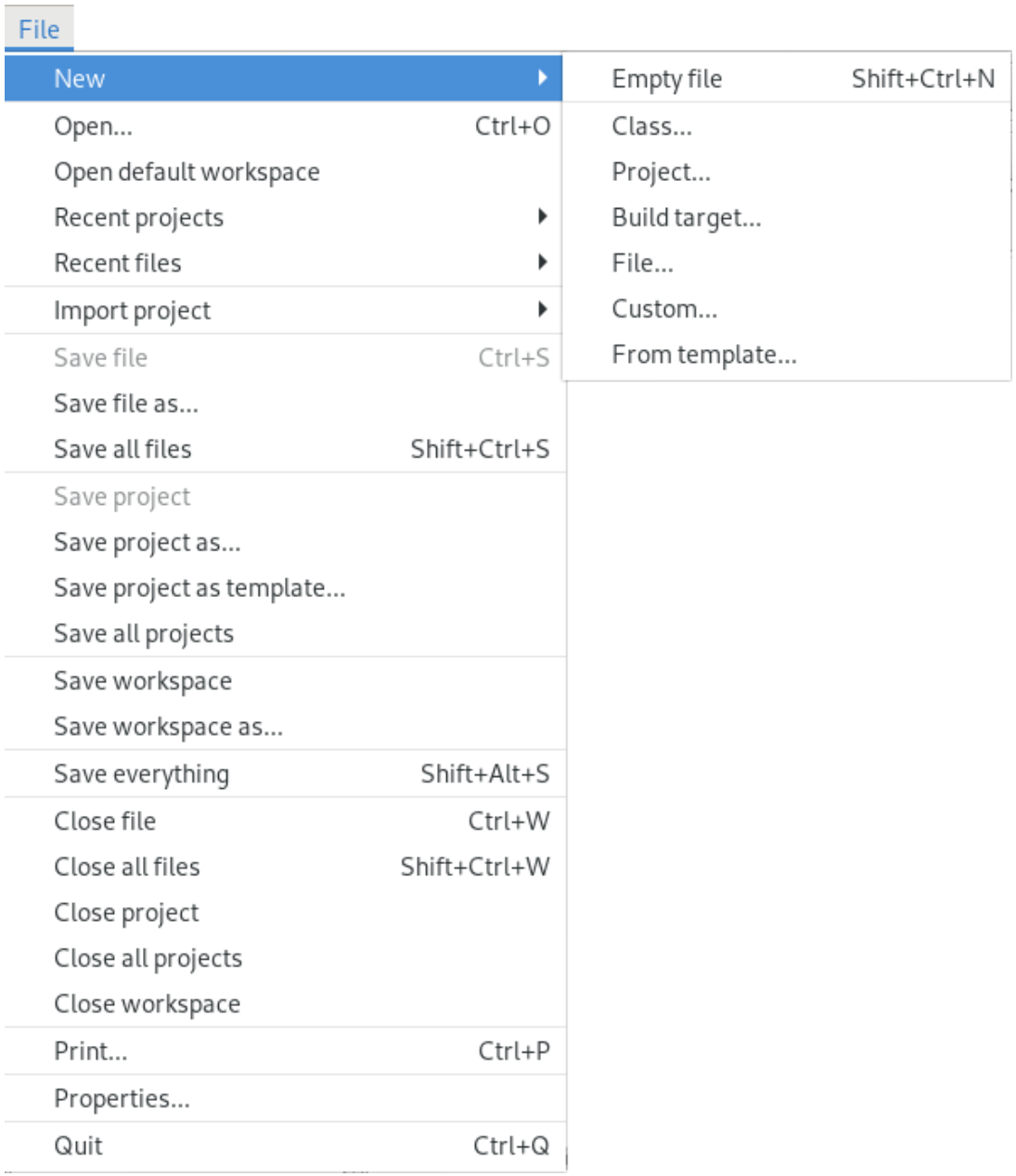
```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<CodeBlocks_project_file>
        <FileVersion major="1" minor="6" />
        <Project>
                <Option title="AppOne" />
                <Option pch_mode="2" />
                <Option compiler="gcc" />
                <Build>
                        <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" />
```

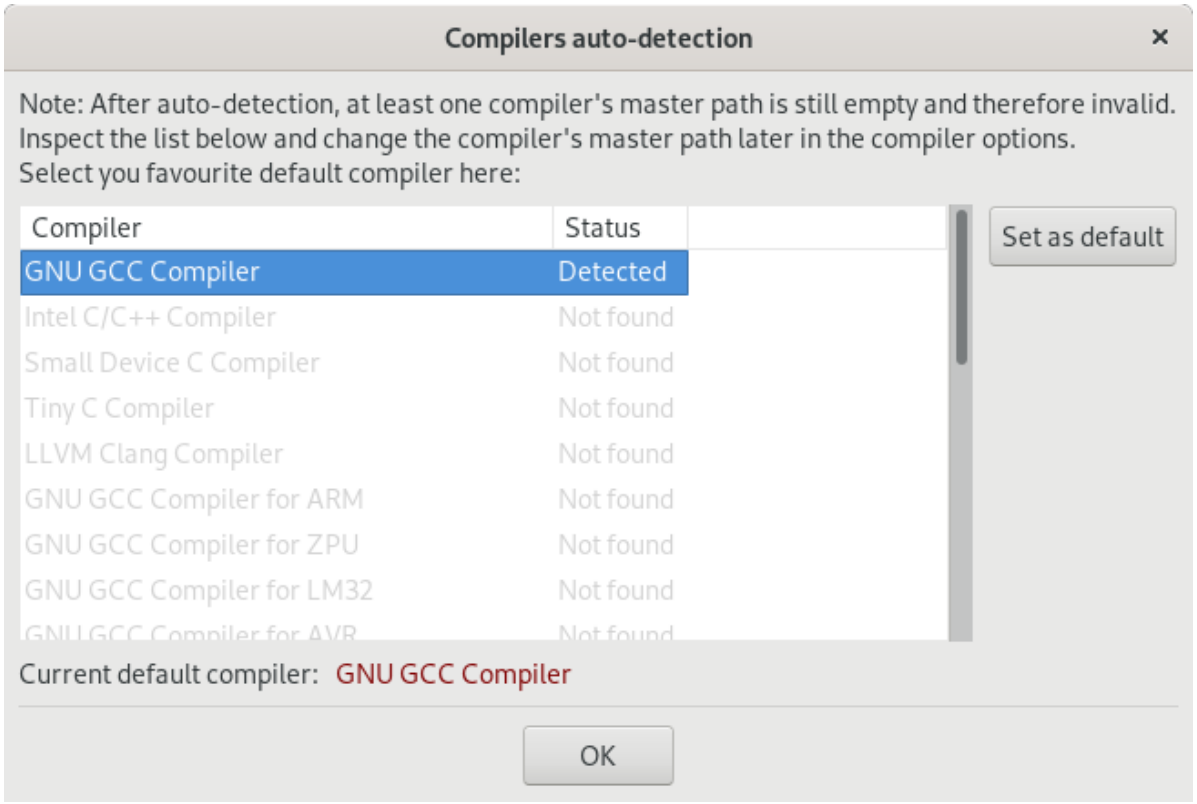
1:1 XML document

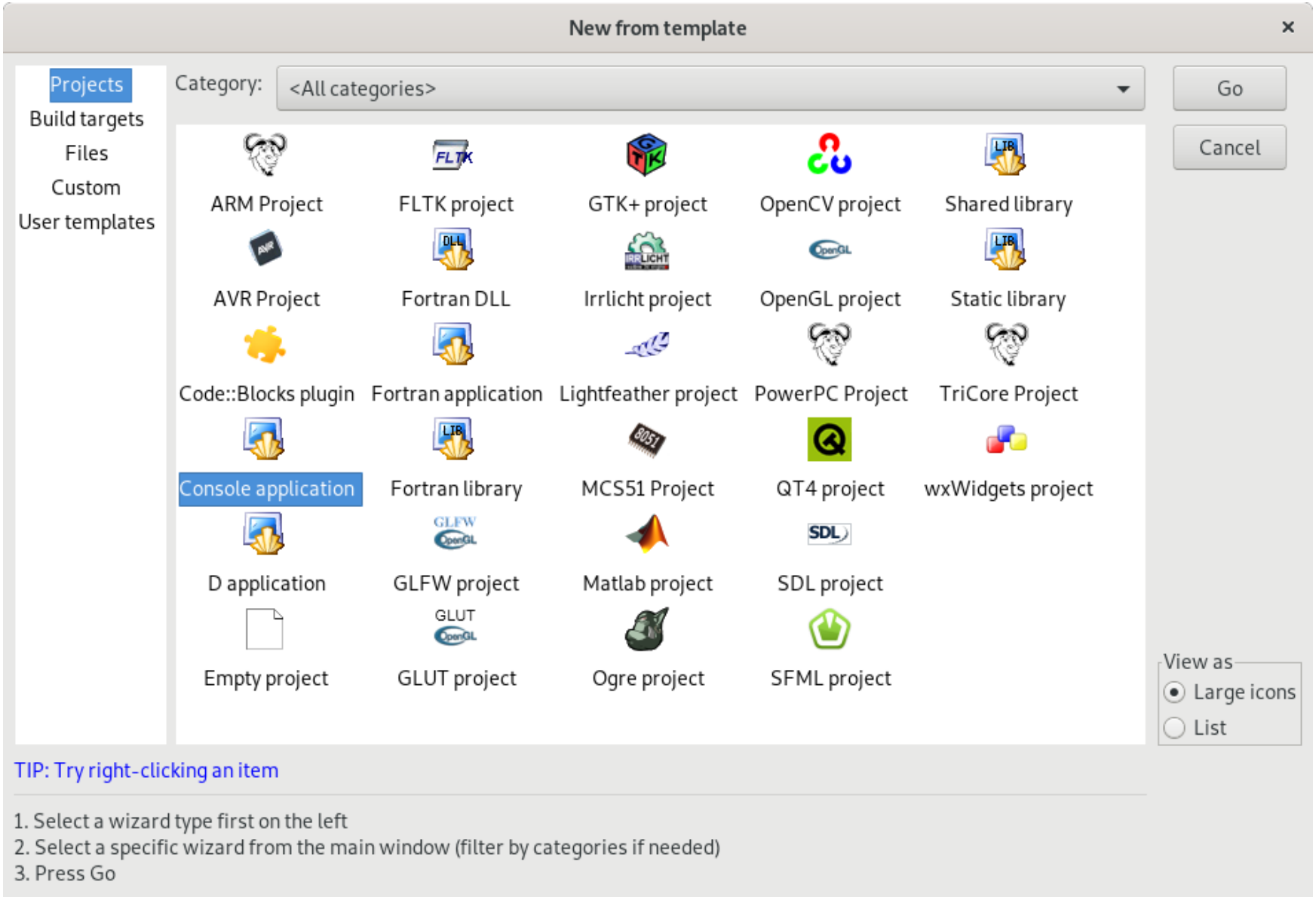
Plik *.cpp – plik *źródłowy*, zawiera kod źródłowy programu.

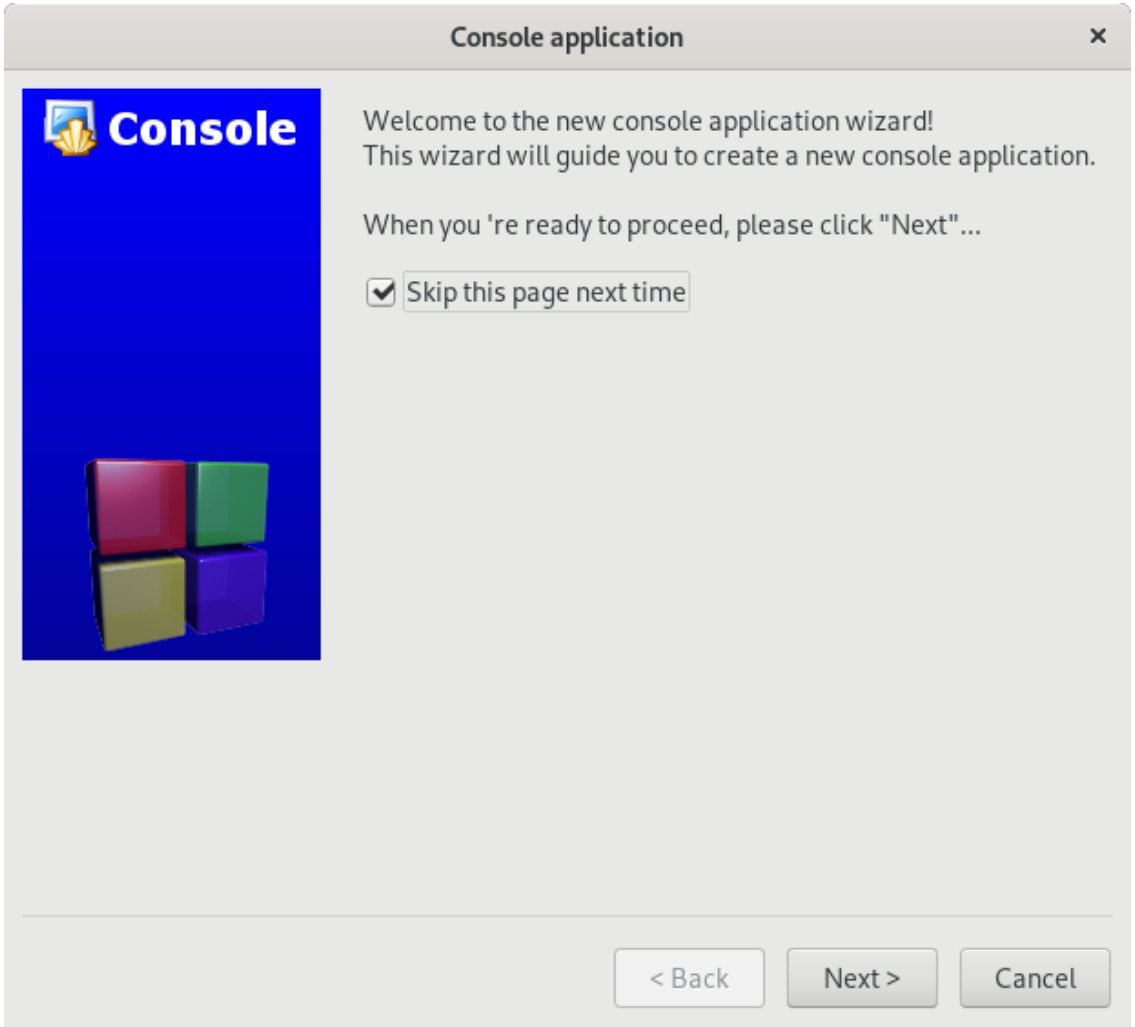


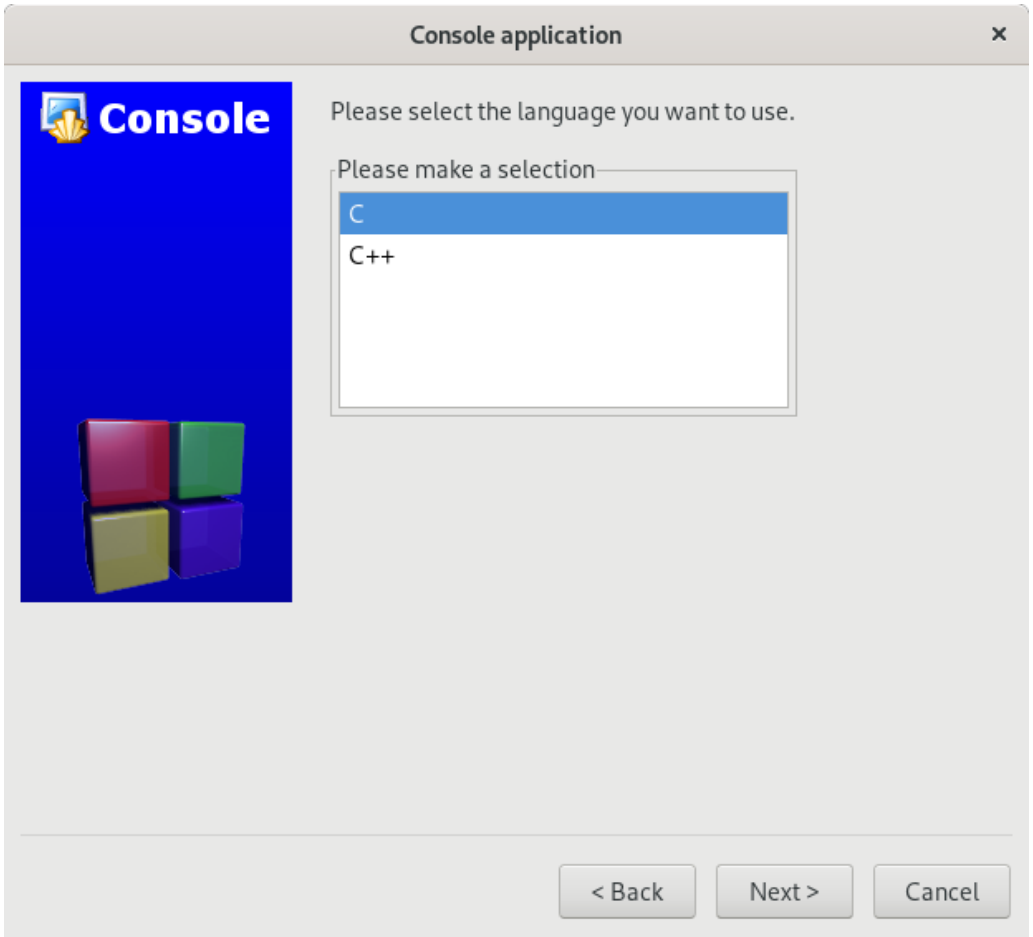
Projekt w języku C

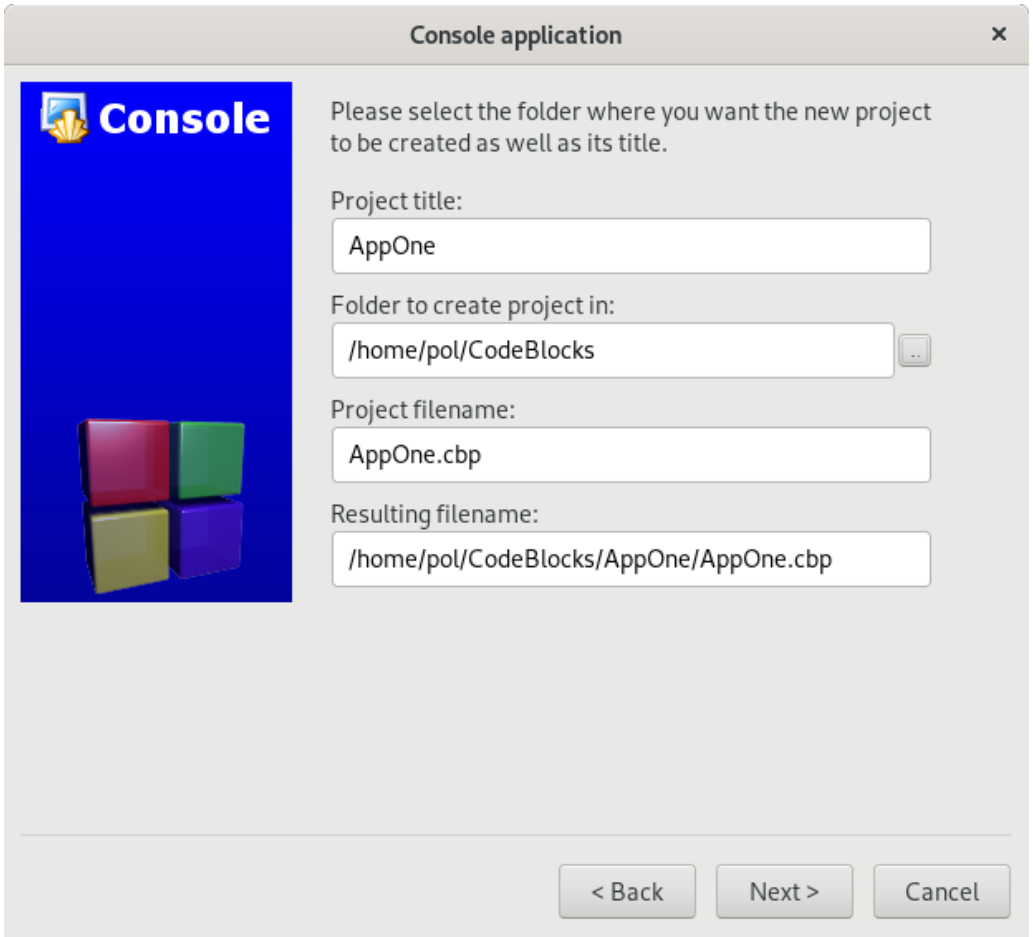


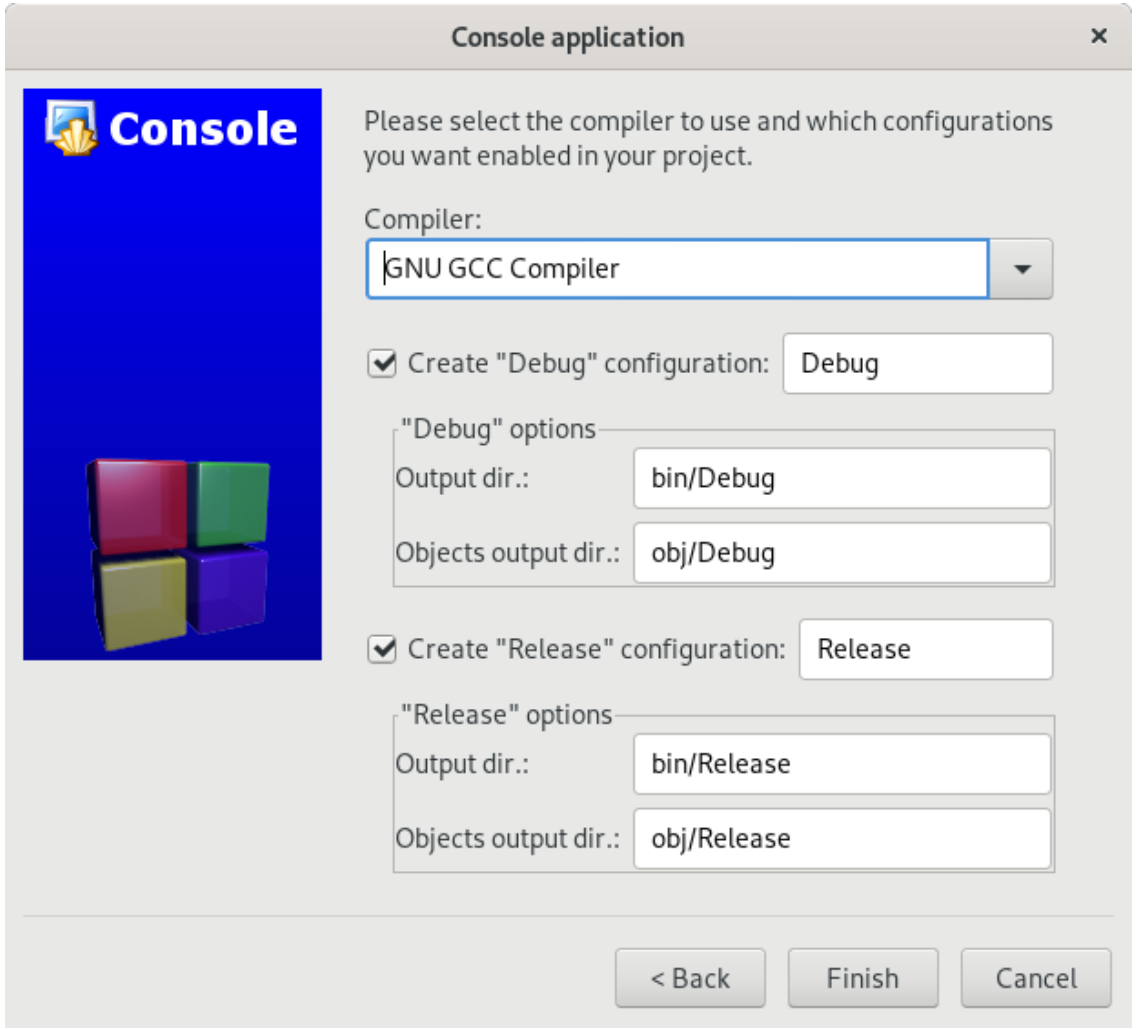

Compilers auto-detection
Note: After auto-detection, at least one compiler's master path is still empty and therefore invalid.
Inspect the list below and change the compiler's master path later in the compiler options.
Select you favourite default compiler here:
Compiler
Status
GNU GCC Compiler
Detected
Intel C/C++ Compiler
Not found
Small Device C Compiler
Not found
Tiny C Compiler
Not found
LLVM Clang Compiler
Not found
GNU GCC Compiler for ARM
Not found
GNU GCC Compiler for ZPU
Not found
GNU GCC Compiler for LM32
Not found
Set as default
Current default compiler: GNU GCC Compiler
OK


New from template
Projects
Build targets
Files
Custom
User templates
Category:
<All categories>
Go
Cancel
ARM Project
FLTK project
GTK+ project
OpenCV project
Shared library
AVR Project
Fortran DLL
Irrlicht project
OpenGL project
Static library
Code::Blocks plugin
Fortran application
Lightfeather project
PowerPC Project
TriCore Project
Console application
Fortran library
MCS51 Project
QT4 project
wxWidgets project
D application
GLFW project
Matlab project
SDL project
Empty project
GLUT project
Ogre project
SFML project
View as
Large icons
List
TIP: Try right-clicking an item
1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go


Console application
Console
Welcome to the new console application wizard!
This wizard will guide you to create a new console application.
When you 're ready to proceed, please click "Next"...
Skip this page next time
< Back
Next >
Cancel


Console application
Console
Please select the language you want to use.
Please make a selection
C
C++
< Back
Next >
Cancel


Console application
Console
Please select the folder where you want the new project to be created as well as its title.
Project title:
AppOne
Folder to create project in:
/home/pol/CodeBlocks
Project filename:
AppOne.cbp
Resulting filename:
/home/pol/CodeBlocks/AppOne/AppOne.cbp
< Back
Next >
Cancel

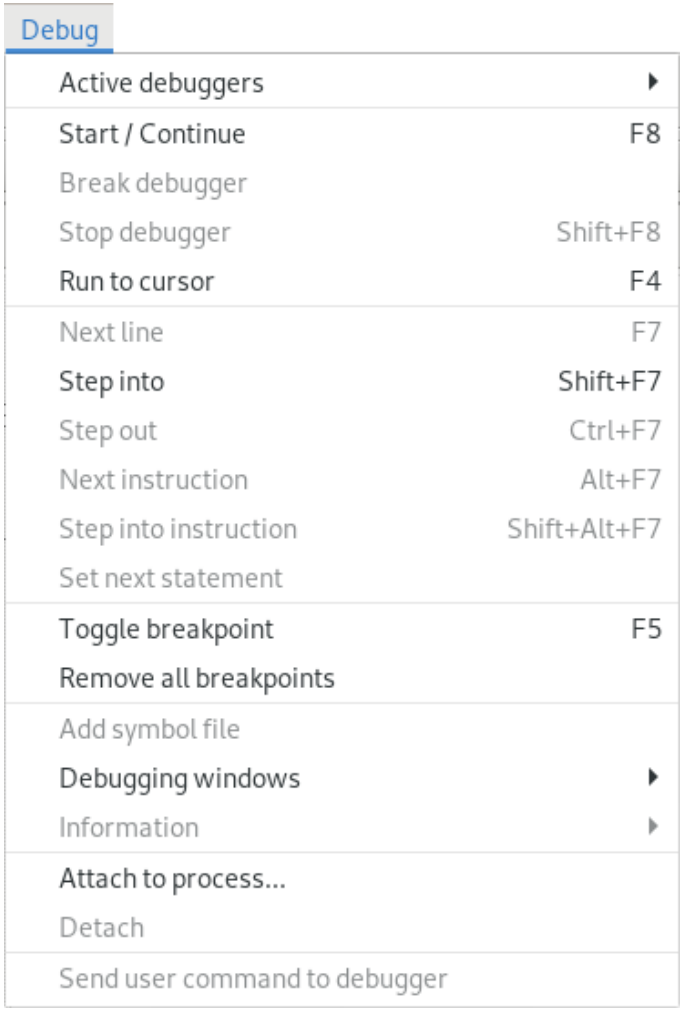

Console application
Console
Please select the compiler to use and which configurations you want enabled in your project.
Compiler:
GNU GCC Compiler
Create "Debug" configuration:
Debug
"Debug" options
Output dir.:
bin/Debug
Objects output dir.:
obj/Debug
Create "Release" configuration:
Release
"Release" options
Output dir.:
bin/Release
Objects output dir.:
obj/Release
< Back
Finish
Cancel

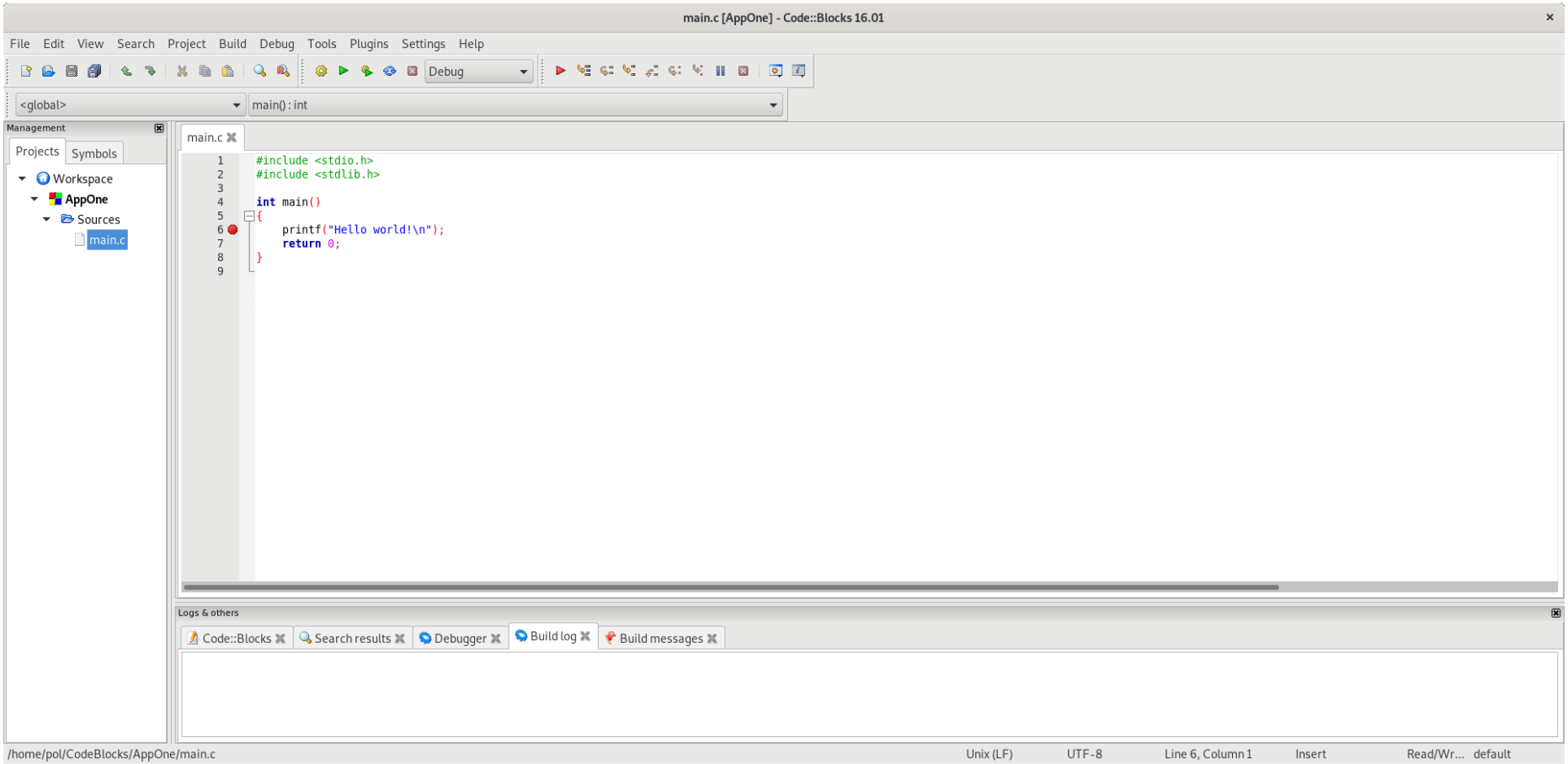
main.c

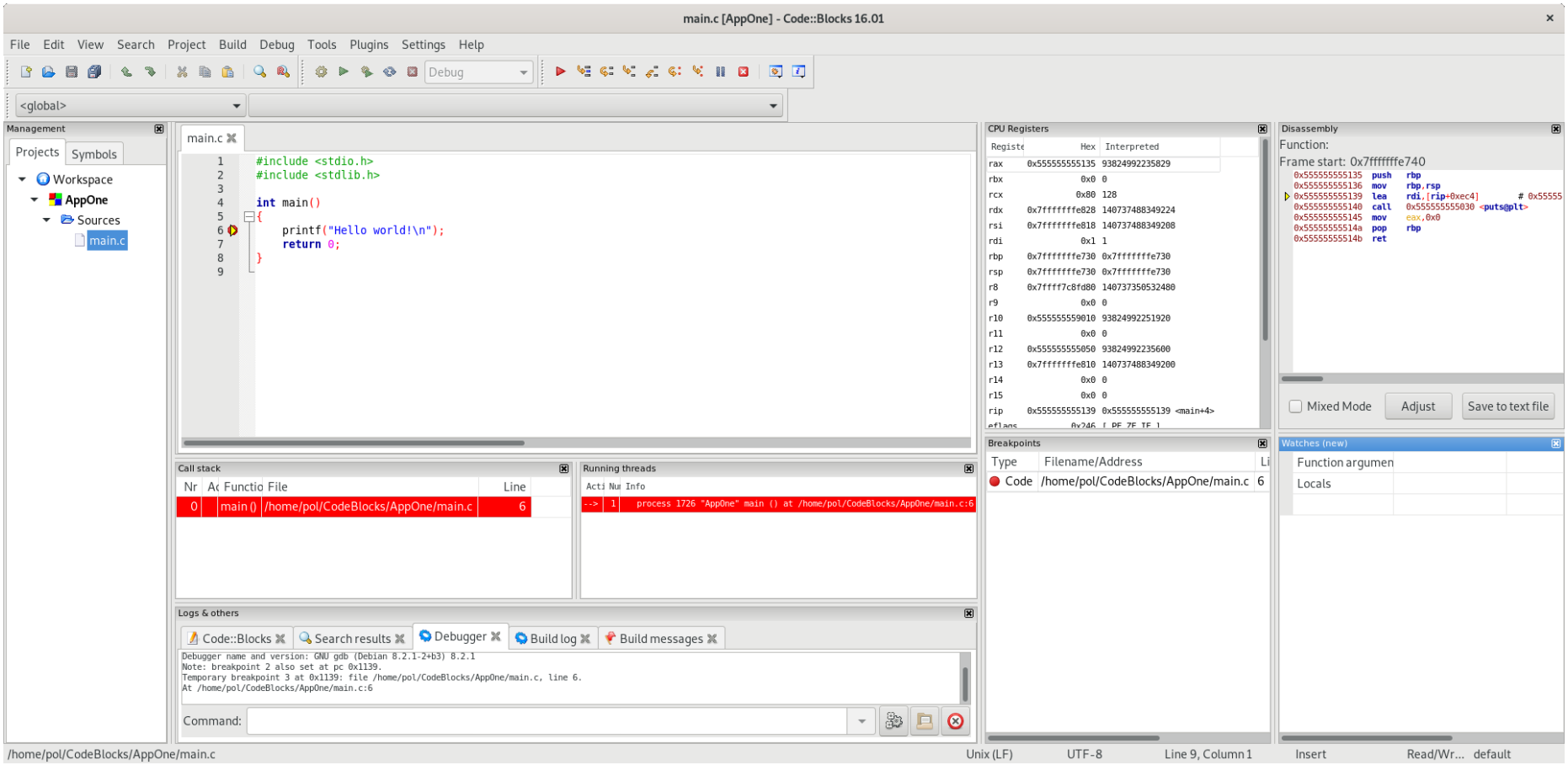
```
1  #include <stdio.h>
2
3
4  int main()
5  {
6      printf("Hello world!\n");
7
8      printf("Enter...");
9      getchar();
10
11     return 0;
12 }
13
```

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
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("Hello world!\n");
return 0;
}
CPU Registers
Disassembly
Function:
Frame start: 0x7fffffffe740
Mixed Mode
Adjust
Save to text file
Breakpoints
Type
Filename/Address
Code
/home/pol/CodeBlocks/AppOne/main.c
Watches (new)
Function argumen
Locals
Call stack
Nr
File
Line
0
main ()
/home/pol/CodeBlocks/AppOne/main.c
6
Running threads
process 1726 "AppOne" main () at /home/pol/CodeBlocks/AppOne/main.c:6
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/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.

Co i jak wywołuje funkcję main?

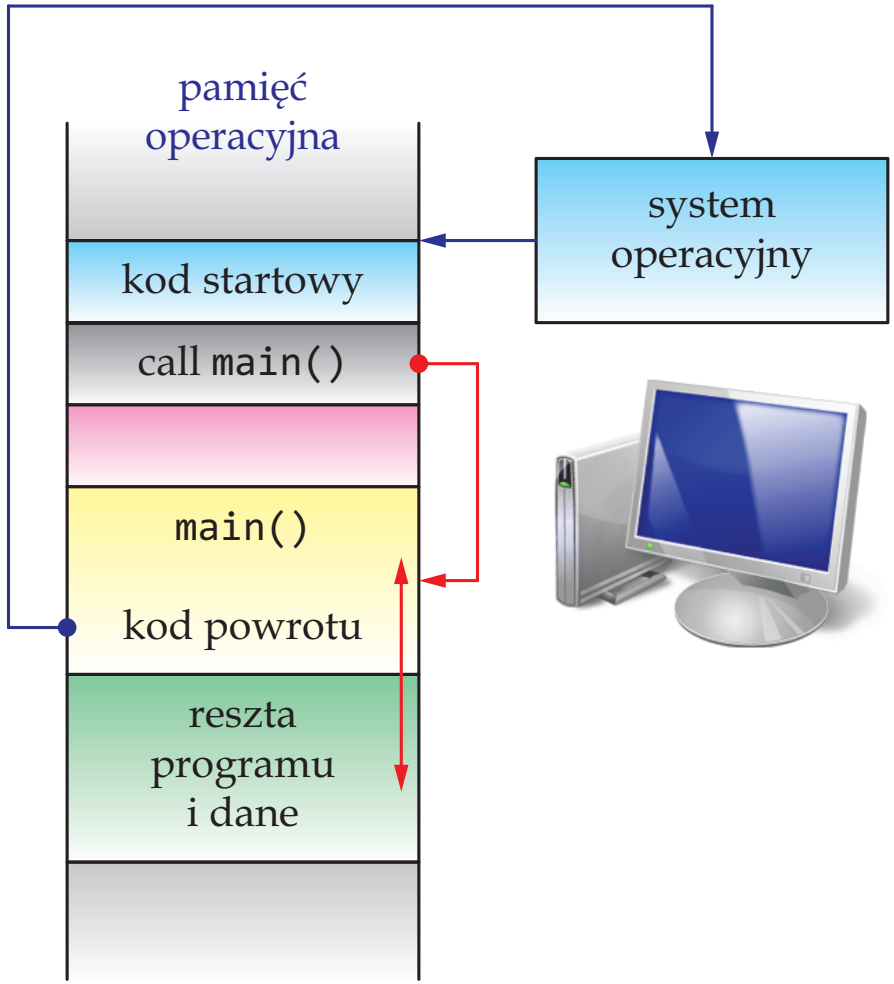
Funkcja `main()` stanowi punkt programu, od którego zaczyna się jego wykonywanie. Upraszczając, funkcja `main()` jest wywoływana przez **system operacyjny**.

Wartość, będąca rezultatem funkcji `main()` jest przekazywana systemowi operacyjnemu (**kod wyjścia programu**).

Kod wyjścia programu może być wykorzystywany w **skryptach systemu operacyjnego**.

Scenariusz uruchomienia kodu wykonywalnego programu:

- użytkownik przekazuje systemowi operacyjnemu zlecenie uruchomienia programu,
- system operacyjny ładuje kod programu do pamięci, przygotowuje jego środowisko i przekazuje sterowanie do bloku **kodu startowego** programu (*startup code*),
- kod startowy wykonuje czynności inicjalizujące, na zakończenie wywołuje funkcję `main()`,
- po zakończeniu działania funkcji `main()` bieżący proces jest kończony i do systemu przekazywany jest kod wyjścia.



Przykładowy skrypt będący tzw. **plikiem wsadowym** (*batch file*) DOS/Windows (`*.bat`) testujący kod wyjścia programu `foo.exe`.

```
@echo off

foo.exe

if errorlevel 1 goto one
if errorlevel 0 goto zero

:zero
  echo Exit Code 0
  goto end

:one
  echo Exit Code 1
  goto end

:end
  pause
```

Zamiast rezultatu w postaci wartości numerycznych można wykorzystać symbole:

- EXIT_SUCCESS – oznacza zakończenie programu bez błędów,
- EXIT_FAILURE – oznacza zakończenie programu z informacją o błędzie.

```
#include <stdlib.h>

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

Stosowanie tych symboli jest zalecane przez standard POSIX.

Biblioteki i dyrektywa #include

Symbole `EXIT_SUCCESS` i `EXIT_FAILURE` nie są częścią języków C/C++, są zdefiniowane w bibliotece języka C `stdlib.h`.

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

Pliki z rozszerzeniem `*.h` to tzw. **pliki nagłówkowe**.

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

Pliki nagłówkowe w języku C mają rozszerzenie *.h. 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 odpowiedni plik nagłówkowy.

```
#include <cstdlib>
#include <iostream>

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

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

Poważny program w języku C

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
  float cm, inch;

  printf_s("Przeliczanie centymetrow na cale\n");in
  printf_s("Podaj wymiar w cm: ");

  scanf_s("%f", &cm);

  inch = 0.3937 * cm;

  printf_s("%gcm to %f\"\n", cm, inch);

  return EXIT_SUCCESS;
}
```

Poważny program w języku C++

```
#include <cstdlib>
#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 EXIT_SUCCESS;
}
```

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

Podstawy obsługi wejścia/wyjścia

```
int puts(const char *str);
```

Funkcja wysyła na standardowe wyjście napis `*str`, a następnie znak nowej linii.

```
char *gets(char *str);
```

Funkcja czyta linię ze standardowego wejścia (usuwa ją stamtąd) i umieszcza w tablicy znakowej wskazywanej przez `*str`. Ostatni znak linii (znak nowego wiersza `\n`) zastępuje zerem (znakiem `\0`).

Funkcja nie sprawdza, czy jest miejsce do zapisu w tablicy `*str`, funkcja ta może być niebezpieczna dla programu. Z tego powodu powodu w niektórych kompilatorach została usuniętą lub zastąpiona inną funkcją, np.: `gets_s()`.

```
int printf
(
  const char *format [,
  argument] ...
);
```

```
int printf_s
(
  const char *format [,
  argument] ...
);
```

```
int _printf_s_l
(
  const char *format,
  locale_t locale [,
  argument] ...
);
```

Funkcje formatują tekst zgodnie z formatem wypisują tekst na standardowe wyjście (stdout).

```
int wprintf_s
(
  const wchar_t *format [,
  argument] ...
);
```

```
int _wprintf_s_l
(
  const wchar_t *format,
  locale_t locale [,
  argument] ...
);
```

Funkcje formatują tekst zgodnie z formatem i wypisują tekst na standardowe wyjście (stdout).

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
  int i = 4;
  float f = 3.1415;
  const char *s = "Monty Python";

  printf_s("i = %d\nf = %.1f\n", i, f);
  printf_s("Wskaznik *s wskazuje na napis: %s\n", s);

  return EXIT_SUCCESS;
}
```

Format składa się ze znaków innych niż znak %, które są kopiowane bez zmian na wyjście oraz sekwencji sterujących, zaczynających się od symbolu procenta, w postaci:

```
%[flagi][szerokość][.precyzja][rozmiar]typ
```

Jeżeli po znaku procenta występuje od razu drugi procent (%%) to cała sekwencja traktowana jest jak zwykły znak procenta (tzn. jest on wypisywany na wyjście).

W sekwencji format możliwe są następujące **flagi**:

- `-` – oznacza, że pole ma być wyrównane do lewej, a nie do prawej,
- `+` – oznacza, że dane liczbowe zawsze poprzedzone są znakiem,
- spacja – oznacza, że liczby nieujemne poprzedzone są dodatkową spacją,
- `#` – powoduje, że wynik jest przedstawiony w alternatywnej postaci:
  - dla formatu `o` powoduje zmianę precyzji, jeżeli jest to konieczne, aby na początku wyniku było zero,
  - dla formatów `x` i `X` niezerowa liczba poprzedzona jest ciągiem `0x` lub `0X`,
  - dla formatów `a`, `A`, `e`, `E`, `f`, `F`, `g` i `G` wynik zawsze zawiera kropkę nawet jeżeli nie ma za nią żadnych cyfr,
  - dla formatów `g` i `G` końcowe zera nie są usuwane,
- `0` – dla formatów `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g` i `G` do wyrównania pola wykorzystywane są zera zamiast spacji za wyjątkiem wypisywania wartości nieskończoność i `NaN`.

Minimalna **szerokość** pola oznacza ile najmniej znaków ma zająć dane pole. Jeżeli wartość po formatowaniu zajmuje mniej miejsca jest ona wyrównywana spacjami z lewej strony (chyba, że podano flagi, które modyfikują to zachowanie). Domyślna wartość tego pola to 0.

Precyzja dla formatów:

- `d`, `i`, `o`, `u`, `x` i `X` – określa minimalną liczbę cyfr, które mają być wyświetlone i ma domyślną wartość 1,
- `a`, `A`, `e`, `E`, `f` i `F` – określa liczbę cyfr, które mają być wyświetlone po kropce i ma domyślną wartość 6,
- `g` i `G` – określa liczbę cyfr znaczących i ma domyślną wartość 1,
- `s` – określa maksymalną liczbę znaków, które mają być wypisane.

Szerokość pola może być albo dodatnią liczbą zaczynającą się od cyfry różnej od zera albo gwiazdką. Podobnie jest z precyzja z tą różnicą, że jest jeszcze poprzedzona kropką.

Rozmiar argumentu:

- dla formatów `d` i `i` można użyć jednego ze modyfikator rozmiaru:
  - `hh` – oznacza, że format odnosi się do argumentu typu `signed char`,
  - `h` – oznacza, że format odnosi się do argumentu typu `short`,
  - `l` – oznacza, że format odnosi się do argumentu typu `long`,
  - `ll` – oznacza, że format odnosi się do argumentu typu `long long`,
  - `j` – oznacza, że format odnosi się do argumentu typu `intmax_t`,
  - `z` – oznacza, że format odnosi się do argumentu typu będącego odpowiednikiem typu `size_t` ze znakiem,
  - `t` – oznacza, że format odnosi się do argumentu typu `ptrdiff_t`,
- dla formatów `o`, u, `x` i `X` można użyć takich samych modyfikatorów rozmiaru jak dla formatu `d` i oznaczają one, że format odnosi się do argumentu odpowiedniego typu bez znaku,
- dla formatu `n` można użyć takich samych modyfikatorów rozmiaru jak dla formatu `d` i oznaczają one, że format odnosi się do argumentu będącego wskaźnikiem na dany typ,
- dla formatów `a`, `A`, `e`, `E`, `f`, `F`, `g` i `G` można użyć modyfikatorów rozmiaru `L`, który oznacza, że format odnosi się do argumentu typu `long double`,
- dodatkowo, modyfikator `l` dla formatu `c` oznacza, że odnosi się on do argumentu typu `wint_t`, a dla formatu `s`, że odnosi się on do argumenty typu wskaźnik na `wchar_t`.

Funkcje z rodziny `printf()` obsługują następujące **typy**:

- `d`, `i` – argument typu `int` jest przedstawiany jako liczba całkowita ze znakiem w postaci `[-]ddd`,
- `o`, `u`, `x`, `X` – argument typu `unsigned int` jest przedstawiany jako nieujemna liczba całkowita zapisana w systemie oktalnym (`o`), dziesiętnym (`u`) lub heksadecymalnym (`x` i `X`),
- `f`, `F` – argument typu `double` jest przedstawiany w postaci `[-]ddd.ddd`,
- `e`, `E` – argument typu `double` jest reprezentowany w postaci `[i]d.ddde+dd`, gdzie liczba przed kropką dziesiętną jest różna od zera, jeżeli liczba jest różna od zera, a `+` oznacza znak wykładnika,
- `g`, `G` – argument typu `double` jest reprezentowany w formacie takim jak `f` lub `e` zależnie od liczby znaczących cyfr w liczbie oraz określonej precyzji,
- `a`, `A` – argument typu `double` przedstawiany jest w formacie `[-]0xh.hhhp+d` czyli analogicznie jak dla e i E, tyle że liczba zapisana jest w systemie heksadecymalnym,
- `c` – argument typu `int` jest konwertowany do `unsigned char` i wynikowy znak jest wypisywany,

- `s` – argument powinien być typu wskaźnik na `char` (lub `wchar_t`),
- `p` – argument powinien być typu wskaźnik na `void`, jest on konwertowany na serię drukowalnych znaków w sposób zależny od implementacji,
- `n` - argument powinien być wskaźnikiem na liczbę całkowitą ze znakiem, do którego zwracana jest liczba zapisanych znaków.

W przypadku formatów `f`, `F`, `e`, `E`, `g`, `G`, `a` i `A` wartość nieskończoność jest przedstawiana w formacie `[-]inf` lub `[-]infinity` zależnie od implementacji.

Wartość `NaN` jest przedstawiana w postaci `[-]nan` lub `[i]nan(sekwencja)`, gdzie sekwencja jest zależna od implementacji. W przypadku formatów określonych wielką literą również wynikowy ciąg znaków jest wypisywany wielką literą.

Jeżeli funkcje zakończą się sukcesem zwracają liczbę znaków w tekście (wypisanym na standardowe wyjście, do podanego strumienia lub tablicy znaków) nie wliczając kończącego `\0`. W przeciwnym wypadku zwracana jest liczba ujemna.

```
int scanf(
    const char *format [,
    argument] ...
);
```

```
int scanf_s(
    const char *format [,
    argument] ...
);
```

```
int _scanf_s_l(
    const char *format,
    locale_t locale [,
    argument] ...
);
```

Funkcje odczytują dane zgodnie z formatem dane ze standardowego wejścia (tj. `stdin`).

```
int wscanf_s(
   const wchar_t *format [,
   argument] ...
);
```

```
int _wscanf_s_l(
   const wchar_t *format,
   locale_t locale [,
   argument] ...
);
```

Funkcje odczytują dane zgodnie z formatem dane ze standardowego wejścia (tj. stdin

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
  int i;
  float f;
  char s[80];

  scanf_s("%d %f", &i, &f);
  scanf_s("%s", s, 80);

  return EXIT_SUCCESS;
}
```

Format składa się ze zwykłych znaków (innych niż znak %) oraz sekwencji sterujących, zaczynających się od symbolu procenta, w postaci:

```
%[*][szerokość][rozmiar]typ
```

Wystąpienie w formacie białego znaku powoduje, że funkcje z rodziny `scanf()` będą odczytywać i odrzucać znaki, aż do napotkania pierwszego znaku nie będącego białym znakiem. Wszystkie inne znaki muszą dokładnie pasować do danych wejściowych.

Wszystkie białe znaki z wejścia są ignorowane, chyba że sekwencja sterująca określa typ `[`, `c` lub `n`.

Jeżeli w sekwencji sterującej występuje gwiazdka to dane z wejścia zostaną pobrane zgodnie z formatem, ale wynik konwersji nie zostanie nigdzie zapisany. W ten sposób można pomijać część danych.

Rozmiar argumentu:

- dla formatów `d`, `i`, `o`, `u`, `x` i `n` można użyć jednego z modyfikatorów rozmiaru:
  - `hh` – oznacza, że format odnosi się do argumentu typu wskaźnik na `signed char` lub `unsigned char`,
  - `h` – oznacza, że format odnosi się do argumentu typu wskaźnik na `short` lub wskaźnik na `unsigned short`,
  - `l` – oznacza, że format odnosi się do argumentu typu wskaźnik na `long` lub wskaźnik na `unsigned long`,
  - `ll` – oznacza, że format odnosi się do argumentu typu wskaźnik na `long long` lub wskaźnik na `unsigned long long`,
  - `j` – oznacza, że format odnosi się do argumentu typu wskaźnik na `intmax_t` lub wskaźnik na `uintmax_t`,
  - `z` – oznacza, że format odnosi się do argumentu typu wskaźnik na `size_t` lub odpowiedni typ ze znakiem,
  - `t` – oznacza, że format odnosi się do argumentu typu wskaźnik na `ptrdiff_t` lub odpowiedni typ bez znaku,
- dla formatów `a`, `e`, `f` i `g` można użyć modyfikatorów rozmiaru:
  - `l` – który oznacza, że format odnosi się do argumentu typu wskaźnik na `double`,
  - `L` – który oznacza, że format odnosi się do argumentu typu wskaźnik na `long double`,
- dla formatów `c`, `s` i `[` modyfikator `l` oznacza, że format odnosi się do argumentu typu wskaźnik na `wchar_t`.

Funkcje z rodziny `scanf()` obsługują następujące typy:

- `d`, `i` – odczytuje liczbę całkowitą, której format jest taki sam jak oczekiwany format przy wywołaniu funkcji `strtol()` z argumentem `base` równym odpowiednio 10 dla `d` lub 0 dla `i`, argument powinien być wskaźnikiem na `int`,
- `o`, `u`, `x` – odczytuje liczbę całkowitą, której format jest taki sam jak oczekiwany format przy wywołaniu funkcji `strtoul()` z argumentem `base` równym odpowiednio 8 dla `o`, 10 dla `u` lub 16 dla `x`, argument powinien być wskaźnikiem na `unsigned int`,
- `a`, `e`, `f`, `g` – odczytuje liczbę rzeczywistą, nieskończoność lub `NaN`, których format jest taki sam jak oczekiwany przy wywołaniu funkcji `strtod()`, argument powinien być wskaźnikiem na `float`,
- `c` – odczytuje dokładnie tyle znaków ile określono w maksymalnym rozmiarze pola (domyślnie 1), argument powinien być wskaźnikiem na `char`,
- `s` – odczytuje sekwencje znaków nie będących białymi znakami, argument powinien być wskaźnikiem na `char`,

- `[` – odczytuje niepusty ciąg znaków, z których każdy musi należeć do określonego zbioru, argument powinien być wskaźnikiem na `char`,
- `p` – odczytuje sekwencje znaków zależną od implementacji odpowiadającą ciągowi wypisywanemu przez funkcję `printf()`, gdy podano sekwencję `p`, argument powinien być typu wskaźnik na wskaźnik na `void`,
- `n` – nie odczytuje żadnych znaków, ale zamiast tego zapisuje do podanej zmiennej liczbę odczytanych do tej pory znaków, argument powinien być typu wskaźnik na `int`,
- `A`, `E`, `F`, `G` i `X` są również dopuszczalne i mają takie same działanie jak `a`, `e`, `f`, `g` i `x`.

Funkcja zwraca `EOF` jeżeli nastąpi koniec danych lub błąd odczytu zanim jakiekolwiek konwersje zostaną dokonane lub liczbę poprawnie wczytanych pól (która może być równa zero).

sekwencja specjalna, wartość, znak, znaczenie			
\a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline
\r	0x0D	CR	Carriage return
\t	0x09	HT	Tab
\v	0x0B	VT	Vertical tab
\\	0x5c	\	Backslash
\'	0x27	'	Quote
\"	0x22	"	Quotation marks
\?	0x3F	?	Question
\O			O = łańcuch cyfr ósemkowych
\xH			H = łańcuch cyfr szesnastkowych

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**.

```
JW23, JW_23
wartmaksymalna, wart_maksymalna, WartMaksymalna, wartMaksymalna
maksdl, maks_dl, MaksDl, maksDl
_5Pi, _5_Pi
```

Słowa kluczowe

Słowa kluczowe to identyfikatory zastrzeżone i nie mogą być inaczej stosowane niż określa to standard języka.

Słowa kluczowe powinny być pisane tak jak je podano, a więc wyłącznie z wykorzystaniem małych liter. Słowa kluczowe wg. normy ANSI C89:

```
auto     break    case     char     const    continue  default  do
double   else     enum     extern   float    for       goto     if
int      long     register return   short    signed    sizeof   static
struct   switch   typedef  while    union    unsigned  void     volatile
```

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

```
asm           dynamic_cast   namespace   reinterpret_cast   try
bool          explicit       new         static_cast        typeid
catch         false          operator    template           typename
class         friend         private     this               using
const_cast    inline         public      throw              virtual
delete        mutable        protected   true               wchar_t
```

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

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
*/
```

Standard ANSI C nie dopuszcza stosowania komentarzy zagnieżdżonych, choć niektóre kompilatory na to zezwalają.

Zmienne

Zmienna to konstrukcja programistyczna posiadająca trzy podstawowe atrybuty:

- symboliczną nazwę,
- miejsce przechowywania,
- wartość.

Zmienna pozwala w kodzie źródłowym na odwoływanie się przy pomocy nazwy do wartości lub miejsca przechowywania. Nazwa służy do identyfikowania zmiennej w związku z tym często nazywana jest **identyfikatorem**.

Miejsce przechowywania przeważnie znajduje się w pamięci komputera i określane jest przez adres i długość danych. Wartość to zawartość miejsca przechowywania.

Zmienna zazwyczaj posiada również czwarty atrybut: **typ**, określający rodzaj danych przechowywanych w zmiennej i co za tym idzie sposób reprezentacji wartości w miejscu przechowywania.

W programie wartość zmiennej może być odczytywana lub zastępowana nową wartością, tak więc wartość zmiennej może zmieniać się w trakcie wykonywania programu, natomiast dwa pierwsze atrybuty (nazwa i miejsce przechowywania) nie zmieniają się w trakcie istnienia zmiennej.

Konstrukcją podobną lecz nie pozwalającą na modyfikowanie wartości jest **stała**.

W językach ze **statycznym typowaniem** zmienna ma określony typ danych jakie może przechowywać. Jest on wykorzystywany do określenia reprezentacji wartości w pamięci, kontrolowania poprawności operacji wykonywanych na zmiennej (kontrola typów) oraz konwersji danych jednego typu na inny.

W językach z **typowaniem dynamicznym** typ nie jest atrybutem zmiennej lecz wartości w niej przechowywanej. Zmienna może wtedy w różnych momentach pracy programu przechowywać dane innego typu.

Deklaracja zmiennej to stwierdzenie, że dany identyfikator jest zmienną, przeważnie też określa typ zmiennej. W zależności od języka programowania deklaracja może być obligatoryjna, opcjonalna lub nie występować wcale.

Definicja oprócz tego, że deklaruje zmienną to przydziela jej pamięć. Podczas definiowania lub deklarowania zmiennej można określić jej dodatkowe atrybuty wpływające na sposób i miejsce alokacji, czas życia, zasięg i inne.

Zasięg zmiennej określa gdzie w treści programu zmienna może być wykorzystana, natomiast **czas życia** zmiennej to okresy w trakcie wykonywania programu gdy zmienna ma przydzieloną pamięć i posiada (niekoniecznie określoną) wartość.

Ze względu na zasięg można wyróżnić podstawowe typy zmiennych:

- **globalne** – obejmujące zasięgiem cały program,
- **lokalne** – o zasięgu obejmującym pewien blok, podprogram.

Podobnie ze zmiennymi w klasie mogą być dostępne:

- tylko dla danej klasy (zmienna **prywatna**),
- dla danej klasy i jej potomków (**zmienna chroniona**),
- w całym programie (**zmienna publiczna**).

Zmienne mogą zmieniać swój pierwotny zasięg na przykład poprzez importowanie/włącznie do zasięgu globalnego modułów, pakietów czy przestrzeni nazw.

Ze względu na czas życia i sposób alokacji zmienna może być:

- statyczna,
- automatyczna,
- dynamiczna.

Dla zmiennej **statycznej** pamięć jest rezerwowana w momencie kompilacji lub ładowania programu. Takimi zmiennymi są zmienne globalne, zmienne klasy (współdzielone przez wszystkie obiekty klasy, a nawet dostępne spoza klasy), statyczne zmienne lokalne funkcji (współdzielone pomiędzy poszczególnymi wywołaniami funkcji i zachowujące wartość po zakończeniu).

Zmiennej **automatycznej** pamięć jest automatycznie przydzielana w trakcie działania programu. Są to przeważnie zmienne lokalne podprogramów i ich parametry formalne, znikają po zakończeniu podprogramu.

Pamięć dla zmiennej **dynamicznej** alokowana jest ręcznie w trakcie wykonywania programu przy pomocy specjalnych konstrukcji lub funkcji. W zależności od języka zwalnianie pamięci może być ręczne lub automatyczne, Zazwyczaj nie posiada własnej nazwy, lecz odwoływać się do niej trzeba przy pomocy wskaźnika, referencji lub zmiennej o semantyce referencyjnej.

Typy zmiennych w języku C

typ, wielkość, charakterystyka		
`char`	1	liczba całkowita od `0` do `255`
`int`	4	liczba całkowita od `-2147483648` do `2147483647`
`float`	4	liczba rzeczywista
		od `-3.4028235e+38` do `-1.401298e-45`
		od `1.401298e-45` do `3.4028235e+38`
`double`	8	liczba rzeczywista długa
		od `-1.79769313486231570e+308` do `-4.94065645841246544e-324`
		od `4.94065645841246544e-324` do `1.79769313486231570e+308`

Typ znakowy char

Zmienne zadeklarowane jako znakowe char są dostatecznie duże aby pomieścić dowolny element zbioru znaków dla danej maszyny bądź systemu operacyjnego.

Wartość zmiennej znakowej to liczba całkowita równa kodowi danego znaku. Zmienna typu char jest zatem krótką liczbą całkowitą i tak może być traktowana, można zmiennych tego typu używać w wyrażeniach.

```
char c, d;

c = 'A';
d = c + 1;

printf_s("%c\n", c); // A
printf_s("%c\n", d); // B

printf_s("%d\n", c); // 65
printf_s("%d\n", d); // 66
```

Literał znakowy jest ciągiem złożonym z jednego lub więcej znaków, zawartych w **apostrofach**.

Wartością literału znakowego, zawierającego tylko jeden znak jest numeryczna wartość tego znaku, w zbiorze znaków maszyny wykonującej program.

Wartość literału wieloznakowego jest zależna od implementacji. W języku C literał znakowy reprezentowany jest jako wartość typu całkowitoliczbowego `int`.

W języku C++ literał znakowy reprezentowany jest przez wartość typu `char`, literał wieloznakowy natomiast przez wartość typu `int`.

Przykładowo: `'A'` – dla maszyn wykorzystujących kod ASCII literał ten reprezentuje wartość całkowitą odpowiadającą kodowi znaku, jest to wartość dziesiętna 65.

Zwyczajowo dane typu `char` reprezentowane są na jednym bajcie i służą do reprezentowania znaków kodowanych wg. ASCII. Do przechowywania kodów znaków wg. kodowania międzynarodowego wykorzystuje się typ `wchar_t`.

Standard ANSI wprowadza typ całkowity `wchar_t`, jest to typ całkowity zdefiniowany w pliku nagłówkowym `stddef.h`.

Stałe rozszerzonego zbioru znaków zapisuje się z prefixem `L`, np.:
`wchar_t x = L'A';`

W języku C++ wprowadzono uniwersalne nazwy znaków, taka nazwa zaczyna się od `\u` lub `\U` i zawiera cyfry szesnastkowe określające kod znaku wg ISO 10646.

To czy właściwy znak się pojawi, zależy nie tylko od języka, ale od jego bibliotek i tego, czy środowisko systemowe obsługuje dany zestaw kodowania.

Typ całkowitoliczbowy int

Zmienne typu całkowitego `int` mają zwykle naturalny rozmiar wynikający z modelu programistycznego architektury maszyny lub środowiska systemowego.

Zwykle w środowiskach 16-bitowych rozmiar danej typu `int` to dwa bajty, w środowiskach 32-bitowych to 4 bajty.

Domyślnie typ `int` reprezentuje liczbę ze znakiem (wartości dodatnie i ujemne).

Rozmiar i zakres typu zmienia się, wraz ze zmianą architektury sprzętowej, oprogramowania systemowego i kompilatorów.

Standardy zakładają, że typ `int` będzie reprezentowany minimalnie na 16-tu bitach (z uwzględnieniem bitu znaku), co odpowiada zakresowi -32768 do 32767.

Typy pochodne typów całkowitych

Modyfikatory `signed` i `unsigned` mogą być stosowane do typów `char` i `int`. Zmieniają one sposób traktowania najstarszego bitu liczby.

Modyfikatory pozwalają na tworzenie specyfikacji typów pochodnych:

- `unsigned int` – typ całkowity służący do reprezentacji liczb całkowitych bez znaku; najstarszy bit liczby jest uznawany za jeden z bitów wartości,
- `signed int` – typ całkowity służący do reprezentacji liczb całkowitych ze znakiem; najstarszy bit liczby jest bitem przechowującym informację o znaku liczby, nie wchodzi do bitów wartości,
- `unsigned char` i `signed char` analogicznie jak dla typu `int`.

Domyślnie typ `int` jest całkowity ze znakiem, natomiast typ `char` często zależy od implementacji.

Modyfikator `short` sygnalizuje chęć skrócenia danej w stosunku do rozmiaru.

Modyfikator `long` sygnalizuje chęć posłużenia się daną dłuższą w stosunku do rozmiaru typu `int`.

Modyfikatory `short` i `long` mogą być stosowane do typu `int`:

- `short int` – typ całkowity służący do reprezentowania liczb o potencjalnie *krótszej* reprezentacji wewnętrznej niż typ `int`, zatem potencjalnie o mniejszym zakresie wartości,
- `long int` – to typ całkowity służący do reprezentowania liczb o potencjalnie *dłuższej* reprezentacji wewnętrznej niż typ `int`, zatem potencjalnie o większym zakresie wartości,
- `long long int` – to typ wprowadzony w C99, służy do reprezentowania bardzo dużych liczb całkowitych (powinien być implementowany na min. 64 bitach).

Standard ANSI zakłada, że `int` oraz `short int` są co najmniej 16-bitowe, `long int` jest co najmniej 32-bitowy.

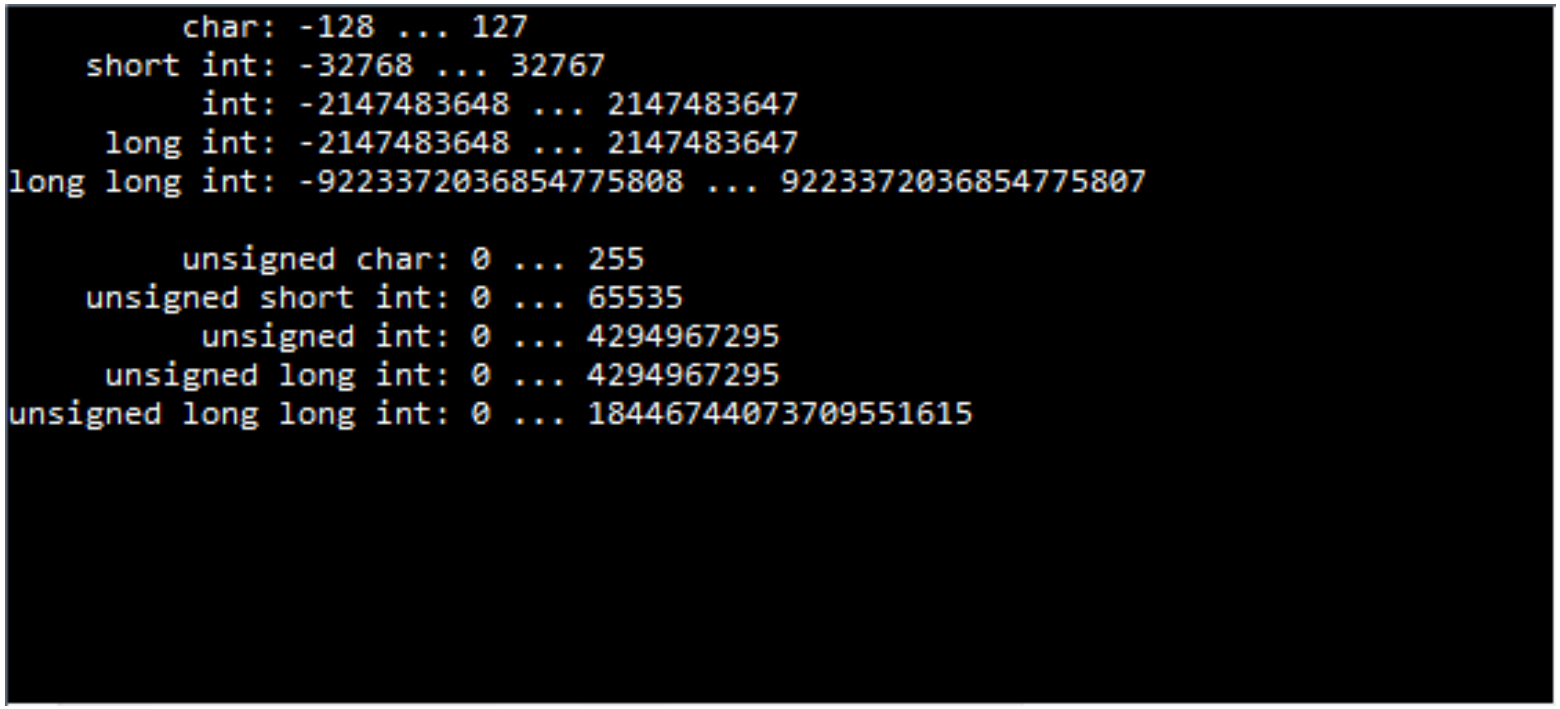
Modyfikatory `short` i `long` wprowadzono po to, by umożliwić posługiwanie się różnymi zakresami liczb całkowitych tam, gdzie programiście może się to przydać.

Dodatkowo można powiedzieć:

$$\text{sizeof(char)} <= \text{sizeof(short int)} <= \text{sizeof(int)} <= \text{sizeof(long int)}$$

Każdy kompilator powinien posiadać dokumentację określającą szczegółowy zakres poszczególnych typów. Czasem warto skompilować i uruchomić program, wykorzystujący stałe zdefiniowane w plikach nagłówkowych `limits.h` i `float.h` określające zakresy liczbowe typów.

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

int main()
{
  printf_s("          char: %d ... %d\n", CHAR_MIN, CHAR_MAX);
  printf_s("    short int: %hd ... %hd\n", SHRT_MIN, SHRT_MAX);
  printf_s("           int: %d ... %d\n", INT_MIN, INT_MAX);
  printf_s("     long int: %ld ... %ld\n", LONG_MIN, LONG_MAX);
  printf_s("long long int: %lld ... %lld\n\n", LLONG_MIN, LLONG_MAX);
  // LONG_LONG_MIN, LONG_LONG_MAX
  printf_s("          unsigned char: 0 ... %u\n", UCHAR_MAX);
  printf_s("    unsigned short int: 0 ... %hu\n", USHRT_MAX);
  printf_s("           unsigned int: 0 ... %u\n", UINT_MAX);
  printf_s("     unsigned long int: 0 ... %lu\n", ULONG_MAX);
  printf_s("unsigned long long int: 0 ... %llu\n", ULLONG_MAX);
  // ULONG_LONG_MAX

  return EXIT_SUCCESS;
}
```


char: -128 ... 127
short int: -32768 ... 32767
int: -2147483648 ... 2147483647
long int: -2147483648 ... 2147483647
long long int: -9223372036854775808 ... 9223372036854775807
unsigned char: 0 ... 255
unsigned short int: 0 ... 65535
unsigned int: 0 ... 4294967295
unsigned long int: 0 ... 4294967295
unsigned long long int: 0 ... 18446744073709551615

Przybliżone zakresy zmiennych

Wybierając typ dla zmiennej trzeba oszacować jej, zakres wartości. Źle dobrane zakresy grożą postaniem przepełnienia zmiennych całkowitoliczbowych.

- typ `char` to ok. 128 na plus i minus, `unsigned char` to 255 na plus,
- typ `short int` to ok. 32 tyś. na plus i minus, `unsigned short int` to ok. 65 tyś. na plus,
- typ `int` (16 bitów) jak `short int`,
- typ `int` (32 bity) to ok. 2 miliardy na plus i minus, `unsigned int` to ok. 4 miliardy na plus (miliard to rząd wielkości odpowiadający komputerowemu gigabajtowi, GB),
- typ `long` jak `int` (32 bity),
- typ `long long int` (64 bity) to ok. 9 trylionów na plus i minus, `unsigned long long` to ok. 18 trylionów na plus (trylion to rząd wielkości odpowiadający komputerowemu eksabajtowi, EB).

Przepełnienie czyli przekroczenie zakresu zmiennej

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

int main()
{
  unsigned short int ui = USHRT_MAX;
  printf_s("%d\n", ui);
  ui++;
  printf_s("%d\n", ui);
  ui++;
  printf_s("%d\n\n", ui);

  signed short int si = SHRT_MAX;
  printf_s("%d\n", si);
  si++;
  printf_s("%d\n", si);
  si++;
  printf_s("%d\n", si);

  return EXIT_SUCCESS;
}
```

```
65535
0
1

32767
-32768
-32767
```

Typy pochodne typów zmiennopozycyjne

Modyfikatory `short` i `long` a typy zmiennopozycyjne:

- można stosować modyfikatory `short` i `long` z typami `float` i `double`, jednak tylko kombinacja `long double` ma sens,
- typ `double` naturalnie rozszerza typ `float` zatem zapis `long float` to po prostu przestarzały synonim typu `double`,
- z kolei typu `double` nie można skrócić, zatem specyfikacja `short double` nie ma sensu,
- nie można również skrócić typu `float`, zatem specyfikacja `short float` nie ma sensu.

Definiowanie synonimów typów

Specyfikacja `typedef` pozwala na przypisanie **identyfikatora** do istniejącej wcześniej **definicji typu**.

```
typedef unsigned char      byte;
typedef unsigned short int word;
typedef unsigned long int  counter;
```

Literały całkowitoliczbowe

Literał całkowity może być zapisywana dziesiętnie, ósemkowo, szesnastkowo.

Wszystkie literały rozpoczynające się od zera traktowane są jako ósemkowe.

Wszystkie literały rozpoczynające się od przedrostka `0x` lub `0X` są traktowane jako szesnastkowe.

```
int i = 10;     // stala dziesietna
int o = 077;    // stala osemkowa
int h = 0xff;   // stala szesnastkowa
```

Literał całkowitoliczbowy może być zakończona przyrostkiem `u` lub `U` co oznacza, że liczba jest **bez znaku**.

Literał całkowitoliczbowy może być zakończona przyrostkiem `l` lub `L` co oznacza, że liczba **jest długa**.

Wartość literału całkowitoliczbowego nie może przekraczać zakresu typu liczby całkowitej długiej bez znaku (`unsigned long int`). Wartości większe będą obcinane.

Dla implementacji zakładającej 32-bitową długość liczby długiej bez znaku, wartość maksymalna wynosi odpowiednio:

```
DEC: 4 294 967 295, OCT: 037777777777, HEX: 0xFFFFFFFF
```

Typ wyliczeniowy

Typ wyliczeniowy pozwala na uporządkowanie listy elementów, które można przedstawić jedynie nazwami. Przykładem mogą być tygodnia, miesiące, kolory.

Typ wyliczeniowy to tak na prawdę, lista nazwanych stałych całkowitych.

```
enum rgbColors
{
  RED,
  GREEN,
  BLUE
};
```

Stałe wyliczeniowe, są typu `int`, mogą wystąpić w każdym miejscu dozwolonym dla danej całkowitej.

Identyfikatory stałych wyliczeniowych powinny być unikatowe w ramach danego wyliczenia.

Każda stała wyliczeniowa ma swoją wartość całkowitą. Pierwsza stała na liście otrzymuje wartość 0, następna 1, itd.

Każda stała występująca w wyliczeniu może posiadać swój **inicjalizator**, przypisujący mu wartość (również ujemną) wyznaczoną przez programistę.

Każdy element wyliczenia nie posiadający inicjalizatora otrzymuje **wartość o jeden większą** od swojego poprzednika na liście.

```
enum bodyType
{
  HATCHBACK = 1,
  ESTATE,
  SEDAN,
  COUPE,
  SUV
};
```

```
int body;

switch(body)
{
  case SEDAN: ...
  case SUV:   ...
}
```

Zmienne wyliczeniowe

Deklarowanie zmiennych wyliczeniowych spotyka się sporadycznie:

```
enum months
{
  JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};

enum months m = MAY;
```

Zwyczajowo, można tak:

```
int m = MAY;
```

Przykład iteracji po kolejnych miesiącach:

```
int m;

for(m = JAN; m <= DEC; m++)
```

C++ nie wymaga słowa enum przy deklaracji zmiennych wyliczeniowych.

Typ logiczny

W języku C++ wprowadzono **typ logiczny** `bool` o predefiniowanych wartościach `true` i `false`.

W standardzie C99 wprowadzono typ logiczny `_Bool` oraz wartości `true` i `false` (plik nagłówkowy `stdbool.h`).

W standardzie C89 wykorzystuje się standardowy typ całkowitoliczbowy oraz wartości całkowite 0 i 1 lub własne definicje:

```
enum boolean
{
  FALSE,
  TRUE
};
```

```
#define TRUE 1
#define FALSE 0
```

```
#define TRUE (0 == 0)
#define FALSE (!TRUE)
```

Typ zmiennopozycyjny

Standard nie określa wewnętrznej reprezentacji danych zmiennopozycyjnych, zwykle implementacje są zgodne formatem IEEE dotyczącym takich liczb.

`float` to typ przeznaczony do reprezentowania liczb rzeczywistych pojedynczej precyzji.

`double` to typ przeznaczony do reprezentowania liczb rzeczywistych w podwójnej precyzji.

Literał zmiennopozycyjny składa się z:

- części całkowitej (ciąg cyfr),
- kropki dziesiętnej,
- części ułamkowej (ciąg cyfr),
- opcjonalnej litery `e` lub `E` oraz wykładnika potęgi ze znakiem,
- opcjonalnego przyrostka `f` lub `F` i `l` lub `L`.

Można pominąć część całkowitą lub część ułamkową (lecz nie obie jednocześnie).

W przypadku braku przyrostków stałe zmiennopozycyjne są typu `double`.

Dodając przyrostek `f` lub `F` można wymusić aby stała była typu `float`, podobnie dodając przyrostek `l` lub `L` wymusza aby stała była typu `long double`.

zapis, znaczenie stałych	
23.45e6	$23,45 \cdot 10^6$
.0	0
0.	0
1.	0
-1.23	$-1,23$
2e-5	$2 \cdot 10^{-5}$
3E+10	$3 \cdot 10^{10}$
.09E34	$0,09 \cdot 10^{34}$

Typ void

Wystąpienie typu `void` (pusty, próżny) w deklaracji oznacza **brak wartości**.

W zależności od kontekstu interpretacja zapisu `void` może się nieznacznie zmieniać, zawsze jednak jest to sygnał, że w danym miejscu nie przewiduje się wystąpienia żadnej konkretnej wartości lub konkretnego typu.

Funkcja bezparametrowa:

```
int funnp(void)
{
   ...
}
```

Funkcja nie udostępniająca rezultatu:

```
void fun(int i)
{
   ...
}
```

Bezparametrowa funkcja, nie udostępniająca rezultatu:

```
void fun(void)
{
   ...
}
```

Rzutowanie trezultatu funkcji na typ `void`:

```
(void)getchar();
```

Plik nagłówkowy `inttypes.h` definiuje *przenaszalne* typu całkowite o określonych właściwościach.

Typy o dokładnym rozmiarze (*exact width types*), np.: `int16_t`, `uint16_t`, `int_32_t`, `uint_32_t`, itp.

Najszybsze typy o minimalnym rozmiarze (*fastest minimum width types*), np.: `int_fast16_t`, `uint_fast16_t`, `int_fast32_t`, `uint_fast32_t`, itp.

Dla obsługi wartości takich typów zdefiniowano specjalne stałe formatujące, np.:`PRIu8`, `PRIu16`, `PRIu32`, `PRIu64`, `PRIuFAST8`, `PRIuFAST16`, `PRIuFAST32`, `PRIuFAST64`, itp.

Tablice

Tablica jest to kontener danych, w którym poszczególne komórki dostępne są z pomocą pewnych kluczy, które najczęściej przyjmują wartości numeryczne.

Tablica zwykle pozwala na przechowywanie wartości tego samego typu. Tablice mogą być **jednowymiarowe** lub **wielowymiarowe**.

Rozmiar tablicy jest albo ustalony z góry (tablice **statyczne**), albo może się zmieniać w trakcie wykonywania programu (tablice **dynamiczne**).

W matematyce odpowiednikiem tablicy jednowymiarowej jest ciąg, a tablicy dwuwymiarowej macierz.

Zgodnie ze standardem C89 i C++:

- tablica zawsze składa się z **ustalonej** i **znanej na etapie kompilacji** liczby elementów,
- liczba elementów tablicy **nie ulega zmianie** w trakcie działania programu – tablice są statyczne.

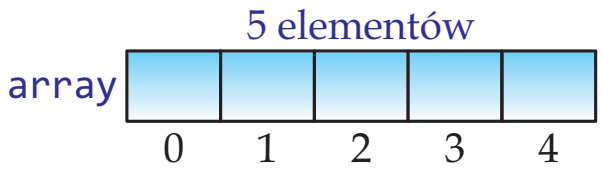
W standardzie C99 istnieją tablice VLA (*Variable Length Array*):

- liczba elementów tablicy może być **zdefiniowana w trakcie wykonania programu** – może być określona wartością zmiennej, wartość ta nie musi być znana na etapie kompilacji,
- liczba elementów tablicy **nie ulega zmianie** w trakcie działania programu – raz utworzona tablica zachowuje swój rozmiar.

Deklarowanie tablic jednowymiarowych

```
int array[5];
```

```
char buffer[80];
```



Parametryzacja liczby elementów tablicy pozwala na łatwiejszą modyfikację liczby przetwarzanych elementów.

```
#define MAXN 5
int array[MAXN];
```

```
#define MAXBUF 80
char buffer[MAXBUF];
```

Zmienna z kwalifikatorem const w języku C nie jest traktowana jako wartość stała i nie może być wykorzystywana do określania rozmiaru tablicy.

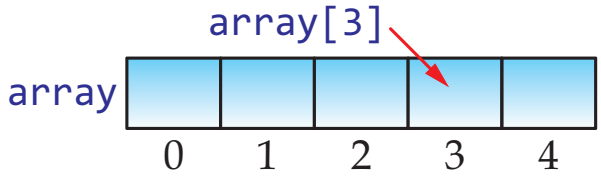
Zmienna z kwalifikatorem const w języku C++ może być wykorzystywana do określania rozmiaru tablicy.

```
const int MAXN = 5;
int array[MAXN];
```

```
const int MAXBUF = 80;
char buffer[MAXBUF];
```

Odwołania do elementów tablic

Elementy tablicy numerowane są zawsze od 0. Zatem jeżeli n oznacza liczbę elementów tablicy, to ostatni jej element ma numer n - 1.



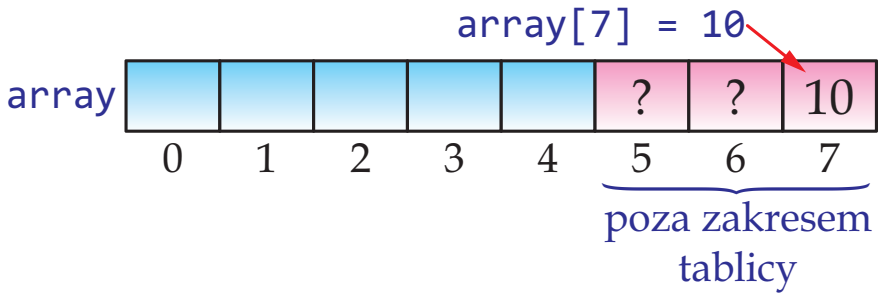

```
array[0] = 1;
```

```
array[n - 1] = 5;
```

```
a = 2 * array[3];
```

```
int i = 0;
int j = n - 1;
a = array[i] + array[j];
```

W języku C i C++ nie ma żadnych wbudowanych mechanizmów zabezpieczających przed odwoływaniem się do *elementów* leżących poza zakresem indeksowym tablic.



Badanie rozmiaru tablicy

```
int monthDays[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int monthDays = sizeof(monthDays) / sizeof(int);
```

lub

```
int monthDays[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int monthDays = sizeof(monthDays) / sizeof(monthDays[0]);
```

Typowe operacje na tablicach

Przetwarzanie tablic realizowane jest zwykle z wykorzystaniem instrukcji **iteracyjnych**. Do przetwarzania tablic najczęściej wykorzystuje się pętlę `for()`.

Ustawianie wartości wszystkich elementów tablicy (na przykład zerowanie):

```
for(i = 0; i < n; i++)
  array[i] = 0;
```

lub z wykorzystaniem operatora postinkrementacji:

```
for(i = 0; i < n; array[i++] = 0);
```

Wczytywanie danych ze strumienia wejściowego do tablicy:

```
for(i = 0; i < n; i++)
{
  printf_s("%d: ", i + 1);
  scanf_s("%d", &array[i]);
}
```

Wyprowadzanie danych z tablicy do strumienia wyjściowego:

```
for(i = 0; i < n; i++)
  printf_s("%d: %d\n", i + 1, array[i]);
```

wersja *uproszczona*:

```
for(i = 0; i < n; printf_s("%d: %d\n", i, array[i++]));
```

i wspak:

```
for(i = n; i > 0; printf_s("%d: %d\n", i + 1, array[--i]));
```

Zagadka:

```
int array[5] = {1, 2, 3, 4, 5};
int i, j;

i = 2;
array[++i] = 2 * ++i; // array = ?
```

Zagadka:

```
int array[5] = {1, 2, 3, 4, 5};
int i, j;

i = 2;
j = array[++i] + array[i]; // j = ?
```

Zagadka:

```
int array[5] = {1, 2, 3, 4, 5};
int i, j;

i = 2;
array[++i] = 2 * ++i; // array = {1, 2, 3, 4, 8}
```

Zagadka:

```
int array[5] = {1, 2, 3, 4, 5};
int i, j;

i = 2;
j = array[++i] + array[i]; // j = 8
```

Sumowanie wartości elementów tablicy:

```
int sum = 0;

for(i = 0; i < n; i++)
  sum += array[i]; // sum = sum + array[i];
```

wersja *uproszczona*:

```
int sum;

for(i = 0, sum = 0; i < n; sum += array[i++]);
```

Zagadka:

```
int sum;

for(i = 0, sum = 0; i < n; i += 2)
  if(array[i] > 0)
    if(!(array[i] % 3))
      sum += array[i];
```

Wyznaczanie sumy co drugiego, dodatniego elementu tablicy, podzielnego przez 3:

```
int sum;

for(i = 0, sum = 0; i < n; i += 2)
  if(array[i] > 0)
    if(!(array[i] % 3))
      sum += array[i];
```

Kopiowanie zawartości tablic (tablice mają te same rozmiary):

```
for(i = 0; i < n; i++)
  b[i] = a[i];
```

wersja *uproszczona*:

```
i = n;
while(--i >= 0)
  b[i] = a[i];
```

W języku C nazwy tablic są traktowane w specyficzny sposób, jest to wskaźnik na pierwszy element tablicy (więcej później).

Z tego powodu można definiować parametry formalne **bez rozmiaru**. Taki parametr może przyjąć tablicę o **dowolnym rozmiarze**.

Przekazywanie tablic jako parametry może wyglądać jak przez wartość (pozornie).

Alternatywne kopiowanie tablic

Tablice to spójne obszary pamięci operacyjnej, dlatego można do ich kopiowania używać funkcji `memmove()` lub `memcpy()` (nagłówek `mem.h`):

```
memmove(b, a, n * sizeof(int));
```

lub

```
memmove(b, a, n * sizeof(b[0]));
```

`memmove()` kopiuje blok `n` bajtów z lokalizacji źródłowej do docelowej. Lokalizacje te mogą się nakładać.

`memcpy()` kopiuje blok `n` bajtów z lokalizacji źródłowej do docelowej. Gdy lokalizacje te się nakładają, działanie funkcji jest niezdefiniowane. Funkcja `memcpy()` jest szybsza niż `memmove()`.

```
memcpy(b, a, n * sizeof(int));
```

lub

```
memcpy(b, a, n * sizeof(b[0]));
```

Na marginesie, alternatywa dla iteracyjnego zerowania tablicy – można do tego wykorzystać funkcję `memset()`:

```
memset(a, 0, n * sizeof(a[0]));
```

`memset()` wypełnia `n` bajtów obszaru pamięci bajtem o zadanej wartości.

Stałe

Stała jest to symbol, któremu przypisana wartość (liczbowa, tekstowa, itp.) nie może być zwykle zmieniana podczas wykonywania programu. Choć wartość ta jest określana tylko raz, można się do niej odwoływać w programie wielokrotnie.

Stosowanie stałej zamiast wielokrotnie tych samych wartości, wyrażeń stałych, literałów itp., nie tylko ułatwia konserwację kodu, ale i dostarcza dla niej nazwę opisową, zwiększając czytelność kodu.

Stała jest często mylona z literałem, który jest zapisem danej wartości w danym punkcie programu. Stała jest natomiast identyfikatorem przypisanym do danego literału.

Wartość stałej w zależności od języka może być:

- znana na etapie kompilacji i nie może się zmienić w trakcie działania programu,
- ustawiona jednorazowo (jeśli dotyczy stałego wskaźnika), a potem nie może się zmienić, jednak obiekt wskazywany przez ten stały wskaźnik może zmieniać swoje wartości (`const` w C++),
- ustawiona jednorazowo (jeśli dotyczy przekazywania parametrów funkcji przez stałą), a potem również nie może się zmienić, ponieważ takie parametry są przekazywane przez wartość (`const` w C/C++).

W językach C/C++ nie ma typowych stałych, w zamian używa się dyrektywy preprocesora #define lub literałów zmiennych (zmiennych wraz ze wszystkimi ich własnościami), które są jest traktowane jak stała.

```
#define PI 3.1415
```

Dyrektywa #define jest poleceniem dla preprocesora, aby ten odpowiednio zmodyfikował tekst kodu źródłowego przed przekazaniem go kompilatorowi – czyli zastąpił odpowiednią nazwę określonym tekstem.

```
#define PI 3.1415

const float foo = 2.7182 * sin(PI);
```

const zabrania zmiany wartości zmiennej w trakcie działania programu. Nie mamy jednak gwarancji, że taka stała będzie miała tę samą wartość przez cały czas wykonania, możliwe jest bowiem dostanie się do wartości stałej (miejsca jej przechowywania w pamięci) pośrednio – za pomocą wskaźników.

Można zatem dojść do wniosku, że słowo kluczowe const służy tylko do poinformowania kompilatora, aby ten nie zezwalał na jawną zmianę wartości stałej.

Próba modyfikacji wartości stałej ma niezdefiniowane działanie (*undefined behaviour*) i w związku z tym może się powieść lub nie, może też spowodować jakieś subtelne zmiany, które w efekcie spowodują, że program będzie źle działał.

Konwersja typów

Konwersja typu, zmiana typu, rzutowanie typu, przekształcenie typu – konstrukcja programistyczna umożliwiająca traktowanie danej pewnego, konkretnego typu, jak daną innego typu.

Konwersją będziemy też nazywać zmianę tej danej albo jej reprezentacji w pamięci operacyjnej, aby wartość tej danej, odpowiadała według przyjętych kryteriów odwzorowania, danej innego, wybranego typu.

Pojęcie konwersji odnosi się także do sytuacji wyboru, rzutowania danych, które nie posiadają przypisanego typu, na wybrany, konkretny typ, celem interpretacji tych danych.

W językach programowania wartości, dane (reprezentowane przez np. literał, wyrażenie, zmienną, parametry, itd.), mogą mieć przypisane różne atrybuty, w szczególności typ danych.

Fizyczną reprezentacją danych jest ciąg bitów, a atrybuty przypisane danej, decydują między innymi o tym:

- jak długi ciąg bitów stanowi określoną daną,
- w jaki sposób jest interpretowany ciąg bitów określonej długości, stanowiący daną,
- ten sam ciąg bitów może być interpretowany zarówno jako liczba, łańcuch znaków, wartość logiczna, itd.

Konwersje typu mogą być rozróżniane według różnych kryteriów podziału.

- Podział ze względu na sposób specyfikacji:
  - jawne,
  - niejawne.
- Podział według konstrukcji programistycznej:
  - automatyczne,
  - wymuszone,
  - operator konwersji,
  - podprogram,
  - referencja.
- Podział według sposobu realizacji:
  - konwersje bez zmiany sposobu reprezentacji danej w pamięci i rozmiaru danej – tylko zmiana interpretacji,
  - konwersje ze zmianą atrybutu rozmiaru danej, bez zmiany sposobu interpretacji danej,
  - konwersje ze zmianą sposobu reprezentacji bez zmiany interpretacji danej,
  - konwersje ze zmianą sposobu interpretacji danej.
- Podział ze względu na utratę informacji:
  - konwersje bez utraty informacji,
  - konwersje z częściową utratą informacji.

Kilk przykładów niejawnego rzutowania:

```
int i = 42.7;              // konwersja z double do int
float f = i;               // konwersja z int do float
double d = f;              // konwersja z float do double
unsigned u = i;            // konwersja z int do unsigned int
f = 4.2;                   // konwersja z double do float
i = d;                     // konwersja z double do int
char *str = "foo";         // konwersja z const char* do char*
const char *cstr = str; // konwersja z char* do const char*
void *ptr = str;           // konwersja z char* do void*
```

Podczas konwersji zmiennych musimy liczyć się z możliwą utratą informacji, jak to ma miejsce w pierwszej linijce.

Niejawna konwersja z typu `const char*` do typu `char*` nie jest dopuszczana przez standard C, jednak literały napisowe (które są typu `const char*`) stanowią tutaj wyjątek.

Wynika on z faktu, że były one używane na długo przed wprowadzeniem słówka `const`.

Do jawnego wymuszenia konwersji służy jednoargumentowy operator rzutowania (), np. (notacja rzutowa z C):

```
double d = 3.14;
int iD = (int)d;
```

```
int val = 3;
float fVal = (float)val;
```

W języku C++ stosuje się notację funkcyjną:

```
double d = 3.14;
int iD = int(d);
```

```
int val = 3;
float fVal = float(val);
```

Dodatkowo wiele bibliotek dodatkowo oferuje funkcję do konwersji odpowiednich typów, zgodne z ich algorytmami zamiany. Na przykład po włączaniu pliku nagłówkowego `stdlib.h` będziemy mieli do dyspozycji m.in.:

- `double atof(const char *str)` – funkcja pobiera liczbę w postaci ciągu znaków, a następnie zwraca jej wartość typu `double`,
- `int atoi(const char *str)` – funkcja pobiera liczbę w postaci ciągu znaków, a następnie zwraca jej wartość typu `int`,
- `long atol(const char *str)` – funkcja pobiera liczbę w postaci ciągu znaków, a następnie zwraca jej wartość typu `long`.

Operator w programowaniu konstrukcja językowa jednoargumentowa, bądź wieloargumentowa zwracająca wartość.

Do podstawowych operatorów, będących elementem większości języków programowania, należą operatory: **przypisania**, **arytmetyczne**, **relacji** (porównania), **logicznie**.

Główne cechy opisujące operator to:

- liczba i typy argumentów,
- typ wartości zwracanej,
- wykonywane działanie,
- priorytet,
- łączność lub jej brak,
- umiejscowienie operatora względem operandów.

Operator przypisania

Przypisanie (podstawienie) jest to operacja nadania, umieszczenia, wpisania do określonej **L-wartości** (jest to wartość, która istnieje dłużej niż przez jedno wyrażenie i można pobrać jej adres, jest nią też zmienna) nowej wartości.

Przypisanie może zostać dokonane:

- instrukcją przypisania,
- operatorem przypisania,
- innym operatorem,
- w inicjalizacji zmiennej,
- w wywołaniu podprogramu,
- w wyniku efektów ubocznych,
- w instrukcji wejścia.

Operator przypisania to jednen z podstawowych operatorów prostych występujących w językach programowania.

Zwykle nie jest słowem kluczowym, choć istnieją języki programowania wymagające lub zezwalające opcjonalnie na użycie słowa kluczowego.

W C/C++ operatorem przypisania jest znak równości =.

Operator przypisania powoduje przypisanie, i zwraca wartość równą wartości przypisanej do L-wartości. Instrukcja przypisania nie zwraca wartości.

Operator przypisania może wystąpić w instrukcji przypisania ale może także wystąpić wewnątrz wyrażeń, tak jak każdy inny operator.

Operator przypisania w C/C++ jest **lewostronnie łączny**. Umożliwia to łączenie przypisań:

```
int i = 5, j, k, l;

l = k = j = i;
```

Zamiast:

```
j = i + 5;
k = j + 10;
l = k * 2;
```

można napisać:

```
l = (k = (j = i + 5) + 10) * 2;
```

Często zamiast:

```
x = sin(alfa);

if(x == 0)
{
   ...
}
```

stosuje się:

```
if((x = sin(alfa)) == 0)
{
   ...
}
```

Operator arytmetyczny to operator, który działając na podanych argumentach reprezentujących wartości liczbowe, w wyniku zwraca również wyznaczoną wartość liczbową, realizując podstawowe operacje arytmetyki.

To jakie operatory arytmetyczne są dostępne w konkretnym języku programowania zależy od jego składni, a to jakie są zasady ich stosowania, w tym priorytet tych operatorów i kolejność opracowywania argumentów, od przyjętej implementacji języka.

Zróżnicowany jest również sposób zapisu operatorów arytmetycznych. Stosuje się zapis, bądź za pomocą symboli (znaku lub znaków nie będących literami), w konwencji zbliżonej do matematycznej, bądź zdecydowanie rzadziej w postaci słów kluczowych.

Operatory arytmetyczne realizują następujące operacje arytmetyczne (przykłady):

- jednoargumentowe:
  - zmiana znaku liczby (wyznaczenie liczby przeciwnej),
  - zachowanie znaku liczby,
  - inkrementacja,
  - dekrementacja,
- dwuargumentowe:
  - dodawanie,
  - odejmowanie,
  - mnożenie,
  - dzielenie,
  - dzielenie całkowitoliczbowe,
  - reszta z dzielenia całkowitoliczbowego,
  - potęgowanie.

operatory arytmetyczne	
*	mnożenie
/	dzielenie
&	reszta z dzielenia całkowitoliczbowego
+	dodawanie
-	odejmowanie, zmiana znaku liczby

Wyrażenie arytmetyczne jest to językach programowania dowolne wyrażenie typu liczbowego. Może być ono złożone ze zmiennych, liczb, funkcji, symboli działań (tu: operatorów), itp.

```
float wynik;

wynik = 10 / 3;     // 3 wynik zalezy od typow literalow i zmiennej
wynik = 10.0 / 3.0 // 3.333333
```

```
int result;

result =  10 * 3; // 30
result =  10 / 3; // 3
result =  10 % 3; // 1
result =  10 + 3; // 13
result =  10 - 3; // 7
result =  5;      // 5
result = -result; // -5
```

Kolejność (priorytety) wykonywania działań jest taka jak w matematyce, można ją regulować za pomocą nawiasów okrągłych. Operatory równoważne wykonywane są od strony lewej do prawej (należy wziąć pod uwagę wiązanie).

wybrane funkcje matematyczne z `math.h`	
`sin()`	sinus
`cos()`	cosinus
`tan()`	tangens
`asin()`	arcus sinus
`acos()`	arcus cosinus
`atan()`	arcus tangens
`exp()`	eksponent (e^x)
`log()`	logarytm naturalny
`log10()`	logarytm dziesiętny
`pow()`	potęga
`sqrt()`	pierwiastek kwadratowy
`ceil()`	zaokrąglenie do liczby całkowitej w górę
`floor()`	zaokrąglenie do liczby całkowitej w dół
`round()`	zaokrąglenie do najbliższej liczby całkowitej
`fabs()`	wartość bezwzględna

Operatory relacji i wyrażenia logiczne

Operator **relacji** jest to operator który działając na podanych argumentach, w wyniku zwraca wartość logiczną, określającą spełnienie bądź nie spełnienie reprezentowanej przez ten operator relacji zachodzącej między argumentami.

Wynikiem działania operatora relacji jest więc wartość reprezentująca zgodnie z zasadami obowiązującymi w składni języka programowania jedną z wartości logicznych: **prawdę** (*true*) lub **fałsz** (*false*).

W języku C będą to wartości całkowitoliczbowe odpowiednio: wartość różna od zera (1) i zero (0).

W językach programowania dostępne są operatory, które badają relacje:

- równości,
- nierówności,
  - negacji równości,
  - nierówności ostrych,
    - mniejsze,
    - większe,
  - nierówności nieostrych,
    - mniejsze lub równe,
    - większe lub równe,
- przynależności (zawierania),
- równoważności.

operatory relacji	
==	czy równe?
!=	czy różne?
>	czy większe?
<	czy mniejsze?
>=	czy większe lub równe?
<=	czy mniejsze lub równe?

Operatory relacji mogą działać na wszystkich typach danych, jednak najczęściej odnoszą się do danych numerycznych.

Porównanie danych całkowitych nie powinno budzić zastrzeżeń, ponieważ operacje wykonywane są w sposób naturalny.

Należy zachować ostrożność przy porównywaniu wartości zmiennoprzecinkowych.

```
double first, second;

first = 1.00000000000001;
second = 1.00000000000002;

if(first == second)
{
   ...
}

if(fabs(first - second) < 0.000001)
{
   ...
}
```

Operator **logiczny** to operator, który działając na argumentach reprezentujących wartości logiczne, w wyniku zwraca również wartość logiczną, realizując podstawowe operacje algebry Boole'a.

Dostępność operatorów logicznych, priorytet i kolejność opracowywania argumentów zależy od przyjętej implementacji języka.

Zróżnicowany jest również sposób zapisu operatorów logicznych, stosuje się zapis, bądź w postaci słów kluczowych, bądź symboli (znaku lub znaków nie będących literami).

Operatory logiczne realizują następujące operacje logiczne:

- jednoargumentowe:
  - negacja,
- dwuargumentowe:
  - koniunkcja,
  - alternatywa,
  - alternatywa wykluczająca,
  - implikacja,
  - ekwiwalencja.

operatory	logiczne
!	negacja (zaprzeczenie)
&&	koniunkcja
||	alternatywa

!

argument	wynik
prawda	fałsz
fałsz	prawda

&&

lewy argument	prawy argument	wynik
prawda	prawda	prawda
prawda	fałsz	fałsz
fałsz	prawda	fałsz
fałsz	fałsz	fałsz

||

lewy argument	prawy argument	wynik
prawda	prawda	prawda
prawda	fałsz	prawda
fałsz	prawda	prawda
fałsz	fałsz	fałsz

Wyrażenie logiczne jest to językach programowania dowolne wyrażenie zawierające operatory relacji i operatory logiczne, stałe, zmienne logiczne, którego wynik jest typu logicznego (prawda lub fałsz).

Wyrażenia logiczne mogą zawierać wyrażenia innego typu, np. arytmetyczne.

```
int result;

result = (5 < 3 * 2) && ('L' > 'A');
result = (5 <> 3 * 2) || ('L' == 'A');
result = !(2 + 2 == 5);
```

```
int value, even, range, result;

value = 25;

even = (liczba % 2) == 0;
range = ((liczba >= 10) && (liczba <= 20));
result = even && range;
```

Dla większości operatorów dwuargumentowych:

```
*   /   %   +   -   <<   >>   &   ^   |
```

można wykorzystać specjalne operatory przypisania (*shorthands*), pozwalające skrócić zapis:

- mnożenie: `a *= b` zamiast `a = a * b`,
- dzielenie: `a /= b` zamiast `a = a / b`,
- reszta z dzielenia całkowitoliczbowego: `a %= b` zamiast `a = a % b`,
- dodawanie: `a += b` zamiast `a = a + b`,
- odejmowanie: `a -= b` zamiast `a = a - b`,
- przesunięcie bitowe w lewo: `a <<= b` zamiast `a = a << b`,
- przesunięcie bitowe w prawo: `a >>= b` zamiast `a = a >> b`,
- koniunkcja bitowa: `a &= b` zamiast `a = a & b`,
- bitowa różnica symetryczna: `a ^= b` zamiast `a = a ^ b`,
- alternatywa bitowa: `a |= b` zamiast `a = a | b`.

Inkrementacja i dekrementacja

Operator **inkrementacji** `++` powoduje zwiększenie wartości argumentu o 1.
Operator **dekrementacji** `--` powoduje zmniejszenie wartości argumentu o 1.

Oba operatory występują w wersji **prefixowej** i **postfixowej** – są wymieniane przed lub za argumentem. Nie zmienia to działania operatora, tylko wpływa na *moment* ich wykonania:

- wersja przedrostkowa zwiększa (zmniejsza) wartość argumentu przed użyciem jego wartości,
- wersja przyrostkowa zwiększa (zmniejsza) wartość argumentu po użyciu jego wartości.

```
int a = 5, b;

b = ++a; // a = 6, b = 6
```

```
int a = 5, b;

b = a++; // a = 6, b = 5
```

Ponieważ operatory te wykonują niejawną instrukcję przypisania to wyrażenia typu:

```
(a + b)++;
```

nie są poprawne.

Na wyrażenia operatorowe trzeba uważać (poniższe przykłady są poprawne, ale co one robią?):

```
x=a+++(b+=c);
```

```
x*=(a!='a')?a--:++a;
```

```
i=++i---i+++i+i--;
```

Operator warunkowy

Często spotyka się *symetryczne* instrukcje warunkowe:

```
if(delta > 0)
  isSolution = 1;
else
  isSolution = 0;
```

```
if(a > b)
  max = a;
else
  max = b;
```

Można je zapisać krócej z wykorzystaniem operatora warunkowego:

```
isSolution = (delta > 0) ? 1 : 0
```

```
max = (a > b) ? a : b;
```

Priorytety niektórych operatorów

operator, wiązanie	
`()` (funkcja, zmiana kolejności) `[]` (odwołanie do tablic)	
`->` `.` (składowe klas)	L
`*` (wskaźnik) `&` (adres) `+` `-` (znak) `!` (negacja) `~` (negacja bitowa)	
`++` `--` (inkrementacja, dekrementacja)	P
`*` (mnożenie) `/` (dzielenie) `%` (reszta)	L
`+` (dodawanie) `-` (odejmowanie)	L
`<<` `>>` (przesunięcia bitowe)	L
`<` `>` `<=` `>=` (porównania)	L
`==` `!=` (równość, nierówność)	L
`&` (koniunkcja bitowa)	L
`^` (bitowa różnica symetryczna)	L
`|` (alternatywa bitowa)	L
`&&` (koniunkcja)	L
`||` (alternatywa)	L
`?` (operator warunkowy)	P
`=` `+=` `-=` `/=` `<<=` `>>=` `&=` `^=` `|=` (przypisania)	P
`,` (połączenie)	L

Litery **L** i **P** określają kolejność wykonywania operacji danego typu. L – łączność lewostronna – a więc zapis:

```
a + b + c + d;
```

jest interpretowany jak:

```
(((a + b) + c) + d);
```

P – łączność prawostronna np. dla przypisania powoduje że zapis:

```
a = b = c = d;
```

jest interpretowany jak:

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

Stąd wyrażenia:

```
a = 1 = b; // jest niepoprawne
a = b = 1; // jest poprawne
```

Instrukcja wyrażeniowa

Instrukcja wyrażeniowa – to każde poprawne wyrażenie w języku C (również wyrażenie puste) zakończone znakiem średnika.

Wykonanie takiej instrukcji polega na wyznaczeniu wartości danego wyrażenia.

```
x = 0;
```

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

```
a + b;
```

```
;
```

Instrukcja złożona – zwana inaczej blokiem, to lista instrukcji ujęta w nawiasy klamrowe {}.

Blok traktowany jest jako pojedyncza instrukcja. Identyfikator zadeklarowany w obrębie bloku ma jego zakres.

Bloki mogą być zagnieżdżone do dowolnej głębokości. W obrębie zagnieżdżonych bloków następuje przesłanianie nazw.

```
{
  int k = 2;
  {
    float k = 10.2;
    printf_s("k = %f", k); // k = 10.2
  }
  printf_s("k = %d", k); // k = 2
}
```

Instrukcja warunkowa

Instrukcja warunkowa jest elementem języka programowania, który pozwala na wykonanie różnych obliczeń (zadań) w zależności od tego czy zdefiniowane przez programistę **wyrażenie logiczne** jest prawdziwe, czy fałszywe.

if - then

```
int a, b;

a = 1;
b = 2;

if(a + b == 3)

  printf_s("Suma jest rowna 3")
```

if - then - else

```
float take, bonus;

take = 29999.99;

if(take >= 30000)
{
  bonus = 0.05 * take;
  printf_s("Premia w wysokosci: %f", bonus);
}
else
{
  bonus = 0;
  printf_s("Premii nie będzie!");
}
```

if - then - else if - then - else

```
int a, b, c;
float delta, x0, x1, x2;

a = 1;
b = 8;
c = 2;

delta = b * b - 4 * a * c;

if(delta > 0)
{
  x1 = (-b - sqrt(delta)) / (2 * a);
  x2 = (-b + sqrt(delta)) / (2 * a);
}
else

  if(delta == 0)

    x0 = -b / (2 * a);

  else

    printf_s("Brak pierwiastkow rzeczywistych");
```

Uwaga na zagnieżdżone instrukcje warunkowe

Która wersja sugerowana przez wcięcia jest poprawna?

```
if(value >= 0)
  if(value > 0)
    printf_s("Liczba dodatnia");
else
  printf_s("Liczba ujemna");
```

```
if(value >= 0)
  if(value > 0)
    printf_s("Liczba dodatnia");
  else
    printf_s("Zero");
```

Instrukcja wyboru

Instrukcja wyboru jest instrukcją umożliwiającą wybór instrukcji do wykonania spośród wielu opcji.

Składnia instrukcji wyboru różni się w zależności od języka programowania, lecz można wyróżnić w niej charakterystyczne elementy:

- nagłówek instrukcji wyboru – słowo kluczowe rozpoczynające instrukcję, nazwa zmiennej lub wyrażenie, na podstawie którego następuje wybór,
- ciało instrukcji wyboru – kolejne instrukcje (bloki instrukcji) podlegające selekcji, poprzedzone frazami zawierającymi wartości, listy lub zakresy porównywane z wyrażeniem (zmienną) z nagłówka instrukcji,
- opcjonalnie fraza domyślna, wykonywana gdy żadna z fraz nie spełni warunku,
- koniec instrukcji wyboru.

```
int body;

printf_s("1. SEDAN\n");
printf_s("2. COUPE\n");
printf_s("3. SUV\n");

printf_s("Podaj typ nadwozia: ");
scanf_s("%d", &body);

if(body == 1)
  printf_s("Lubisz eleganckie limuzyny!");

if(body == 2)
  printf_s("Pewnie lubisz szybka jazde!");

if(body == 3)
  printf_s("Widze, ze ciagnie Cie w teren!");
```

W powyższym przykładzie wykorzystana jest wielokrotnie sama zmienna, porównanie następuje z wartościami znanymi na etapie kompilacji.

```
int body;

printf_s("1. SEDAN\n");
printf_s("2. COUPE\n");
printf_s("3. SUV\n");

printf_s("Podaj typ nadwozia: ");
scanf_s("%d", &body);

switch(body)
{
  case 1:
    printf_s("Lubisz eleganckie limuzyny!");
    break;

  case 2:
    printf_s("Pewnie lubisz szybka jazde!");
    break;

  case 3:
    printf_s("Widze, ze ciagnie Cie w teren!");
    break;
}
```

```
enum bodyType
{
  SEDAN = 1,
  COUPE,
  SUV
};

int body;

printf_s("1. SEDAN\n");
printf_s("2. COUPE\n");
printf_s("3. SUV\n");

printf_s("Podaj typ nadwozia: ");
scanf_s("%d", &body);
...
```

```
...
switch(body)
{
  case SEDAN:
    printf_s("Lubisz eleganckie limuzyny!");
    break;

  case COUPE:
    printf_s("Pewnie lubisz szybka jazde!");
    break;

  case SUV:
    printf_s("Widze, ze ciagnie Cie w teren!");
    break;

  default:
    printf_s("Chyba wolisz pedalowac!");
}
```

Pętle

Pętla to jedna z podstawowych konstrukcji programowania strukturalnego (obok instrukcji warunkowej i instrukcji wyboru). Umożliwia cykliczne wykonywanie ciągu instrukcji:

- określoną liczbę razy (pętla **iteracyjna**, licznikowa),
- do momentu zajścia pewnych warunków (pętla **repetycyjna**, warunkowa),
- dla każdego elementu kolekcji (pętla **foreach**, po kolekcji),
- w nieskończoność.

Pętla iteracyjna, to rodzaj pętli, w której następuje wykonanie określonej liczby iteracji. Do kontroli przebiegu wykonania pętli iteracyjnej stosuje się specjalną zmienną, którż nazywa się **zmienną sterująca**, **kontrolną** lub **licznikową**.

W ramach pętli przejście do kolejnej iteracji wiąże się ze zmianą wartości zmiennej sterującej o określoną wielkość i sprawdzenie warunku, czy nowa wartość zmiennej sterującej znajduje się nadal w dopuszczalnym zakresie wartości, określonym dla tej zmiennej.

Kolejność wykonywania działań jest następująca:

- przypisanie wartości początkowej do zmiennej sterującej,
- sprawdzenie, czy wartość zmiennej sterującej mieści się w dopuszczalnym zakresie wartości, tzn. jej wartość jest równa lub mniejsza od wartości granicznej, albo jest równa lub większa od wartości granicznej:
  - jeżeli wartość zmiennej sterującej nie mieści się w dopuszczalnym zakresie wartości to kończy wykonywanie pętli,
  - jeżeli wartość zmiennej sterującej mieści się w dopuszczalnym zakresie wartości to nie przerywa działania,
- wykonuje iterację,
- zmienia wartość zmiennej sterującej o zadany krok, wartość ta może być zwiększana lub zmieszana.

Zmienna sterująca służy do kontroli przebiegu realizacji pętli iteracyjnej. Przyjmuje ona kolejne wartości z zadanego zakresu zmieniane o określoną wartość kroku.

W różnych językach programowania mogą być stawiane określone wymagania i ograniczenia dotyczące zmiennych sterujących. Ograniczenia te mogą dotyczyć takich atrybutów tej zmiennej jak np. dozwolony typ danych, zasięgu.

Zmiana wartości zmiennej sterującej może nastąpić:

- automatycznie, w sposób ukryty,
- jawnie, w kodzie bloku pętli, o ile dany język programowania dopuszcza taką konstrukcję.

Ogólniejszą konstrukcją jest pętla **repetycyjna** (warunkowa), która jest wykonywana, aż do odpowiedniej zmiany warunków.

Warunek zawarty w definiowanej pętli jest pewnym wyrażeniem, które najczęściej zwraca wartość typu logicznego. Istnieją języki programowania, w których składnia nie przewiduje takiego typu danych.

W językach tych stosuje się wyrażenia zwracające pewną wartość innego typu, która następnie podlega odpowiedniej interpretacji, np. wartość zero może być utożsamiana z wartością **false** typu logicznego, a pozostałe wartości z wartością **true**.

Zapis wyrażenia, oraz typ wartości wyrażenia, zależny jest od składni konkretnego języka programowania. Powszechnym jest zapis wyrażeń kontrolnych analogicznie do zapisu tych wyrażeń dla instrukcji warunkowej.

Szczególne znaczenie mają w tym przypadku operatory porównań, choć warunek może zostać wyrażony także całkiem inaczej, np. jako wywołanie funkcji zwracającej wartość logiczną, bądź jako identyfikator zmiennej logicznej, której wcześniej przypisano rezultat ewaluacji wyrażenia warunkowego.

Ponadto operator logiczny realizujący operację negacji pozwana na rekompensatę ewentualnego braku pętli powtarzanej przy spełnieniu lub niespełnieniu warunku, gdyż zastosowanie tego operatora do podanego warunku jest równoważne zastosowaniu frazy przeciwnej.

Warunki decydujące o kontynuacji lub zaprzestaniu wykonywania pętli mogą być sprawdzane:

- na początku pętli, przed wykonaniem pierwszej instrukcji zawartej w bloku definiowanej pętli,
- wewnątrz pętli, w jej bloku, po wykonaniu części instrukcji,
- na końcu pętli, po wykonaniu wszystkich instrukcji zawartych w bloku definiowanej pętli.

Jeżeli warunek jest sprawdzany na początku pętli, to może nastąpić taka sytuacja, że instrukcje zawarte w pętli nigdy nie zostaną wykonane. Będzie to miało miejsce w sytuacji, gdy przy pierwszym wykonaniu warunek nie będzie spełniony.

Inaczej jest, gdy warunek jest sprawdzany na końcu pętli. W tym przypadku instrukcje zawarte w pętli zostaną wykonane co najmniej jeden raz.

Często pożądane jest, aby instrukcje pętli zostały wykonane dla każdego elementu tablicy, kolekcji itp.

Oczywiście można to zrobić za pomocą pętli iteracyjnych lub repetycyjnych, ale często szybszym i bardziej przejrzystym sposobem jest użycie pętli typu **foreach**, która zwalnia programistę z obowiązku *ręcznego* iterowania po kolekcji.

Działanie pętli foreach, polega na powtarzaniu kolejnych iteracji dla wszystkich elementów wybranego kontenera danych, takiego jak, np. tablica, lista, kolekcja, kolejka, itp.

Pętla taka automatycznie przed przejściem do wykonania kolejnej iteracji przypisuje zadanej w nagłówku pętli zmiennej sterującej wartość kolejnego elementu.

Działanie tej pętli polega na wykonaniu następujących kroków:

- ustaleni jako bieżący, pierwszego element kontenera danych, jeżeli kontener jest pusty, zakończenie działanie pętli,
- przypisanie wartość bieżącego elementu do zmiennej sterującej,
- wykonanie iteracji,
- sprawdzenie czy istnieje kolejny element w kontenerze:
  - jeżeli istnieje, to ustala jako bieżący kolejny element kontenera i wykonuje iterację,
  - jeżeli nie istnieje to kończy działanie.

do - while

```
int index, counter;

index = 20;
counter = 0;

do
{
  index--;
  counter++;
}
while(indeks > 10);

printf_s("Indeks = %d, licznik = %d", index, counter);
// index = 10, counter = 10
```

Instrukcja stanowiąca ciało iteracji wykona się **przynajmniej raz**.

Wyrażenie występujące w nawiasach określa **warunek kontynuacji**, zatem iteracja kończy się gdy wartość wyrażenia będzie zerowa.

Innymi słowy, pętla wykonuje się dopóty, dopóki **warunek jest prawdziwy**.

while

```
int index, counter;

index = 20;
counter = 0;

while(index > 10)
{
  index--;
  counter++;
}

printf_s("Indeks = %d, licznik = %d", index, counter);
// index = 10, counter = 10
```

Instrukcja stanowiąca ciało iteracji może **nie wykonać się ani razu**.

Wyrażenie występujące w nawiasach określa **warunek kontynuacji**, zatem iteracja kończy się gdy wartość wyrażenia będzie zerowa.

Innymi słowy, pętla wykonuje się dopóty, dopóki **warunek jest prawdziwy**.

Iteracja `for()` w języku C/C++ stanowi uogólnienie schematu iteracji `while()`.

```
int counter;

counter = 0; // inicjalizacja

while(index < 10) // warunek kontynuacji
{
  printf_s("Licznik = %d\n", counter); // cialo iteracji
  counter++; // modyfikacja zmiennej sterujacej
}
```

```
int counter;

for(counter = 0; counter < 10; counter++)
// inicjalizacja, warunek kontynuacji, modyfikacja zmiennej
  printf_s("Licznik = %d\n", counter); // cialo iteracji
```

W części inicjalizacyjnej w C++ wolno deklarować zmienne.

Poszczególne sekcje iteracji mogą być dowolnymi legalnymi, wyrażeniami w języku C. Mogą to też być wyrażenia przecinkowe.

```
int counter;

for(counter = 0; counter < 10; printf_s("Licznik = %d\n", counter),
  counter++);
```

Pętla `for()` może występować bez warunku kontynuacji.

```
char key;

printf_s("Wpisz dowolna litere, zero konczy iteracje.\n");

for ( ; ; )
{
  printf_s("Klawisz: ");
  key = getchar();

  while((getchar()) != '\n');

  if(key == '0')
    break;

  if(key >= '0' && key <= '9')
  {
    printf_s("Wpisales cyfre %c\n", key);
    continue;
  }
  printf_s("Wpisales litere %c o kodzie %d\n", key, key);
}
```

Instrukcje break i continue

Instrukcja `break` pozwala na **natychmiastowe zakończenie** nawet zagnieżdżonej, dowolnej instrukcji **pętli** lub **wyboru**.

Instrukcja `continue` pozwala na **przerwanie bieżącego przebiegu** dowolnej iteracji i przejście do wykonania następnego jej przebiegu.

W przypadku iteracji `while()` i `do while()` instrukcja powoduje przeniesienie sterowania do fazy testowania warunku kontynuacji.

W przypadku iteracji `for()` sterowanie przenoszone jest do wyrażenia modyfikującego a potem do testowania warunku.

Podprogramy

Podprogram – termin związany jest z programowaniem proceduralnym.
Podprogram to wydzielona część programu wykonująca jakieś operacje.
Podprogramy stosuje się, aby uprościć program główny, zwiększyć czytelność kodu.
Wartościową cechą podprogramu jest możliwość wielokrotnego jego wywołania.

W wielu językach programowania dzieli się podprogramy na funkcje i procedury.

Funkcja ma wykonywać obliczenia i zwracać jakąś wartość, nie powinna natomiast mieć żadnego innego wpływu na działanie programu.

Procedura natomiast nie zwraca żadnej wartości, zamiast tego wykonuje pewne działania.

Przez **zwracanie wartości** należy rozumieć możliwość użycia wywołania funkcji wewnątrz wyrażenia.

Procedury często też zwracają wartości, ale poprzez odpowiednie parametry. Podział ten występuje w językach takich jak **Pascal** i **Visual Basic**.

Oprócz powyższego podziału, wyróżnić także należy **podprogram główny**, czyli taki od którego rozpoczyna się wykonywanie skompilowanego programu.

W językach programowania zastosowano zasadniczo kilka różnych rozwiązań. Albo zdefiniowana jest w składni odrębna jednostka (`program` w **Pascalu**) albo stosuje się specjalną dyrektywę języka, informującą aby program łączący dany podprogram wybrał jako podprogram główny (`OPTIONS(MAIN)` w **PL**). W języku **C** i pokrewnych definiuje się zwykłą funkcję lecz o specjalnym identyfikatorze `main`.

W językach programowania stosuje się różne terminologie i oznaczenia podprogramów:

- w wielu językach programowania, a szczególnie tych, w których występuje tylko jeden rodzaj podprogramu, np. tylko funkcje, nie ma specjalnego oznaczenia (słowa kluczowego) podprogramu, przykładem jest język **C**,
- procedury:
  - `procedure`, np. **Pascal**, **Ada**, **Algol**, **PL**,
  - `subroutine`, np. **Fortran**, lub w skróconej postaci `sub`, np. **Basic** i **Visual Basic**,
  - `part`, `perform`,
- funkcje:
  - `function`, np. **Pascal**, **Ada**, **Visual Basic**,
  - `procedure` - `return()`, np. **PL**, **Algol**.

Podprogram może być identyfikowany:

- przez **nazwę** – identyfikator przypisany do podprogramu w jego deklaracji, jest to najczęściej spotykany przypadek w językach wysokiego poziomu (np.: **Ada**, **C**, **Visual Basic**),
- przez **etykietę** (**Basic**, **Visual Basic**),
- przez **liczbę** – literał całkowity (**Basic**, **Visual Basic**),
- przez **referencję** (adres) – w językach wysokiego poziomu takie wskaźniki przechowywane są w zmiennych typu **proceduralnego** lub **wskaźnikowego**.

Wywołanie podprogramu może być:

- **funkcyjne** – w wyrażeniu, do którego podprogram zwraca obliczoną wartość, taka forma wywołania dotyczy tylko podprogramów mających cechy funkcji, tzn. zwracających wartość,
- **proceduralne** – czyli jako instrukcja, poprzez nazwę z listą argumentów, lub po słowie kluczowym.

Konkretne implementacje języków często dopuszczają wywołanie funkcji w postaci proceduralnej, tzn. poza wyrażeniami. W tym przypadku zwracana przez podprogram wartość jest ignorowana (**C**).

Podprogram jako samodzielna, wydzielona część algorytmu, zazwyczaj musi komunikować się z otoczeniem. Taką komunikację realizuje się za pomocą:

- zmiennych globalnych,
- argumentów (parametrów aktualnych), przypisywanych zdefiniowanym w podprogramie,
- rezultatów funkcji (wartości zwracanych do miejsca wywołania),
- pól obiektu (dla metod danego obiektu),
- innych, rzadko stosowanych lub nie zalecanych (np.: obszarów wspólnych, zmiennych nakładanych).

Funkcje

```
#include <stdlib.h>
#include <stdio.h>

float squareArea(float side)
{
  return side * side;
}

int main()
{
  float d, p;

  printf_s("Podaj dlugosc boku: ");
  scanf_s("%f", &d);

  pole = squareArea(d);
  printf_s("Pole = %f", p);

  return EXIT_SUCCESS;
}
```

W języku C/C++ nie występuje podział podprogramów na procedury i funkcje. Wszystkie podprogramy są funkcjami.

Istnieje możliwość wykorzystywania funkcji jak procedur, bądź deklarowania funkcji tak, by przypominały procedury.

Słowo kluczowe `void`, będące nazwą typu, oznacza brak, jakiejkolwiek wartości.

Jeżeli typem rezultatu będzie typ określany słowem kluczowym `void`, to oznacza, iż funkcja nie udostępnia rezultatu – staje się wtedy czymś podobnym do procedury.

```
#include <stdlib.h>
#include <stdio.h>

float squareArea(float side)
{
  return side * side;
}

void printInfo(void)
{
  printf_s("Obliczam pole kwadratu\n");
  printf_s("Podaj dlugosc boku: ");
}
...
```

```
...
int main()
{
  float d, p;

  printInfo();
  scanf_s("%f", &d);

  p = squareArea(d);
  printf_s("Pole = %f", p);

  return EXIT_SUCCESS;
}
```

Jeżeli w miejscu listy parametrów formalnych występuje słowo kluczowe `void`, to oznacza, że funkcja nie posiada parametrów.

Jeżeli funkcja posiada rezultat, w ciele funkcji powinna wystąpić instrukcja `return`, a po niej, wyrażenie o typie zgodnym z typem rezultatu funkcji.

Na liście parametrów formalnych, dla każdego parametru określamy jego typ.

Nawiasy po nazwie funkcji są konieczne, nawet gdy funkcja nie ma parametrów. Nawiasy występują zarówno przy definicji, deklaracji jak i przy wywołaniu funkcji.

W języku C puste nawiasy oznaczają **zmienną liczbę parametrów**:

```
int foo(...)
{
   ...
}
```

```
int foo()
{
   ...
}
```

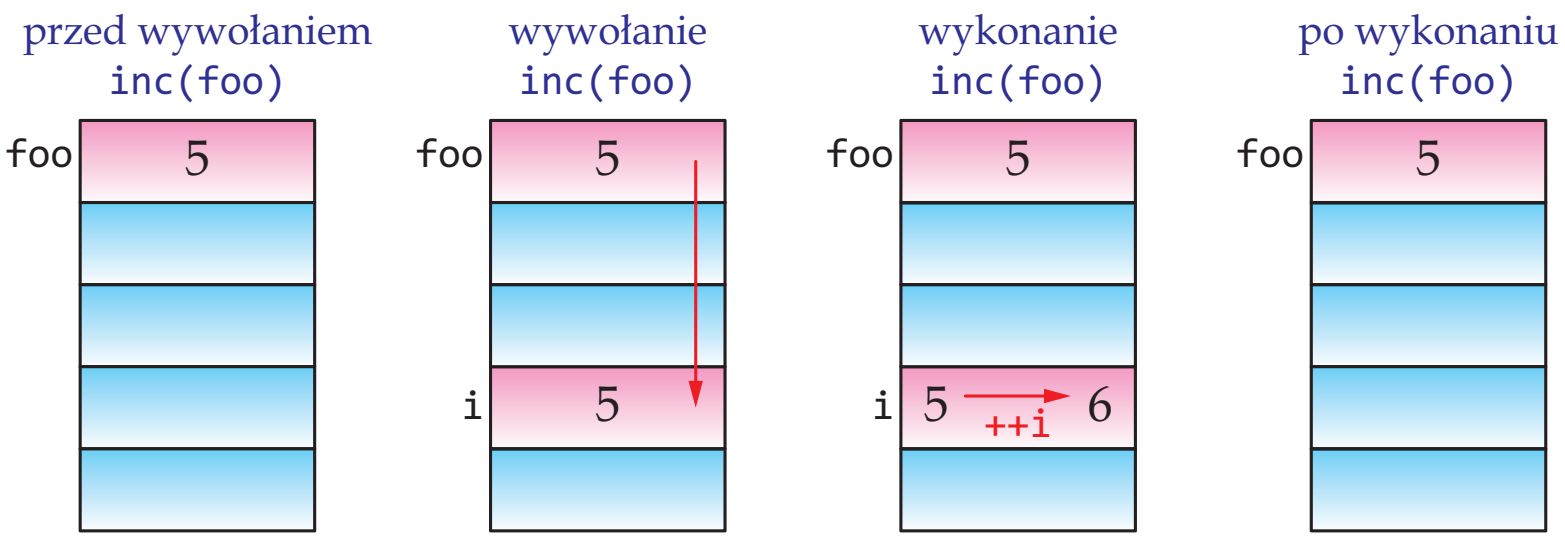
W języku C++ puste nawiasy oznaczają **brak parametrów**:

```
int foo(void)
{
   ...
}
```

```
int foo()
{
   ...
}
```

Przekazywanie parametrów

```
void inc(int i)
{
  ++i;
}

int foo = 5;

inc(foo);
printf_s("foo = %d", foo); // foo = 5
```



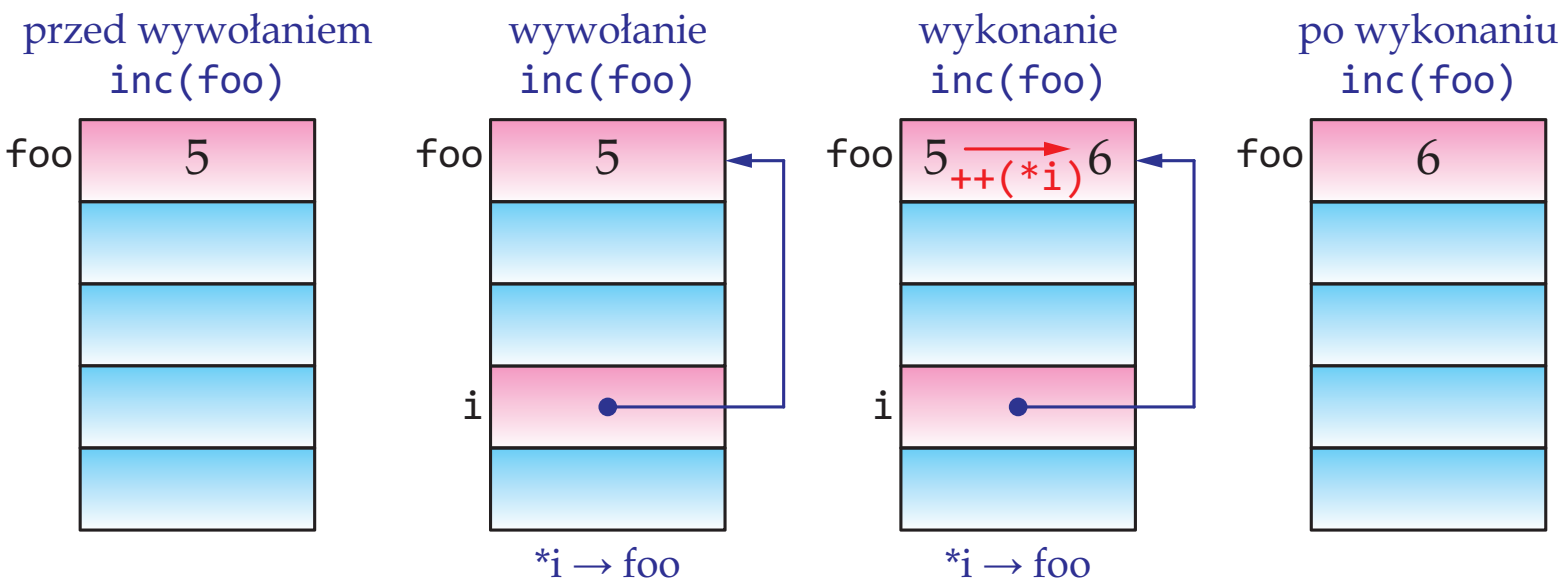
Przy przekazywaniu parametrów przez wartość, wartość **parametru aktualnego** wywołania funkcji **kopiowana** jest do **parametru formalnego** funkcji.

Po skopiowaniu parametr aktualny i formalny są od siebie **niezależne**.

Żadna modyfikacja parametru formalnego funkcji **nie przenosi się** na parametr aktualny wywołania.

Wnętrze funkcji nie jest w stanie zmodyfikować parametru formalnego funkcji.

```
void inc(int *i)
{
  ++(*i);
}

int foo = 5;

inc(&foo);
printf_s("foo = %d", foo); // foo = 6
```



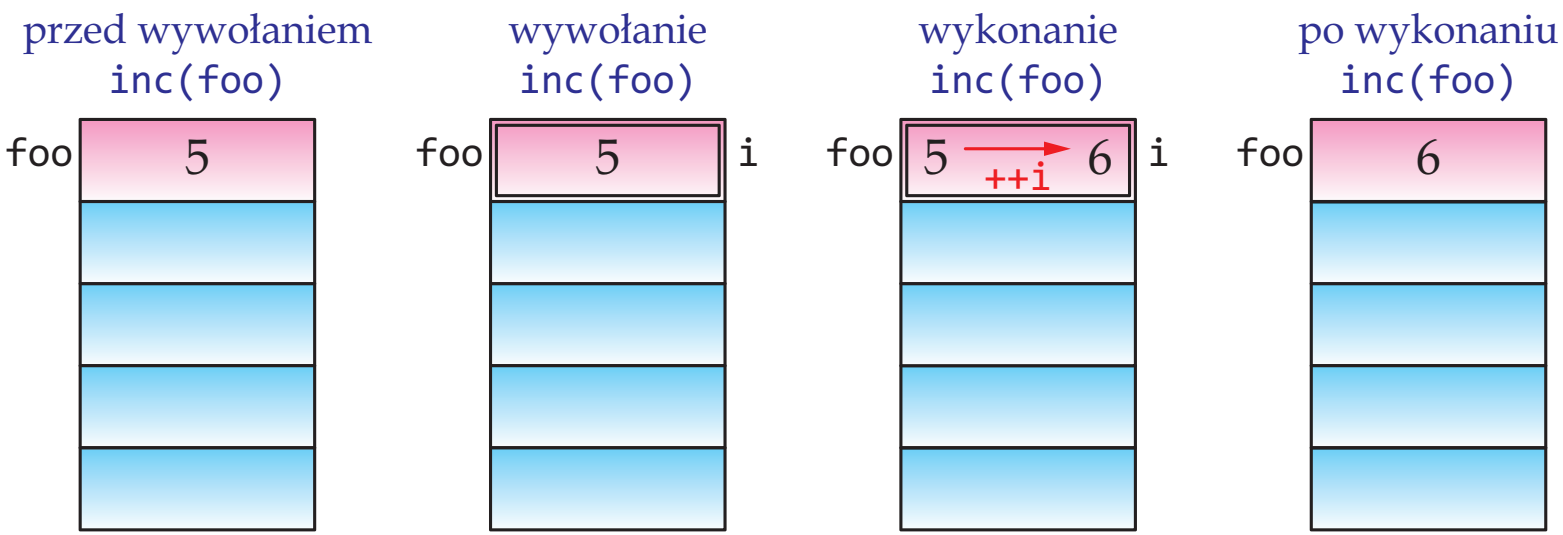
Inną formą przekazywania parametrów przez wartość jest argument **wskaźnikowy**.

Wskaźniki także są przekazywane przez **wartość**, wynika to z faktu że wskaźnik też jest typem zmiennej (zmienna wskaźnikowa).

Argumenty wskaźnikowe wskazują na zmienne **z poza funkcji**, więc wewnątrz funkcji nie są tworzone kopie.

Funkcja może zmodyfikować wiele zmiennych z poza funkcji (bez użycia `return`).

```
void inc(int &i)
{
  ++i;
}

int foo = 5;

inc(foo);
printf_s("foo = %d", foo); // foo = 6
```



Przy przekazywaniu parametrów przez referencję, **parametr aktualny** wywołania funkcji jest **nakładany** na **parametr formalny** funkcji.

Od tego momentu parametr aktualny i formalny odnoszą się do **tej samej lokalizacji** (adresu) w pamięci operacyjnej.

Każda modyfikacja parametru formalnego funkcji **przenosi się** na parametr aktualny wywołania.

Wnętrze funkcji może zmodyfikować parametr formalny funkcji.

Prototypy funkcji

Prototyp jest to struktura oprogramowania, która informuje kompilator lub interpreter języka programowania o możliwościach podprogramu (funkcja, procedura, metoda) lub klasy. Prototyp jest więc deklaracją oddzieloną od definicji w postaci samodzielnego nagłówka podprogramu.

Definicja funkcji:

```
float squareArea(float side)
{
  return side * side;
}
```

Prototyp, czyli deklaracja funkcji:

```
float squareArea(float);
```

Oczywiście zadeklarowana funkcja musi mieć gdzieś zapisaną definicję.
W prototypie nie trzeba podawać nazw argumentów, można zdefiniować tylko ich typy.

```
float squareArea(float);

int main()
{
  float d, p;

  printf_s("Podaj dlugosc boku: ");
  scanf_s("%f", &d);

  p = squareArea(d);
  printf_s("Pole = %f", p);

  return EXIT_SUCCESS;
}

float squareArea(float side)
{
  return side * side;
}
```

Starsze implementacje C dopuszczały wywoływanie funkcji wcześniej kompilatorowi nieznanych (nowsze czasem też pozwalają).

W trakcie kompilowania wywołania nieznanej funkcji przez domniemanie przyjmowano, że jej rezultatem jest wartość `int` i nic nie wiadomo na temat jej parametrów.

Nie pozwala to kompilatorowi kontrolować poprawności wywołania funkcji.

Więcej o strukturze programu

Zmienne automatyczne

Zmienne **automatyczne** (auto) mogą być definiowane na początku każdego bloku w C89. W standardzie C99 i C++ w każdym dozwolonym syntaktyką języka miejscu. Parametry formalne funkcji to też zmienne automatyczne.

Zmienne klasy auto **pojawiają** się wraz z wejściem sterowania do bloku w którym są zadeklarowane i **znikają** wraz z wyjściem sterowania z bloku.

Zmienne deklarowane wewnątrz bloku są automatycznymi, jeżeli nie podano klasy pamięci albo jawnie użyto specyfikatora `auto`.

Zmienne auto nie zachowują wartości pomiędzy swoimi kolejnymi kreacjami i mają przypadkowe wartości (jeśli nie zostały zainicjowane).

Zmienne automatyczne są lokowane na stosie. Stos ma ustalony i ograniczony rozmiar.

Potencjalnie niebezpieczna może być definicja zmiennych lokalnych zajmujących duże obszary pamięci (np. duża tablica w funkcji).

Rozmiar stosu można kontrolować w ustawieniach kompilatora lub konsolidatora.

W C++ można definiować zmienne w obrębie zasięgu instrukcji.

```
for(int i = 0; i < 10; i++)
  printf_s("i = %d\n", i);

i = 0; // blad
```

Zmienne statyczne

Zmienne **statyczne** mogą być deklarowane w bloku lub zewnętrzne dla wszystkich bloków.

Jeżeli zmienna wewnątrz bloku będzie zadeklarowana ze specyfikatorem `static` będzie **przechowywać wartość** po opuszczeniu i ponownym wejściu do bloku.

Zmienna statyczna jest inicjowana wartością inicjalizatora lub nadawana jej jest wartość zero odpowiedniego typu.

```
void limitFoo(void)
{
  static int counter = 0;

  if(counter < 10)

    counter++;

  else

    return;
}
```

Zmienne rejestrowe

Deklaracja zmiennej jako `register` jest równoważna z deklaracją auto, ale wskazuje że deklarowany obiekt będzie intensywnie wykorzystywany, i w miarę możliwości będzie umieszczony w rejestrze procesora.

Jeżeli nie jest możliwe umieszczenie zmiennej w rejestrze, pozostaje ona w pamięci.

Zmienne rejestrowe pozwalają poprawić szybkość wykonania operacji. Jednak większość współczesnych kompilatorów wykorzystuje optymalizację rejestrową, zatem wiele zmiennych i tak przechowywanych jest w rejestrach.

```
int fastFoo(register int i)
{
  ...
}
```

Zmienne extern

To zmienne zewnętrzne deklarowane na zewnątrz wszystkich funkcji. Zasięg zmiennej zewnętrznej rozciąga się **od miejsca deklaracji do końca pliku**.

Zmienne zewnętrzne istnieją stale, nie pojawiają się i nie znikają, zachowują swoje wartości i są dostępne dla wszystkich funkcji programu występujących w zakresie danej zmiennej.

Zmienna zewnętrzna jest raz inicjowana wartością inicjalizatora lub nadawana jest jej wartość zerowa odpowiednia dla typu.

Jeżeli dla zmiennej zewnętrznej użyjemy specyfikacji `static`, to oznacza to *uprywatnienie* (ograniczenie dostępu) w obrębie danego pliku źródłowego.

Zmienne zewnętrzne oraz ich właściwe definiowanie i deklarowanie mają istotne znaczenie przy organizacji programów wielomodułowych.

```
int foo;

void getFoo()
{
  scanf_s("%d", &foo);
}
void printFoo()
{
  printf_s("foo = %d", foo);
}

int main()
{
  getFoo();
  printFoo();
  ...
}
```

Zmienne o nieprzewidywalnie zmiennej wartości

Specyfikator `volatile` oznacza obiekt o wartościach zmieniających się w nieprzewidywalny sposób, inaczej obiekty ulotne.

Obiektami takimi mogą być zmienne odnoszące się do obiektów zewnętrznych w stosunku do programu – zmiennych nałożonych na porty we/wy, odnoszących się do obszarów BIOS czy systemu operacyjnego.

Takie zmienne nie powinny być poddawane optymalizacjom – szczególnie optymalizacji rejestrowej. Specyfikacja ta zapobiega wszystkim potencjalnym optymalizacjom.

```
volatile unsigned char kbdState = ... ; // zmienna związana z BIOS
while(kbdState & LR_SHIFT)
{
  if(kbdState & L_SHIFT)
  ...
}
```

Zmienna `kbdState` jest często wykorzystywana, kompilator może przenieść ją do rejestru procesora. Sprawi to, że ten fragment programu nie będzie działał poprawnie.

Funkcje w module

Funkcje mogą być grupowane w **moduły**. Języki C/C++ nie oferują syntaktycznie zdefiniowanej modularyzacji.

Program może się składać z kompilowanych oddzielnie części – zwanych często modułami – łączonych potem przez konsolidator w wykonywalny plik.

Przyjęta konwencja budowania modułów zakłada oddzielne części: **publiczne** (nagłówki) i **implementacyjne**.

Część publiczna zawiera opis elementów dostępnych do użytki w innych programach. W C/C++ to tzw. **plik nagłówkowy** (`*.h`, `*.hpp`, `*.hxx`). Plik nagłówkowy jest plikiem tekstowym.

Część implementacyjna modułu zawiera wszystko co jest potrzebne do działania elementów udostępnianych przez moduł. W C/C++ to plik kodu źródłowego (`*.c`, `*.cpp`, `*.cxx`).

Część implementacyjna może być udostępniana także w wersji skompilowanej (`*.o`, `*.obj`, `*.lib`).

Część publiczna modułu figfun.h:

```
double squareArea(double side);
double squarePerimeter(double side);
double circleArea(double radius);
double circlePerimeter(double radius);
```

Część implementacyjna modułu figfun.c:

```
double squareArea(double side)
{
  return side * side;
}
double squarePerimeter(double side)
{
  return 4 * side;
}
double circleArea(double radius)
{
  return 3.14 * radius * radius;
}
double circlePerimeter(double radius)
{
  return 2 * 3.14 * radius;
}
```

Moduły zwykle włączają własny plik nagłówkowy. W tym konkretnym przypadku nie jest to konieczna.

Moduły mogą też włączać inne pliki nagłówkowe (`math.h` i stała `M_PI`).

```
#include "foofun.h"
#include <math.h>

double squareArea(double side)
{
  return side * side;
}
double squarePerimeter(double side)
{
  return 4 * side;
}
double circleArea(double radius)
{
  return M_PI * radius * radius;
}
double circlePerimeter(double radius)
{
  return 2 * M_PI * radius;
}
```

Elementy eksportowane przez moduł

Stałe w formie symboli preprocesora:

```
#define KEY_UP 0x48
#define KEY_DOWN 0x50
#define KEY_LEFT 0x4b
#define KEY_RIGHT 0x4d
```

Typy wyliczeniowe:

```
enum keyCodes
{
  KEY_UP = 0x48,
  KEY_DOWN = 0x50,
  KEY_LEFT = 0x4b,
  KEY_RIGHT = 0x4d
};
```

Nazwy typów:

```
typedef unsigned char byte;
typedef unsigned short int word;
typedef unsigned long int counter;
```

Definicje stałych:

```
const int maxSpeed = 50;
const double myPI = 5.0;
```

Prototypy funkcji:

```
double squareArea(double side);
double squarePerimeter(double side);
double circleArea(double radius);
double circlePerimeter(double radius);
```

Zmienne. Plik nagłówkowy (*.h) zawiera deklarację zmiennej:

```
extern int errorFoo;
```

Implementacja (*.c) zawiera definicję zmiennej:

```
int errorFoo = 0;
```

Każda funkcja i zmienna zewnętrzna mogą mieć ograniczony dostęp w obrębie modułu.

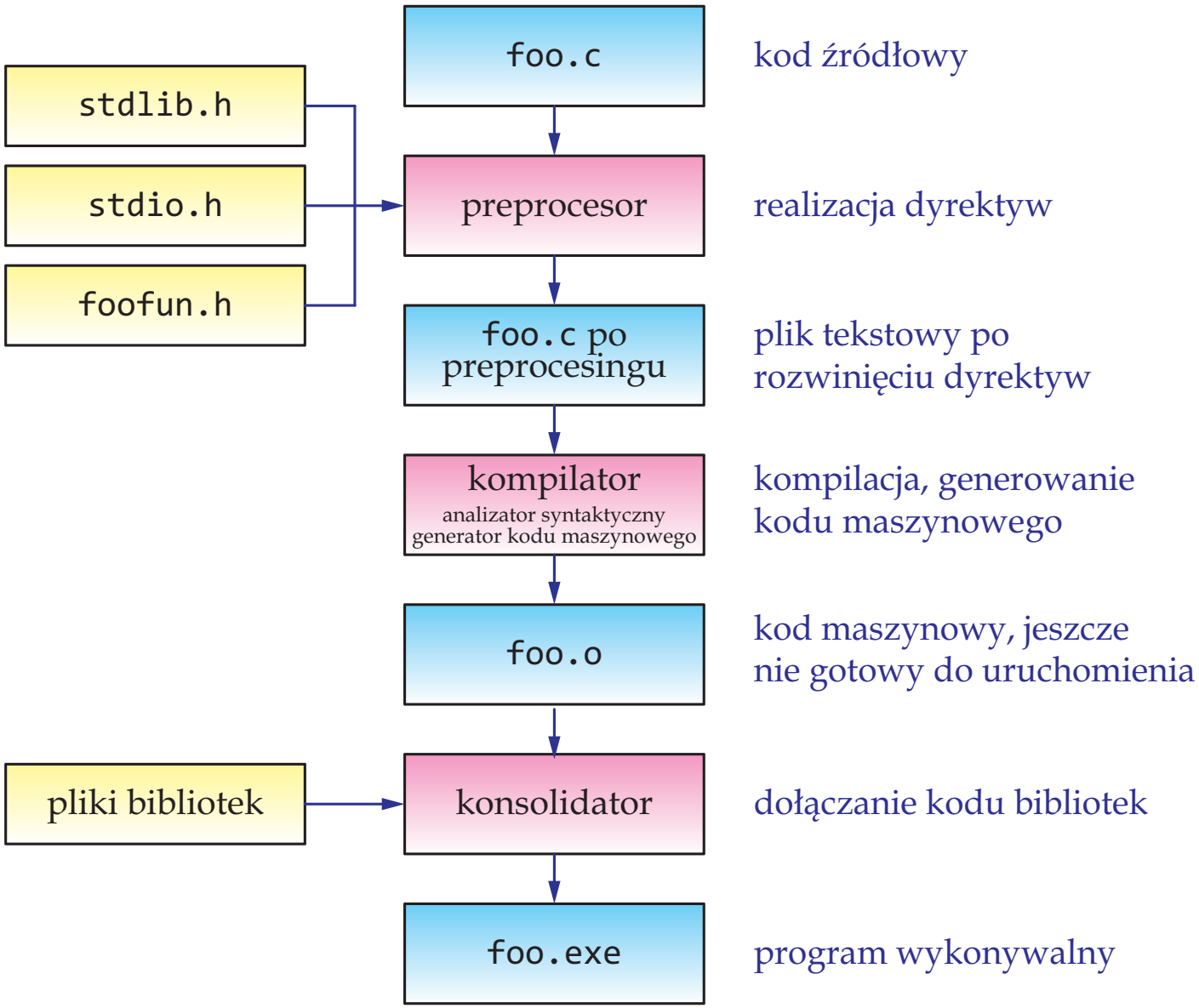
```
static int privateFoo(void)
{
   ...
}
```

Kompilacja warunkowa

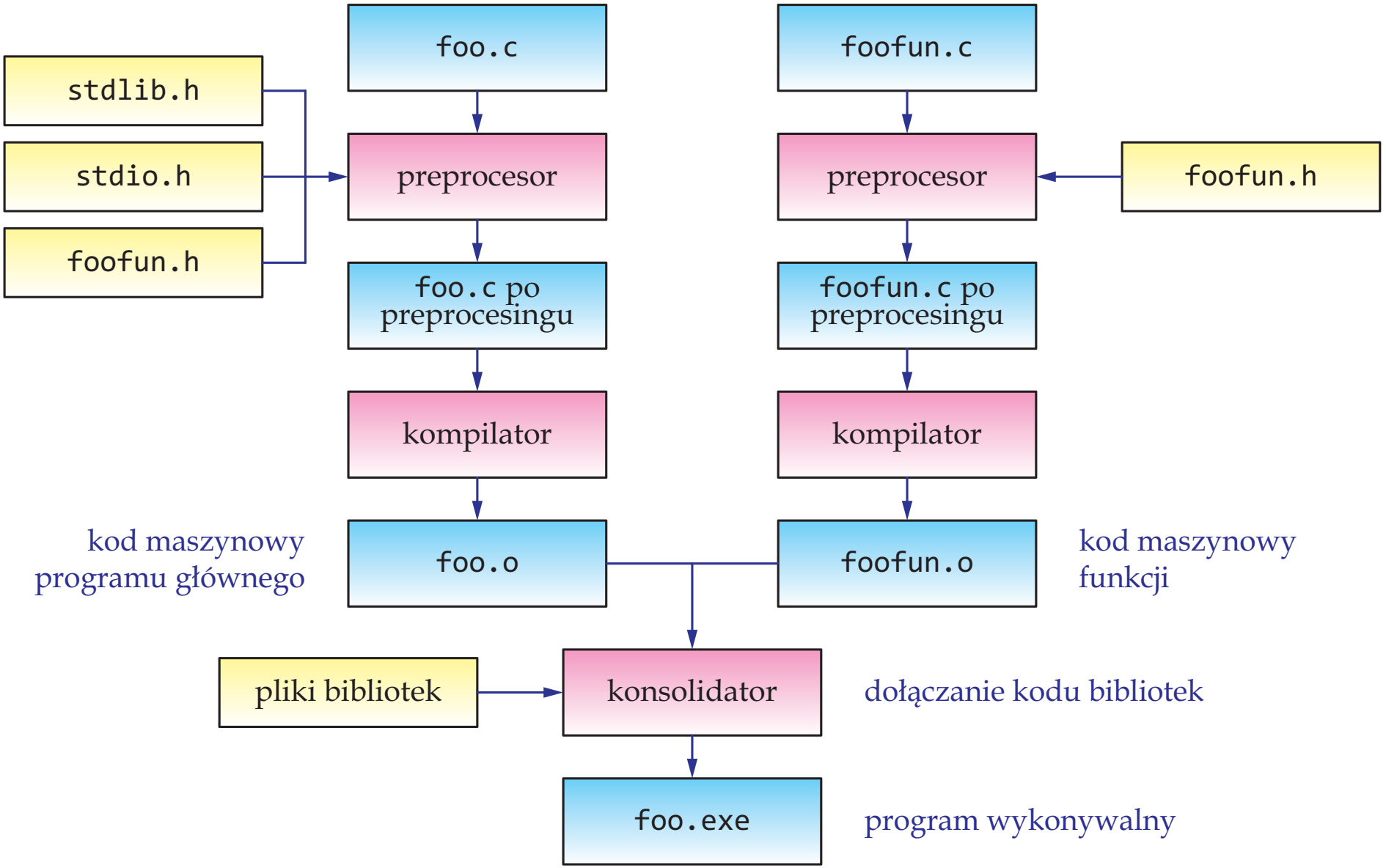
Jeżeli moduł eksportuje definicje **typów** czy **stałych**, zwykle trzeba zabezpieczać plik nagłówkowy przed **wielokrotnym włączaniem** w tym samym zakresie.

```
#ifndef _figfun_h_
#define _figfun_h_

const double myPI = 5.0;

double squareArea(double side);
double squarePerimeter(double side);
double circleArea(double radius);
double circlePerimeter(double radius);

#endif
```

Kompilacja i konsolidacja



Kompilacja i konsolidacja programu *jednomodułowego*



Kompilacja rozłączna

Zarządzanie kompilacją

Zakładamy, że używamy kompilatora **GCC** (*Gnu Compiler Collection*). Polecenie

```
gcc foo.c
```

- spowoduje kompilację programu `foo.c`,
- wygenerowanie pliku pośredniego `foo.o`,
- połączenie `foo.o` z plikami bibliotek i wygenerowanie pliku wynikowego `a.exe`.

Użycie flagi `-o` pozwoli na określenie nazwy pliku wynikowego:

```
gcc -o foo.exe foo.c
```

Kompilacja rozłączna dwóch modułów (`foo.c` i `foofun.c`):

```
gcc -c foo.c
gcc -c foofun.c
```

W wyniku dostaniemy dwa pliki: `foo.o` i `foofun.o`. Można je połączyć w wykonywalny program poleceniem:

```
gcc -o foo.exe foo.o foofun.o
```

Automatyzacja procesu kompilacji

Programem automatyzującym proces kompilacji programów, na które składa się wiele zależnych od siebie plików jest na przyklad **make**.

Program przetwarza plik reguł `Makefile` i na tej podstawie stwierdza, które pliki źródłowe wymagają kompilacji. Dzięki temu nie ma potrzeby kompilacji całego projektu.

Istnieje kilka implementacji jak np.: BSD make, **GNU make**, Microsoft make.

Przykładowy Makefile:

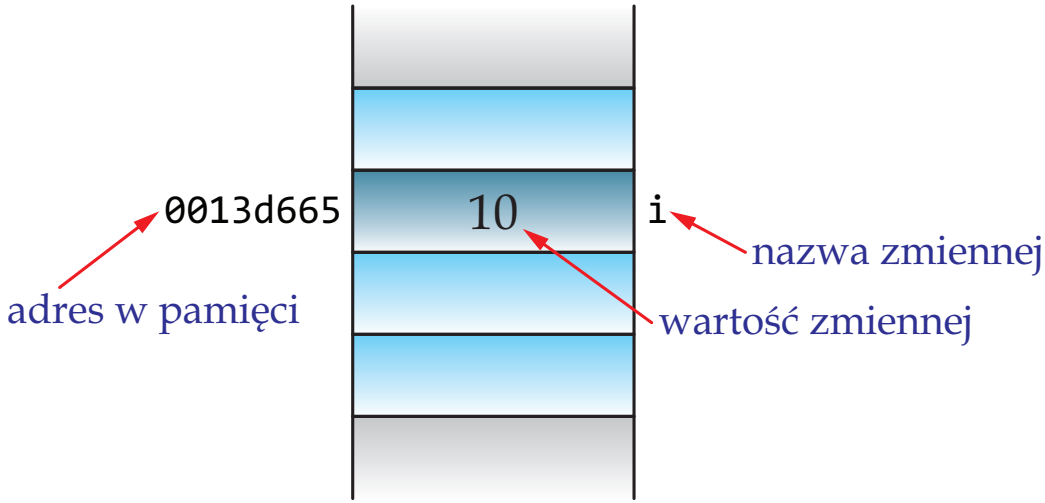
```
all: foo.exe

foo.exe: foofun.o foo.o
    gcc -o foo.exe foo.o foofun.o

foofun.o: foofun.cpp foofun.h
    gcc -c foofun.c

foo.o: foo.c foo.h
    gcc -c foo.c

clean:
    rm -f *.o foo.exe
```

Zmienna

Zmienna – (w skrócie) jest obiektem rezydującym w pamięci, przeznaczonym do przechowywania wartości danego typu. Ma **wartość** korespondującą z **typem** oraz **nazwę**. Rozmiar zajmowanej pamięci jest zależny od jej typu.

```
int i;
i = 10;
```

Nazwa identyfikuje zmienną w programie, zwalnia programistę od zastanawiania się, pod jakim adresem zmienna jest zlokalizowana.



Zmienna wskaźnikowe

Zmienna wskaźnikowa – jest przeznaczona do **lokalizowania** (wskazywania) obiektu w pamięci.

Jej jedyną rolą jest umożliwienie odwołania się do wskazywanego obiektu. Może między innymi wskazywać: **inne zmienne**, **bloki pamięci**, **funkcje**.

Zmienna wskaźnikowa może być również **wskazywana** przez **inną zmienną** wskaźnikową.

Zmienna wskaźnikowa może wskazywać: **konkretny obiekt** w pamięci, nie wskazywać **żadnego obiektu** (wiąże się to z zastosowaniem specjalnych wartości zmiennej) lub wskazywać na *nie wiadomo co* – co jest niepożądane.

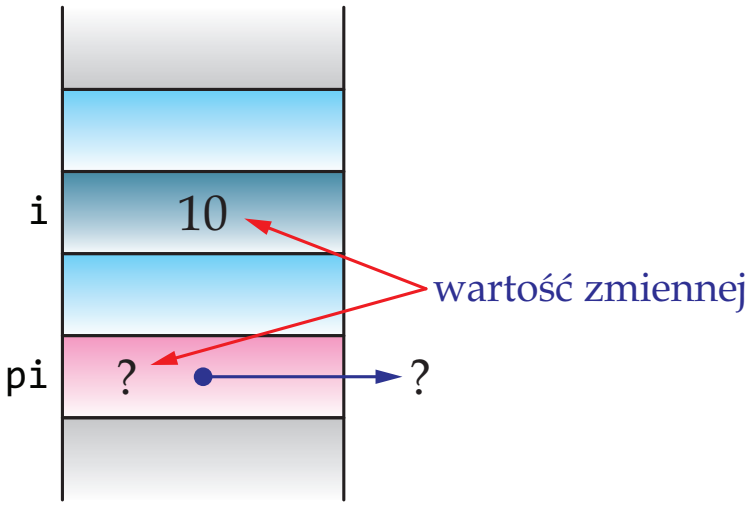
Deklaracja i definicja zmiennej wskaźnikowej

Deklarowana zmienna będzie wskaźnikiem, kompilator będzie znał jej rozmiar.

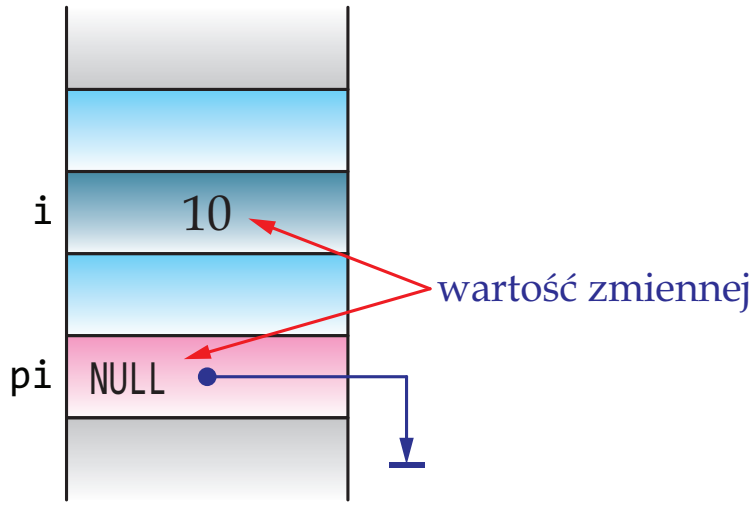
```
int *pi;
```

Deklarowana zmienna wskaźnikowa będzie przeznaczona do lokalizowania obiektów typu `int`.

```
int i = 10
int *pi;
```




```
int i = 10
int *pi = NULL;
```



Tak zdefiniowana zmienna wskaźnikowa:

```
int *pi;
```

Ma wartość początkową zależną od kontekstu deklaracji. Jeżeli zmienna jest klasy `auto`, to jej wartość jest przypadkowa.

W pliku nagłówkowym `stddef.h` jest definiowana stała `NULL`, reprezentująca wskaźnik pusty, niezależny od platformy i implementacji.

Tak zdefiniowana zmienna:

```
int *pi = NULL;
```

jest wskaźnikiem pustym, tak więc nie wskazuje żadnego obiektu w pamięci.

Stała NULL jest definiowana jako wartość 0 lub 0L. Można zatem posługiwać się wartością 0 zamiast NULL.

W języku C praktykuje się stosowanie NULL.

W języku C++ praktykuje się stosowanie wartości 0.

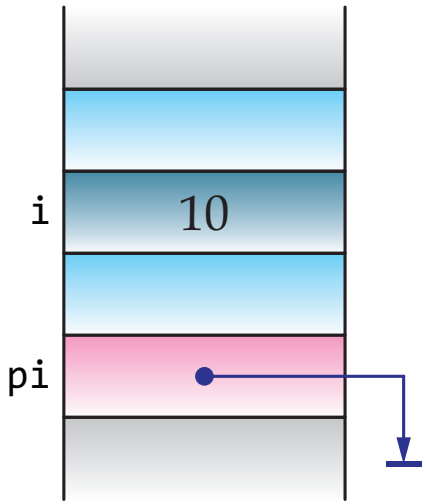
Dobrą praktyką jest jawne inicjowanie zmiennych wskaźnikowych oraz ustawianie na wartość pustą dla wskaźników niezakotwiczonych. Można wtedy sprawdzić, czy zmienna nie jest pusta:

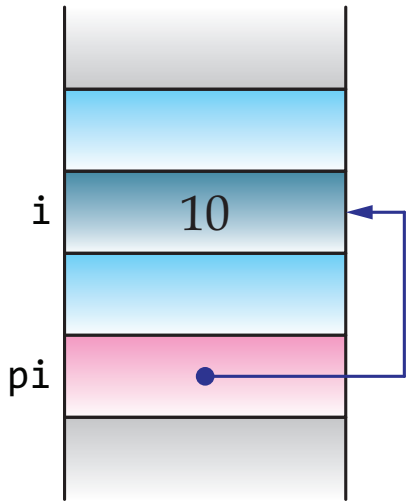
```
if(pi != NULL}
{
   ...
}
```

Przypisywanie wartości i odwołanie

```
int i = 10;
int *pi = NULL;

pi = &i;
```

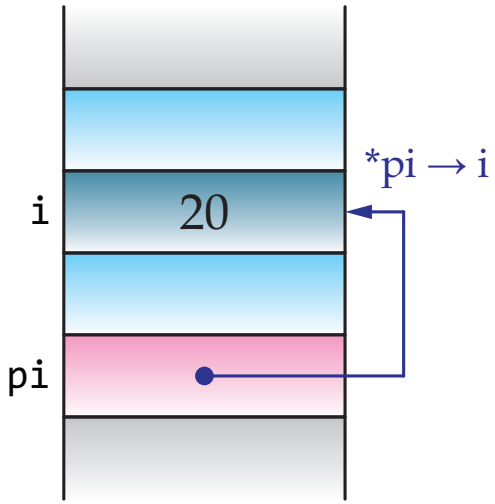
Wyrażenie `&i` **pobiera adres** (lokalizuje w pamięci) zmiennej `i`.





```
int i = 10;
int *pi = NULL;

pi = &i;
*pi = 20;
```

Wyrażenie `*pi` oznacza obiekt wskazywany przez zamienną `pi`. Wyrażenie to może wystąpić wszędzie tam, gdzie może wystąpić `i`.



Wskaźnik typu void *

Typ `void *` oznacza wskaźnik nie związany z żadnym typem.

Tego typu wskaźnik może wskazywać daną dowolnego typu.

```
int i = 5;
char c = 'A';
float f = 3.14;

void *ptr;

ptr = &i;

ptr = &c;

ptr = &f;
```

Aby odwołać się do obiektu wskazywanego przez `void *`, należy poinformować kompilator jaki jest jego typ, czyli rzutujemy typ wskaźnika:

```
int i = 5;
char c = 'A';
float f = 3.14;

void *ptr;

ptr = &i;
printf_s("i = %d\n", *(int *)ptr);

ptr = &c;
printf_s("c = %c\n", *(char *)ptr);

ptr = &f;
printf_s("f = %f\n", *(float *)ptr);
```

Zmienne referencyjne

Referencja jest typem danych, który przechowuje adres zmiennej i której nie można zmienić – do referencji można przypisać adres **tylko raz**.

Używanie zmiennej referencyjnej niczym się nie różni od używania **zwykłej zmiennej**. Operacje jakie wykona się na zmiennej referencyjnej, zostaną odzwierciedlone na zmiennej zwykłej, z której pobrano adres.

```
int i = 10;
int &ri = i;
```

Referencja musi być przypisana do zmiennej – **musi być zainicjowana**. Nie można przypisać jej wartości NULL.

Referencja musi być zainicjowana takim samym typem jaki jest typ referencji.

Przypisywanie wartości i odwołanie

```
int i = 10;
int *pi = &i;

cout << "i = " << i << endl;
cout << "!*pi = " << *pi << endl;

++(*pi);

cout << "!*pi = " << *pi << endl;
```

```
int i = 10;
int &ri = i;

cout << "i = " << i << endl;
cout << "ri = " << ri << endl;

++ri;

cout << "ri = " << ri << endl;
```

```
int i = 10;
int *pi = &i;
int &ri = i;

cout << "i = " << i << endl; // i = 10
cout << "&i = " << &i << endl; // &i = 0016FC28

cout << "pi = " << pi << endl; // pi = 0016FC28
cout << "*pi = " << *pi << endl; // *pi = 10

cout << "ri = " << ri << endl; // ri = 10
cout << "&ri = " << &ri << endl; // &ri = 0016FC28
```

Dynamiczny przydział pamięci

Z przedmiotu o finezyjnej nazwie i skrócie **WSK**, pamiętamy...

Sterta (*heap*) to wydzielony obszar **wolnej pamięci**:

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

Stos (*stack*) to wydzielony obszar **pamięci roboczej**:

- przeznaczony do przechowywania danych automatycznych,
- nie jest bezpośrednio kontrolowany przez programistę,
- ograniczony pod względem rozmiaru,
- przydzielany wg. zasady LIFO (*Last In - First Out*).

Dynamiczny przydział pamięci polega na zarezerwowaniu fragmentu pamięci w obszarze **sterty** dla obiektu w trakcie działania programu.

Wymaga określenia **wielkości obszaru pamięci**, przydziału i zapamiętania jego początku w **zmiennej wskaźnikowej**.

Jeśli przydzielenie pamięci powiedzie się, można z niej korzystać i koniecznie **zwolnić**, jeśli nie jest już potrzebna.

Funkcje zarządzające przydziałem i zwalnianiem bloków pamięci operują na wskaźnikach `void *`. Przydzielane bloki są amorficzne – obszary pamięci o rozmiarze liczonym w bajtach.

Wykorzystanie funkcji przydzielających i zwalniającej pamięć wymaga włączenia pliku nagłówkowego `stdlib.h` lub `cstdlib` w C++.

Funkcje realizujące przydział pamięci:

```
void * malloc(size_t size);
```

Rezultatem funkcji `malloc()` jest wskaźnik do obszaru pamięci przeznaczonego dla obiektu o rozmiarze `size`. Rezultatem jest `NULL`, jeżeli polecenie nie może być zrealizowane. Obszar nie jest inicjowany.

```
void * calloc(size_t nitems, size_t size);
```

Rezultatem funkcji `calloc()` jest wskaźnik do obszaru pamięci przeznaczonego dla `nitems` obiektów o rozmiarze `size`. Rezultatem jest `NULL`, jeżeli polecenie nie może być zrealizowane. Obszar jest inicjowany zerami.

```
void * realloc(void *ptr, size_t size);
```

Funkcja dokonuje próby zmiany rozmiaru bloku wskazywanego przez `*ptr`, który był poprzednio przydzielony wywołaniem funkcji `calloc()` lub `malloc()`.

Jeżeli nowy rozmiar jest większy od poprzednio przydzielonego, dodatkowe bajty mają nieokreśloną wartość. Jeżeli nowy rozmiar jest mniejszy, bajty z różnicowego obszaru są zwalniane.

Jeżeli `ptr` ma wartość `NULL` to funkcja działa jak `malloc()`.

Funkcja zwalniająca pamięć:

```
void free(void *ptr);
```

Zwalnia obszar pamięci wskazywany przez `ptr`. Parametr musi być wskaźnikiem do obszaru pamięci przydzielonego uprzednio przez `malloc()`, `calloc()` lub `realloc()`.

Dynamiczny przydział pamięci dla pojedynczych danych typu `int`, `char` czy `double` najczęściej nie ma sensu.

Dynamiczny przydział pamięci jest wykorzystywany dla obiektów zajmujących dużo pamięci operacyjnej oraz dla złożonych struktur danych.

W języku C++ można używać opisanych funkcji. Jednak w tym języku wprowadzono specjalne operatory zarządzające pamięcią: `new` oraz `delete`.

Ich wykorzystanie jest zalecane a w wielu przypadkach konieczne.

Operatory są częścią języka i mają wbudowane mechanizmy kontroli typów.

```c
int *ptr = NULL;

ptr = malloc(sizeof(int));

if(ptr != NULL)
{
  *ptr = 10;

  printf_s("*ptr = ", ++(*ptr));

  free(ptr);
  ptr = NULL;
}
```

```c
int *ptr = 0;

ptr = new (nothrow) int;

if(ptr != 0)
{
  *ptr = 10;

  cout << "*ptr = " << ++(*ptr);

  delete ptr;
  ptr = 0;
}
```

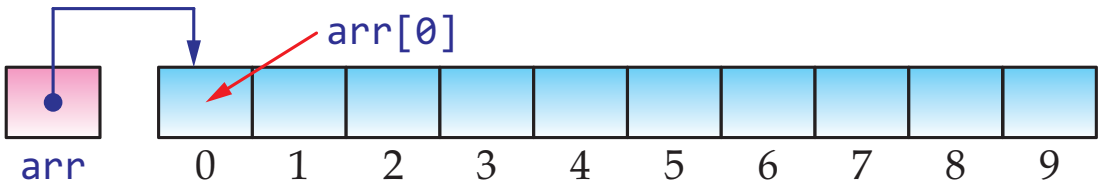
Obecnie częściej stosuje się wyjątki:

```
try
{
  int *ptr = new int;

  *ptr = 10;

  cout << "*ptr = " << ++(*ptr);

  delete ptr;
  ptr = 0;
}
catch(bad_alloc)
{
  ... // obsluga wyjatku (brak pamieci)
}
```

Tablice a wskaźniki

Nazwa tablicy to wskaźnik na jej pierwszy element.

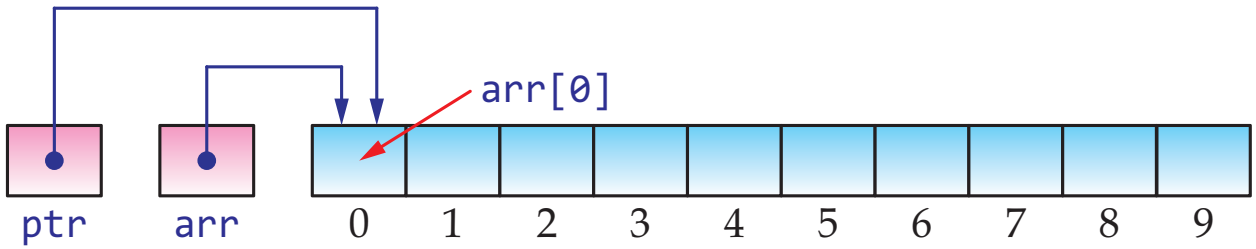
```
int arr[10];
```



Przypisania równoznaczne:

```
int arr[10];
int *ptr;

ptr = arr;
```

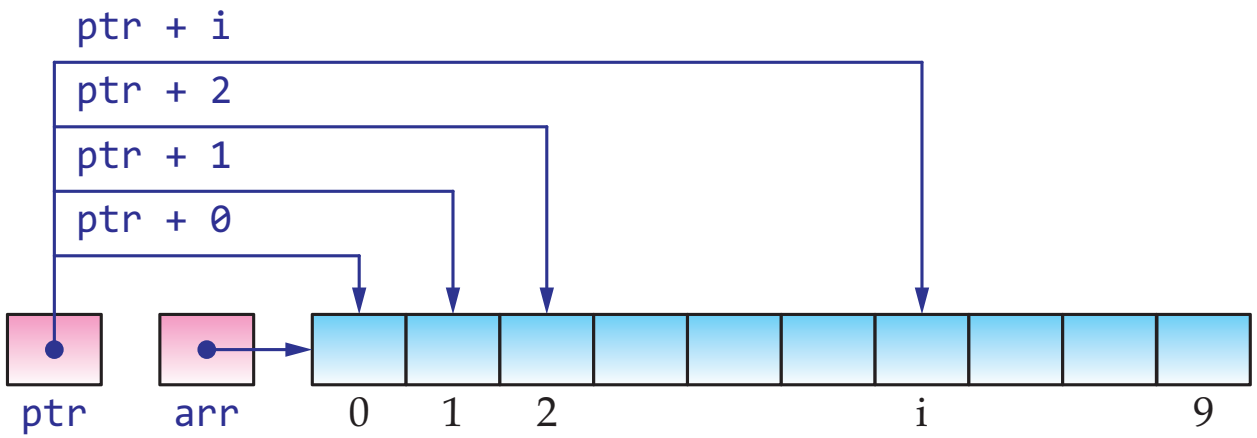
```
int arr[10];
int *ptr;

ptr = &arr[0];
```



```
arr[0] = 3;
arr[1] = 7;
arr[2] = 9;

tab[i] = 13;
```

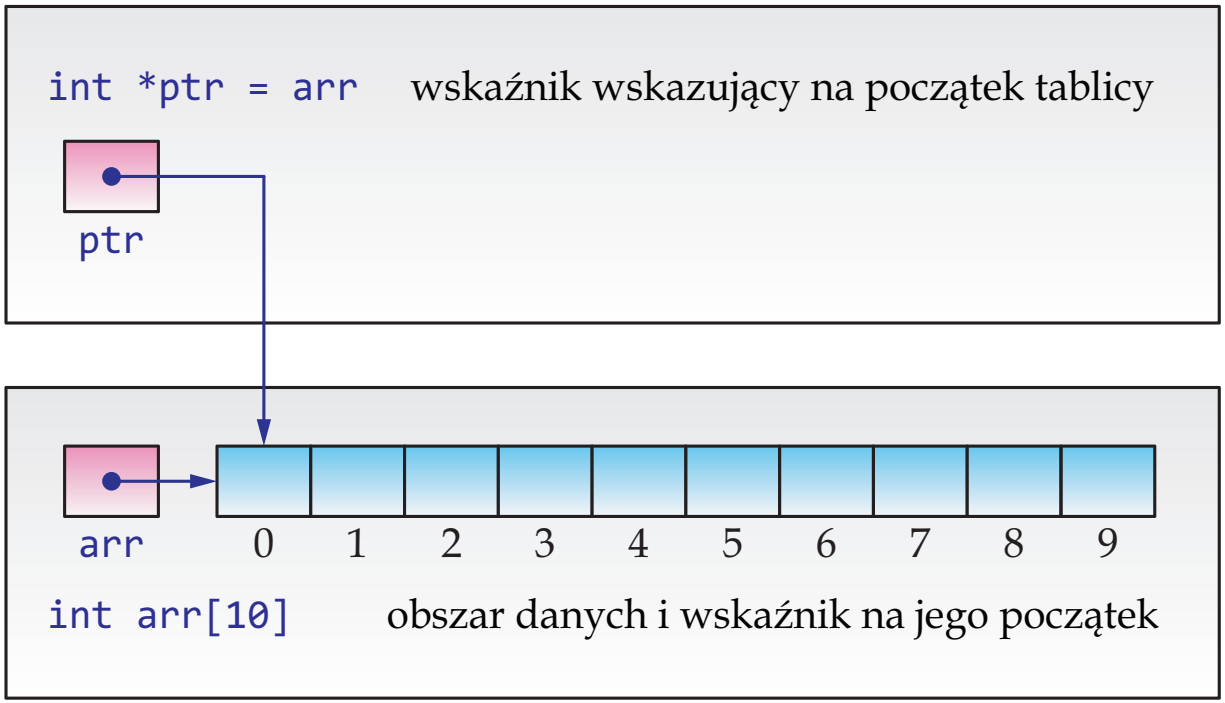
```
*arr = 3;
*(arr + 1) = 7;
*(arr + 2) = 9;

*(arr + i) = 13;
```

```
*ptr = 3;
*(ptr + 1) = 7;
*(ptr + 2) = 9;

*(ptr + i) = 13;
```



Wskaźnik to nie tablica!

```
int arr[10];
int *ptr = arr;
```



Nazwa tablicy jest niemodyfikowalnym wskaźnikiem na jej pierwszy element. Nazwy tablicy **nie można modyfikować**! Zwykłe wskaźniki można.

```
int arr[10];
int *p = arr;

p = arr + 3; // ok
p++; // ok

arr = p; // blad
arr++; // blad
```

Wskaźniki a const

Ustalony wskaźnik na niemodyfikowalny obiekt (wymaga inicjalizacji):

```
const int *const p = &i
```

Ustalony wskaźnik na modyfikowalny obiekt (wymaga inicjalizacji):

```
int *const p = &i;
```

Zwykły wskaźnik na niemodyfikowalny obiekt:

```
const int *p;
```

Przykład:

```
void strcopy(char d[], char s[])
{
  int i;
  for(i = 0; s[i] != '\0'; i++)
    d[i] = s[i];
  d[i] = '\0';
}
```

```
void strcopy(char *d, const char *s)
{
  while(*d++ = *s++);
}
```

Odwołanie:

```
tab[i]
```

jest tożsame z:

```
*(tab + i)
```

a dodawanie jest przemienne, zatem:

```
*(i + tab)
```

Zagadka – czy odwołanie:

```
*(i + tab)
```

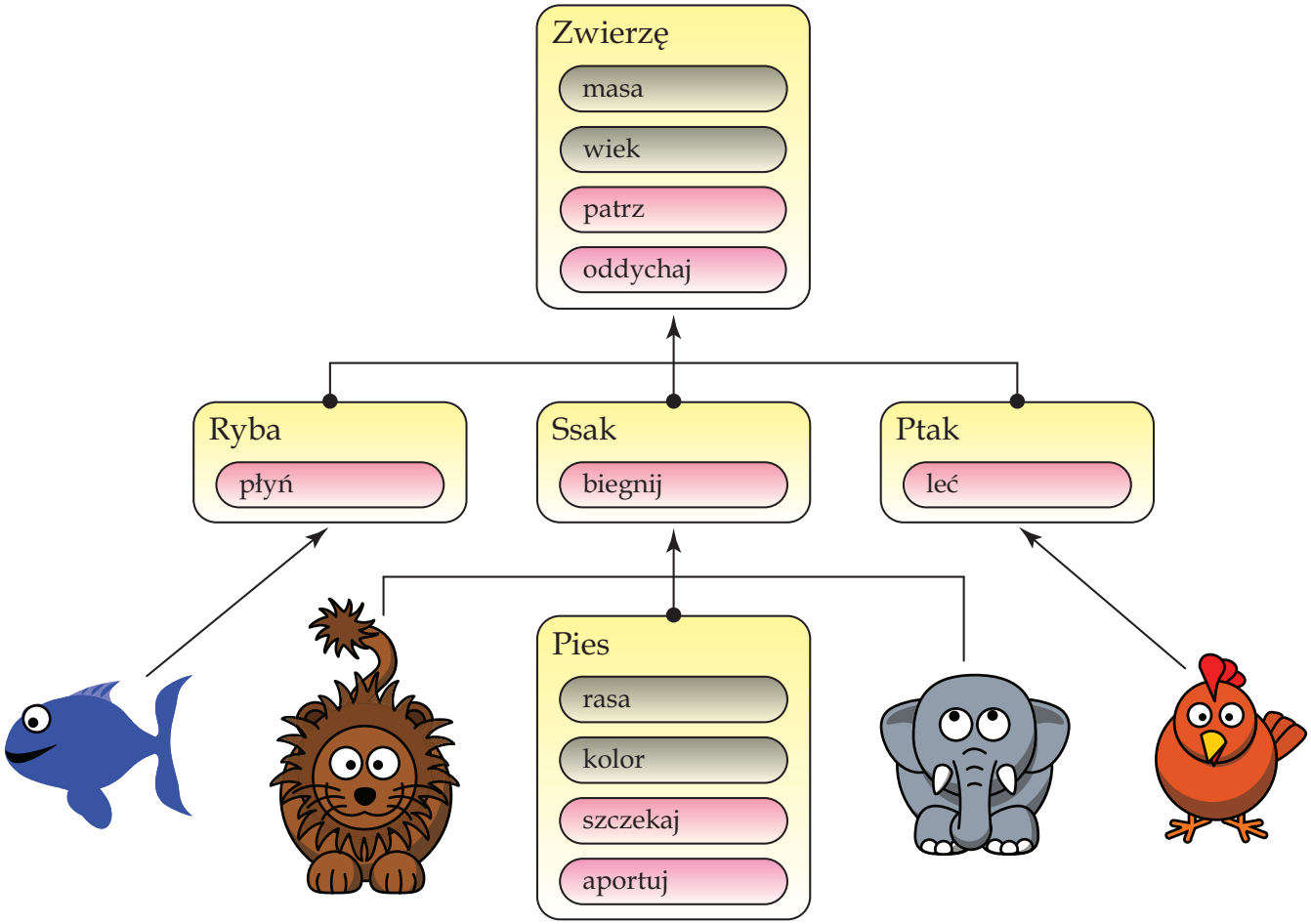
można zapisać w postaci:

```
i[tab]
```

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

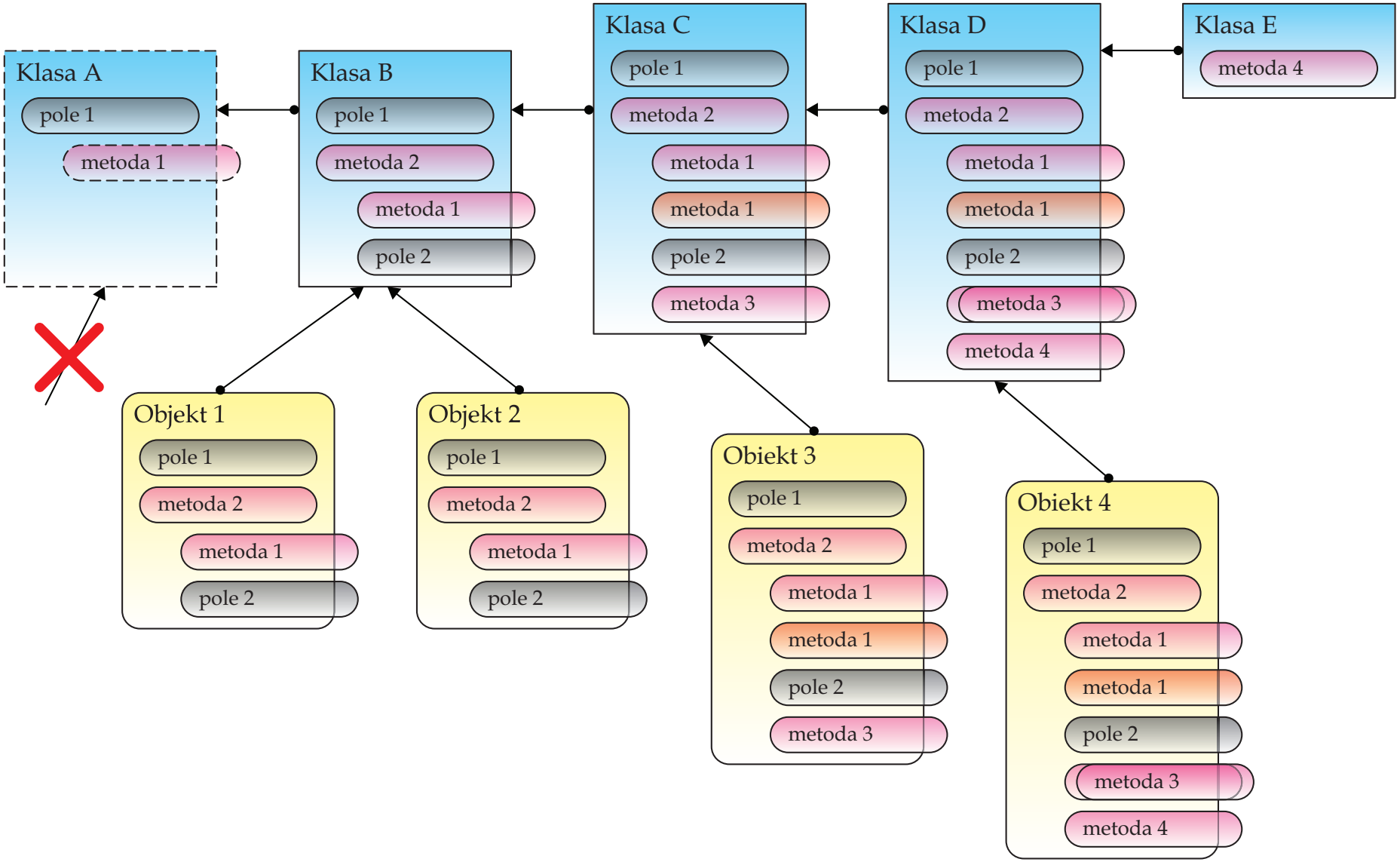
Wymyślając nowy lub analizując istniejący program obiektowy nasz mózg ma ułatwione zadanie. Dlatego ludzie są w stanie łatwiej zapanować nad kodem i tym samym tworzyć większe programy. Łatwiej jest również zrozumieć kod i pomysły innych programistów i tym samym współpracować w zespole oraz ponownie wykorzystywać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

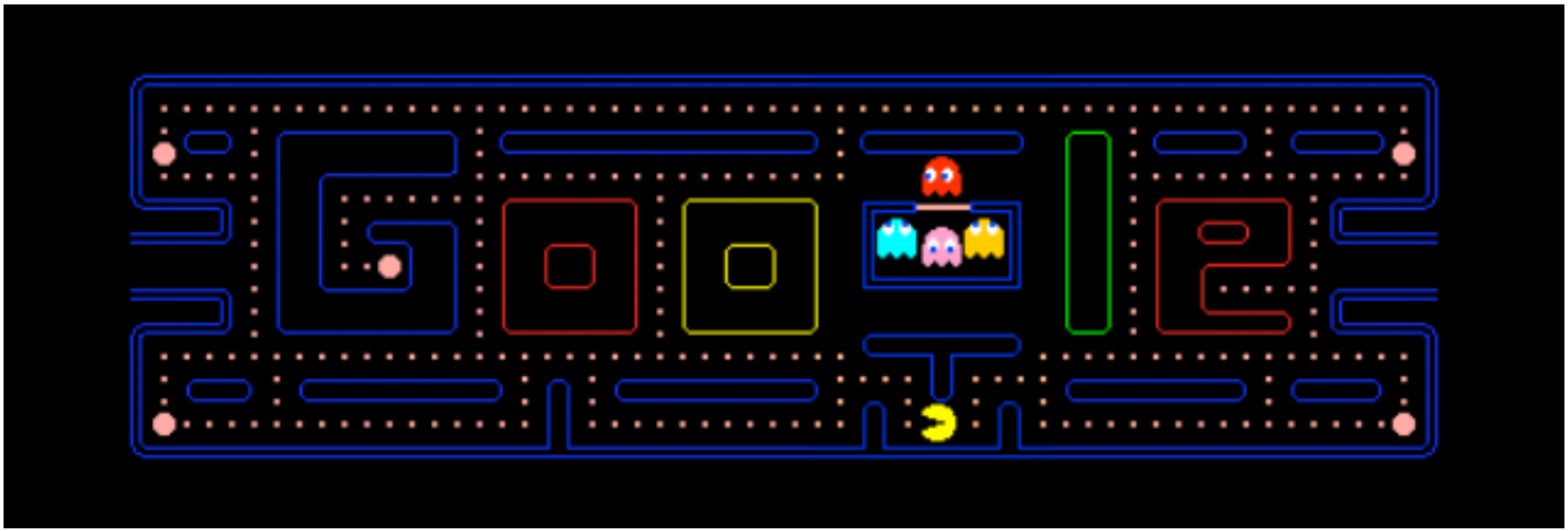

Zwierzę
masa
wiek
patrz
oddychaj
Ryba
płyń
Ssak
biegnij
Ptak
leć
Pies
rasa
kolor
szczekaj
aportuj

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
metoda 1
pole 2
Klasa C
pole 1
metoda 2
metoda 1
metoda 1
pole 2
metoda 3
Klasa D
pole 1
metoda 2
metoda 1
metoda 1
pole 2
metoda 3
metoda 4
Klasa E
metoda 4
Objekt 1
pole 1
metoda 2
metoda 1
pole 2
Objekt 2
pole 1
metoda 2
metoda 1
pole 2
Obiekt 3
pole 1
metoda 2
metoda 1
metoda 1
pole 2
metoda 3
Obiekt 4
pole 1
metoda 2
metoda 1
metoda 1
pole 2
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); // wyswietl znak na zadanej pozycji
  key = getKey(); // obsluga nacisniecia klawisza
  writeChar(x, y, ' '); // nadpisanie znaku
  switch(key)
  {
    case KEY_UP:
      if(y > 1) --y; // idz wyzej
      break;
    case KEY_DOWN:
      if(y < 24) ++y; // idz nizej
      break;
    case KEY_LEFT:
      if(x > 1) --x; // idz w lewo
      break;
    case KEY_RIGHT:
      if(x < 80) ++x; // idz 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.

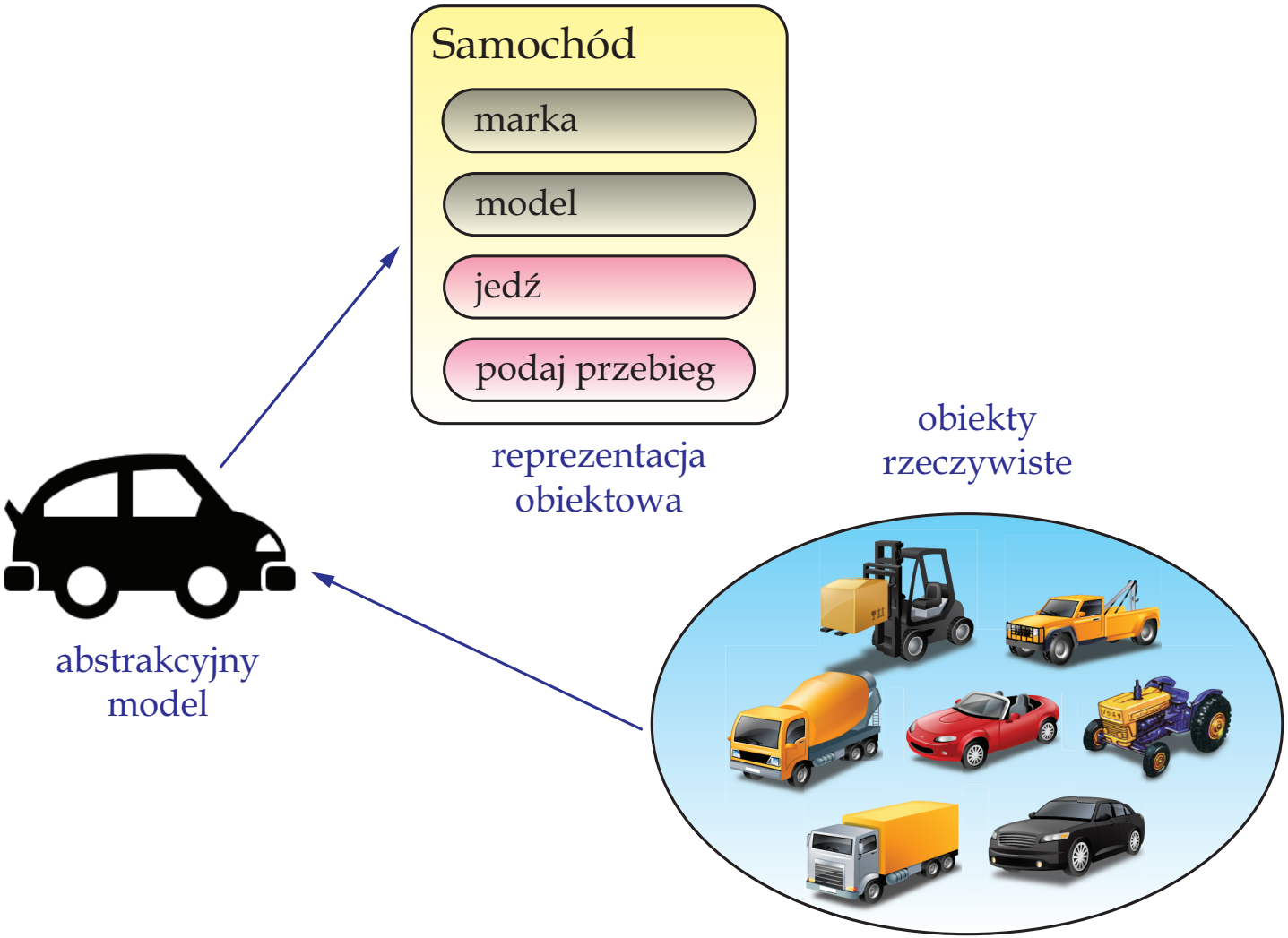
Zachowanie – zdolność do działania opisują operacje, które obiekt może wykonać. operacje są realizowane przez wykonanie **metod** (*methods*), czasem zwanych funkcjami składowymi (*member functions*).Metody mogą otrzymywać parametry i zwracać wartości, zwykle zmieniają stan obiektu. Metody są częścią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

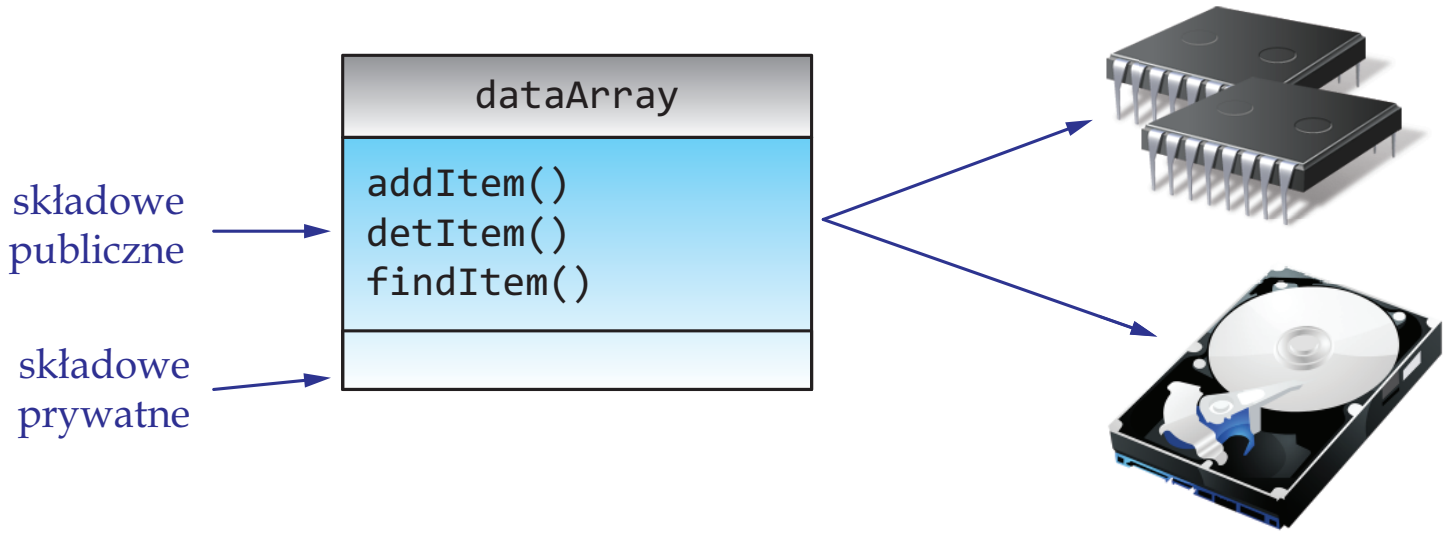
Enkapsulacja

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; // wartosc domyslna
}
```

```
void TPacman::setY(int newY)
{
  if(isYOnScreen(newY))
    y = newY;
  else
    y = 1; // wartosc domyslna
}
```

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); // blad
```

```
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)
{
}
```

Technika przeciążania funkcji

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; // blad, 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.

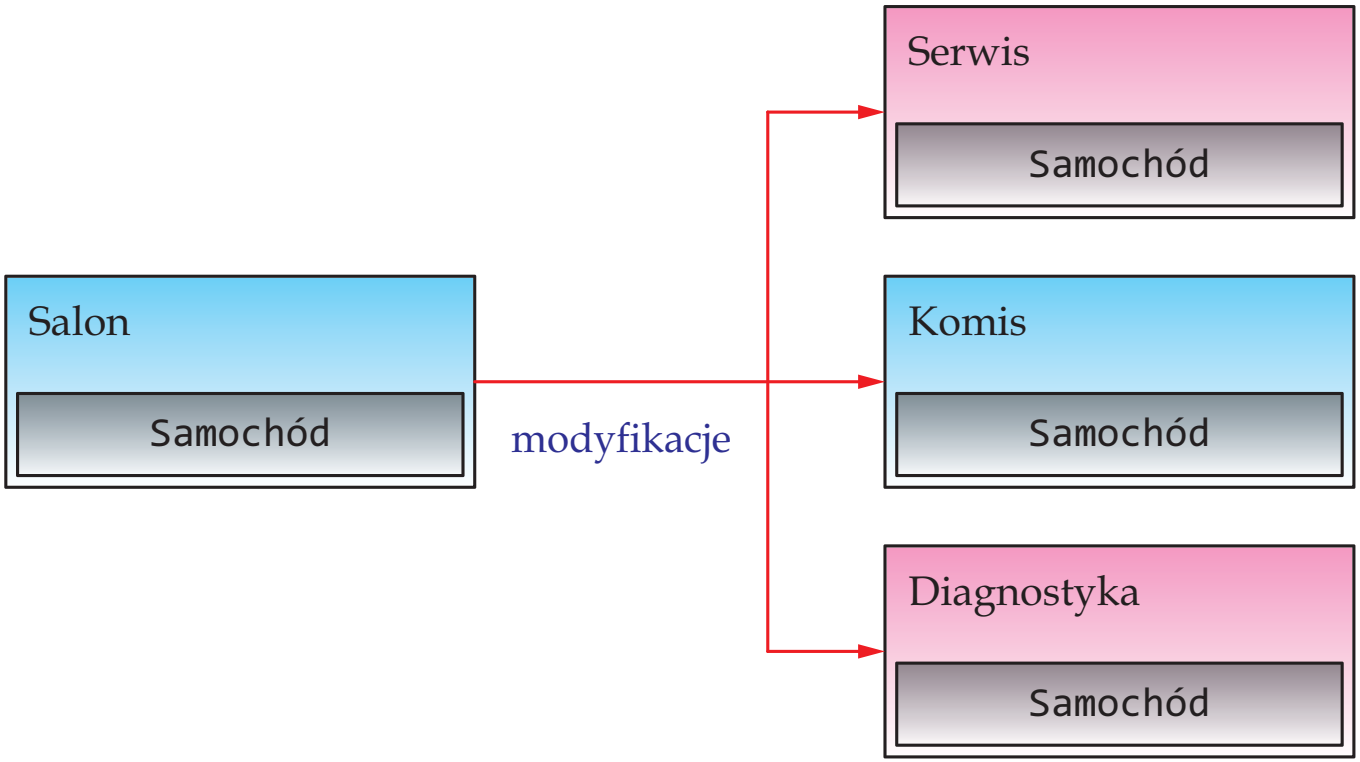
Dla obiektów specjalizowanych nie trzeba redefiniować całej funkcjonalności, lecz tylko tę, której nie ma obiekt ogólniejszy. W typowym przypadku powstają grupy obiektów zwane klasami, oraz grupy klas zwane drzewami. Odzwierciedlają one wspólne cechy obiektów.

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:

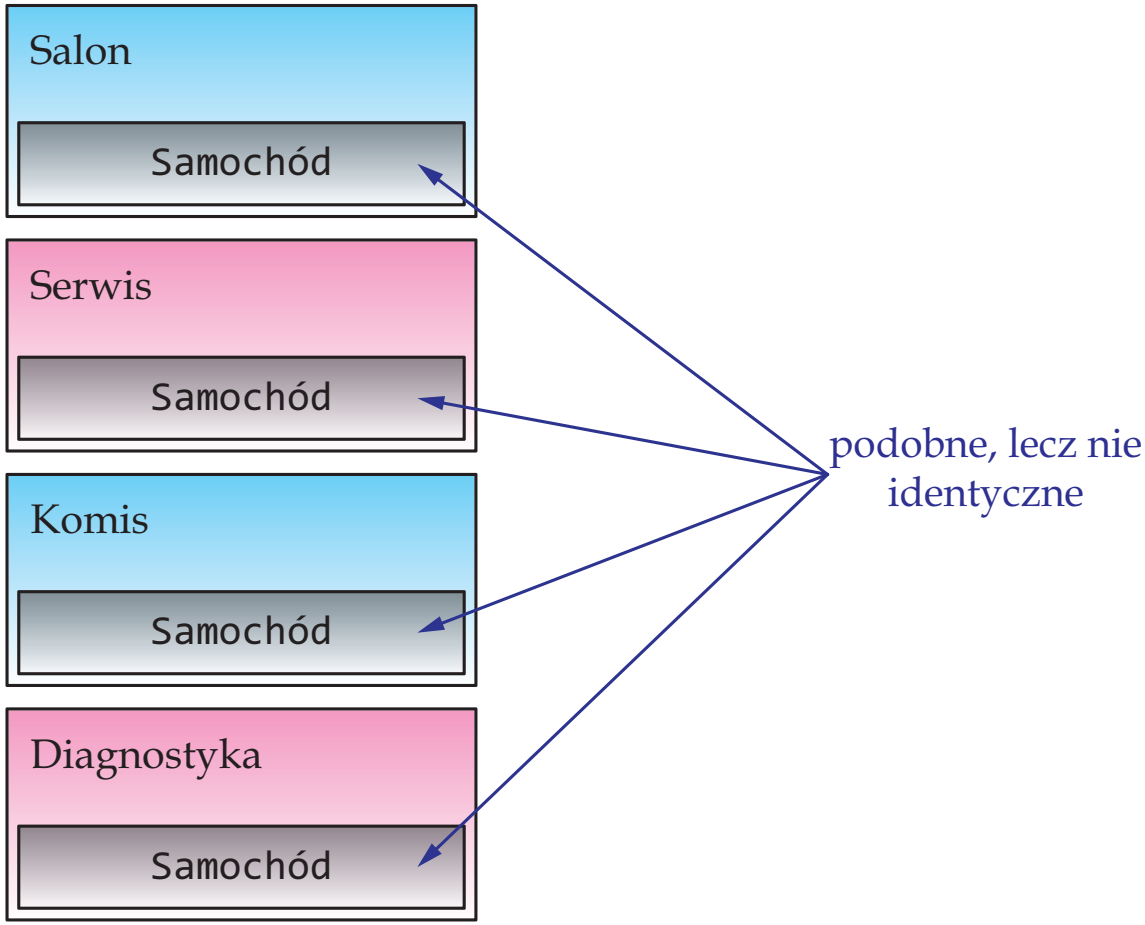
- powtarzalność nieidentycznych obiektów powoduje to, że musimy modyfikować istniejący kod, dostosowując go do specyfiki projektu,
- modyfikacje potrzebne z punktu widzenia jednego systemu mogą być nieużyteczne lub nieakceptowalne z punktu widzenia innego systemu,
- modyfikacje dostosowujące kod do specyfiki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



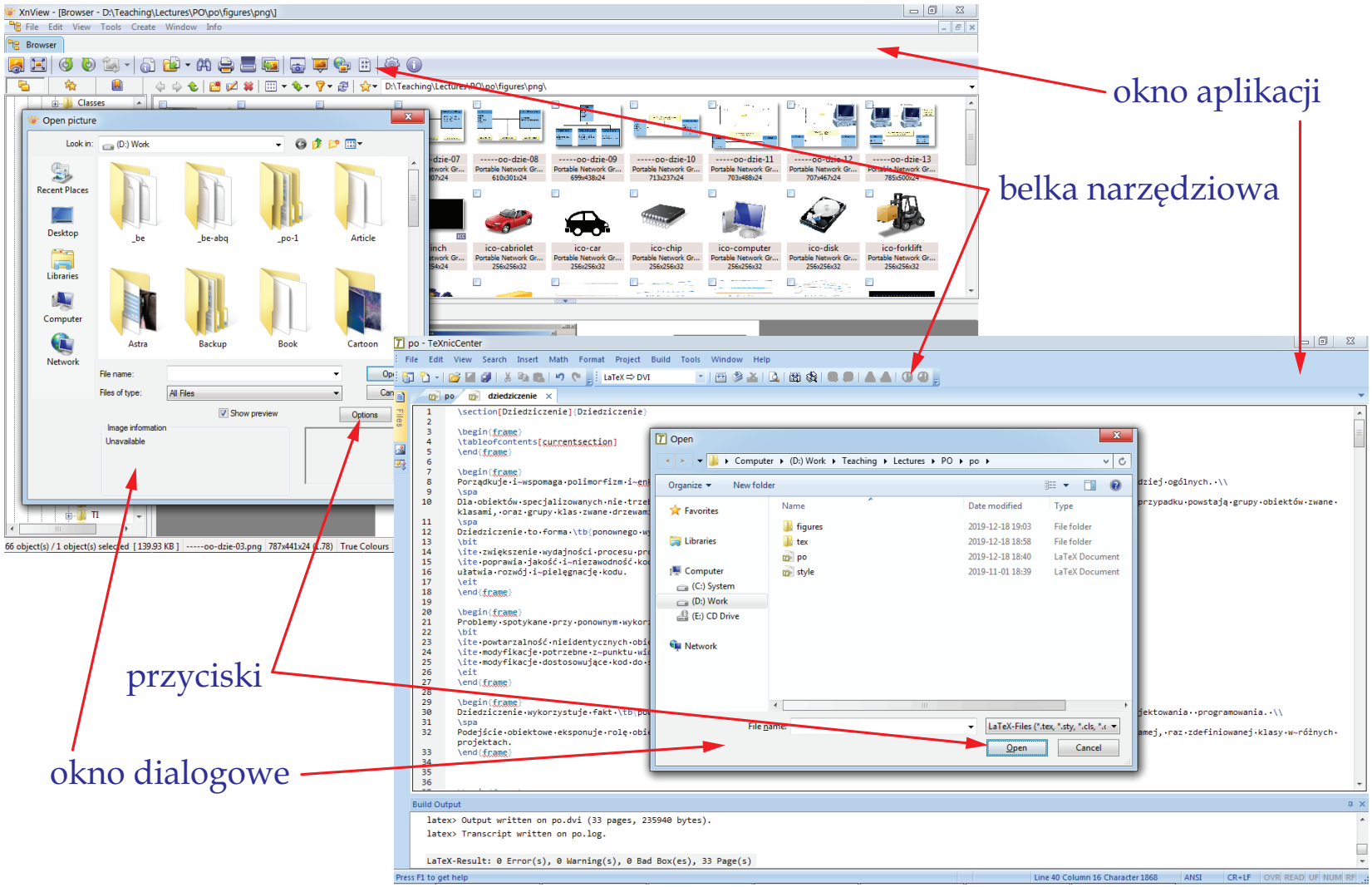
Dziedziczenie wykorzystuje fakt **powtarzalności elementów oprogramowania**. Powtarzalność taką można obserwować na etapie analizy, projektowania programowania.

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.

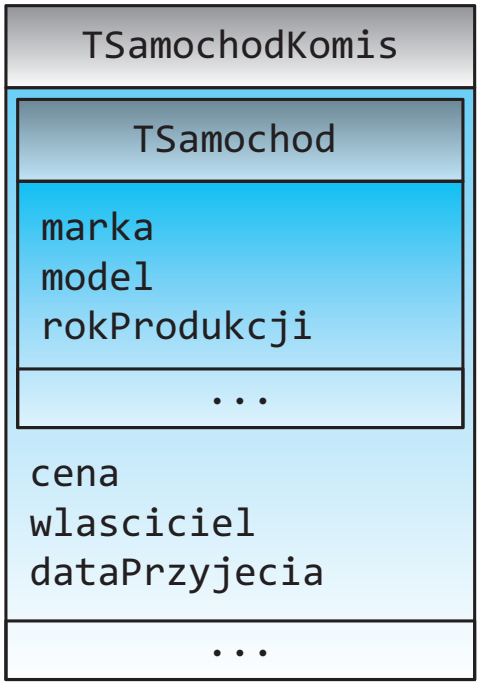


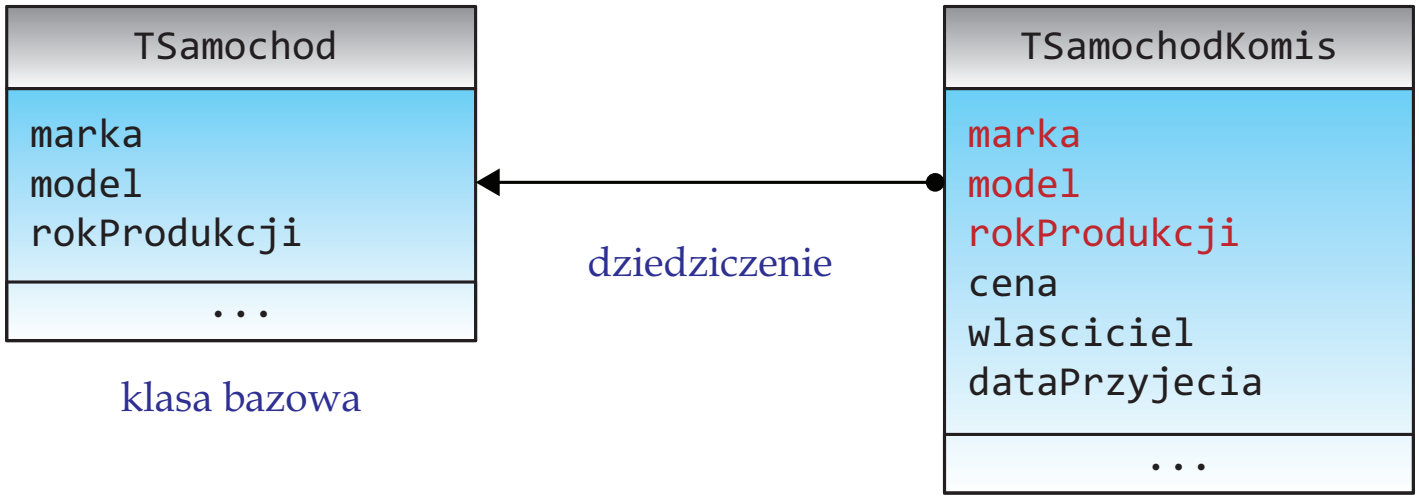
W obrębie obiektów implementacyjnych powtarzalność obiektów identycznych jest znacznie częstsza, co jednak nie jest regułą.

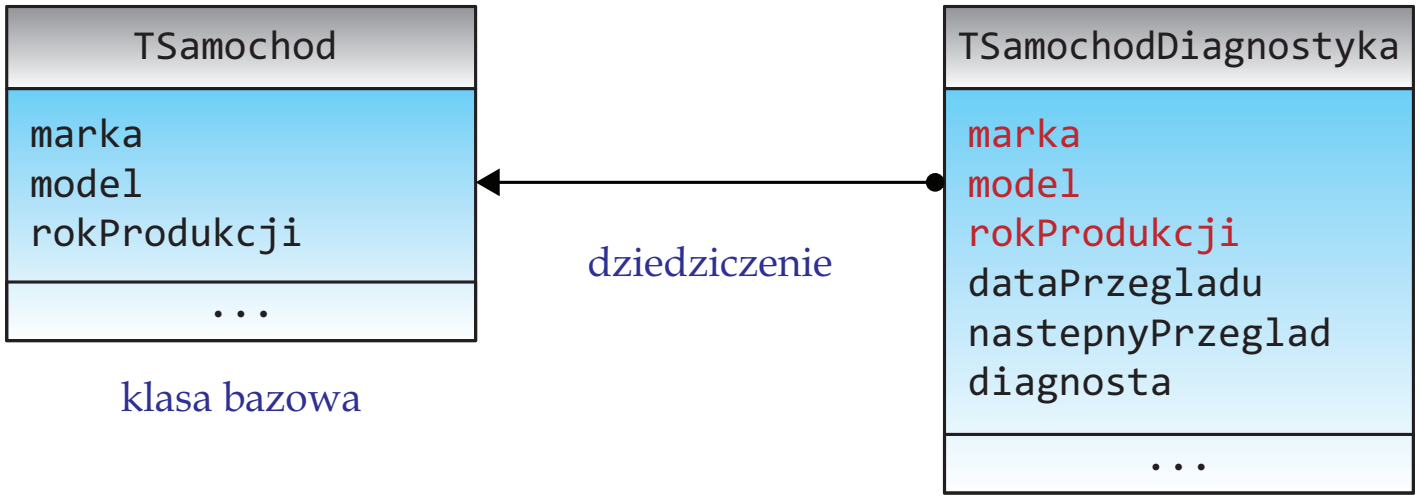


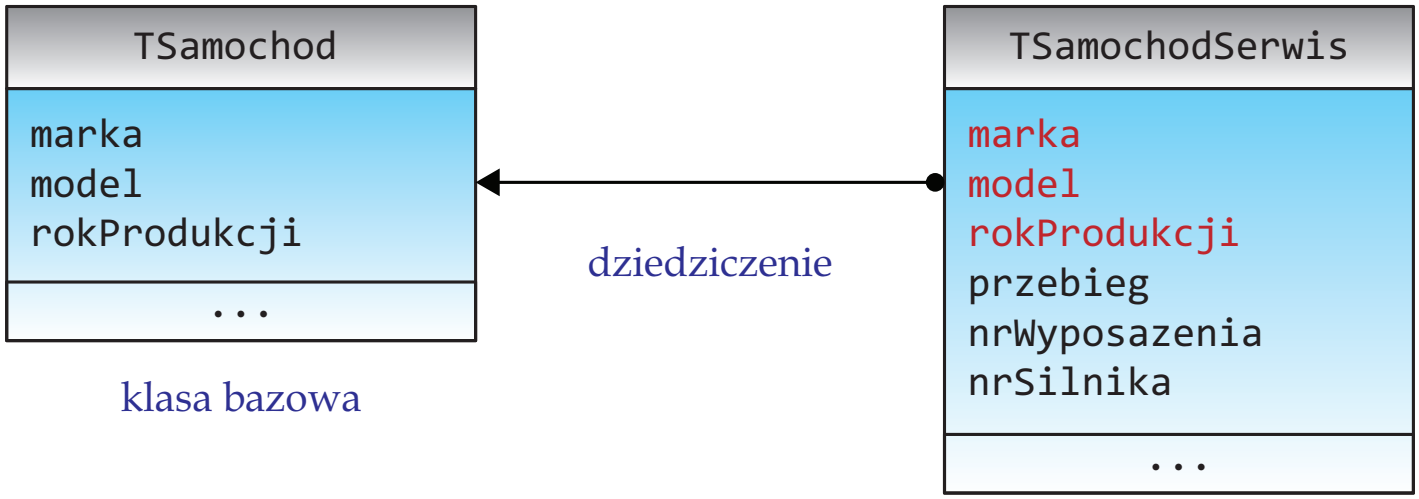
Dziedziczenie polega na **budowaniu nowych klas**, zwanych **klasami potomnymi**, na podstawie **klas już istniejących**, zwanych **klasami bazowymi**.

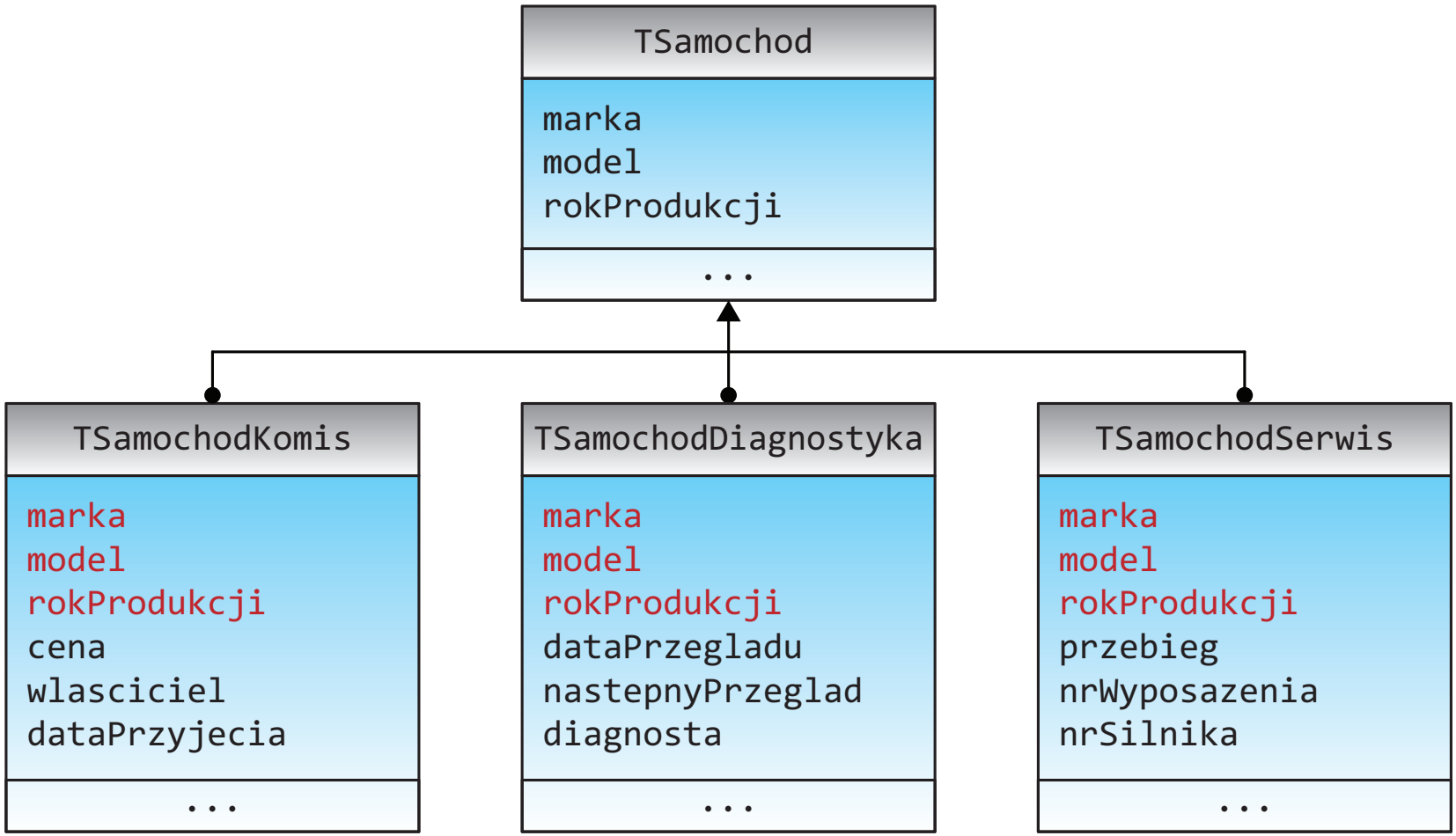
Każda klasa pochodna **dziedziczy wszystkie właściwości** klasy bazowej, **rozszerzając je o nowe** pola i metody.




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

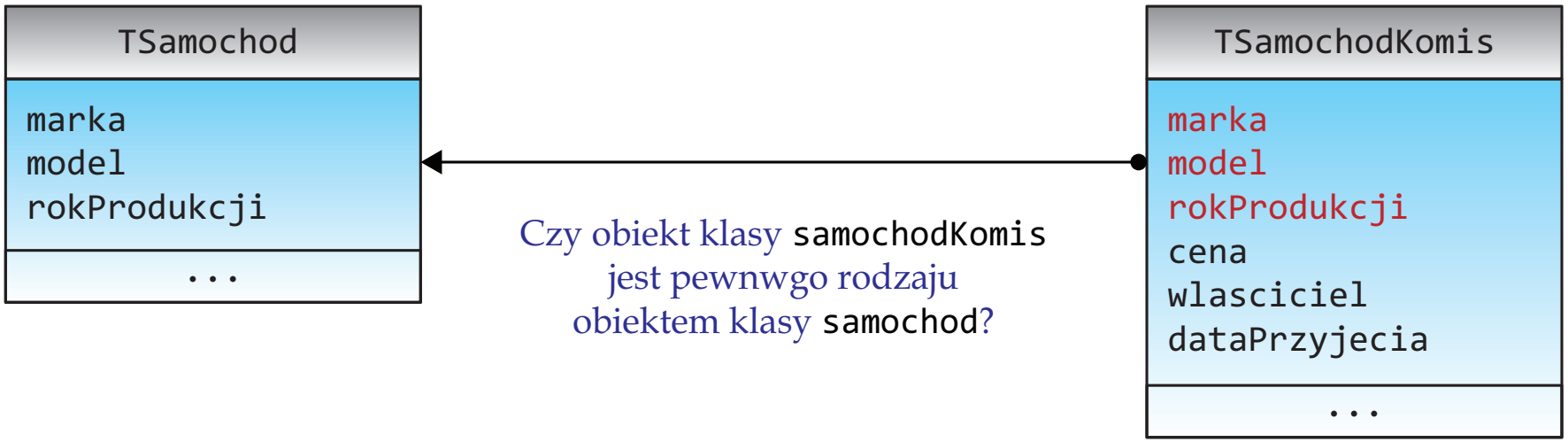

TSamochod
marka
model
rokProdukcji
...
TSamochodKomis
marka
model
rokProdukcji
cena
wlasciciel
dataPrzyjecia
...
TSamochodDiagnostyka
marka
model
rokProdukcji
dataPrzegladu
nastepnyPrzeglad
diagnosta
...
TSamochodSerwis
marka
model
rokProdukcji
przebieg
nrWyposazenia
nrSilnika
...

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.

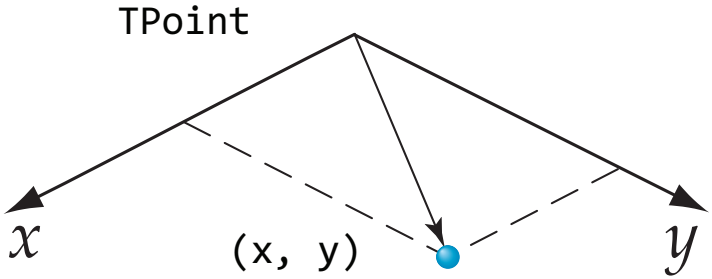
Interpretacja:

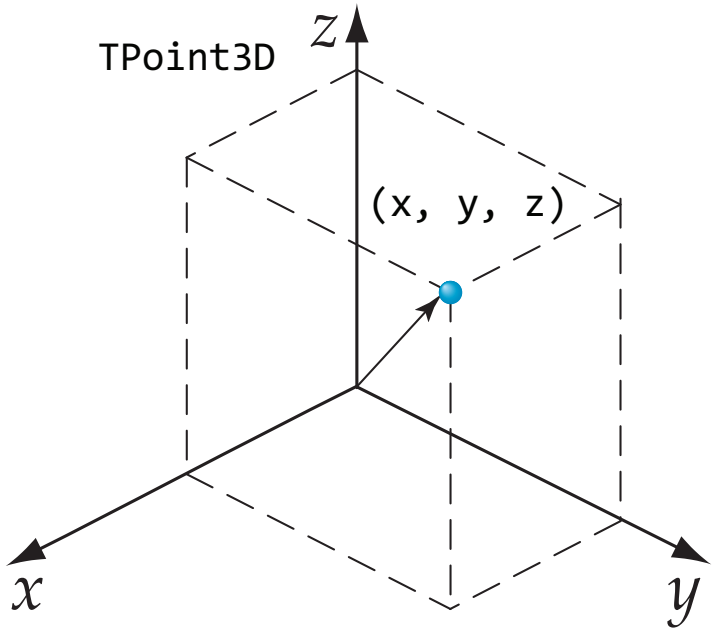
- dwie odpowiedzi nigdy: brak związku specjalizacja-generalizacja,
- dwie odpowiedzi zawsze: obiekty A i B są **synonimiczne** (np. dwie różne nazwy dla tej samej klasy)
- jeżeli: A is-a B = zawsze oraz B is-a A = czasami wtedy **A jest specjalizacją B**.

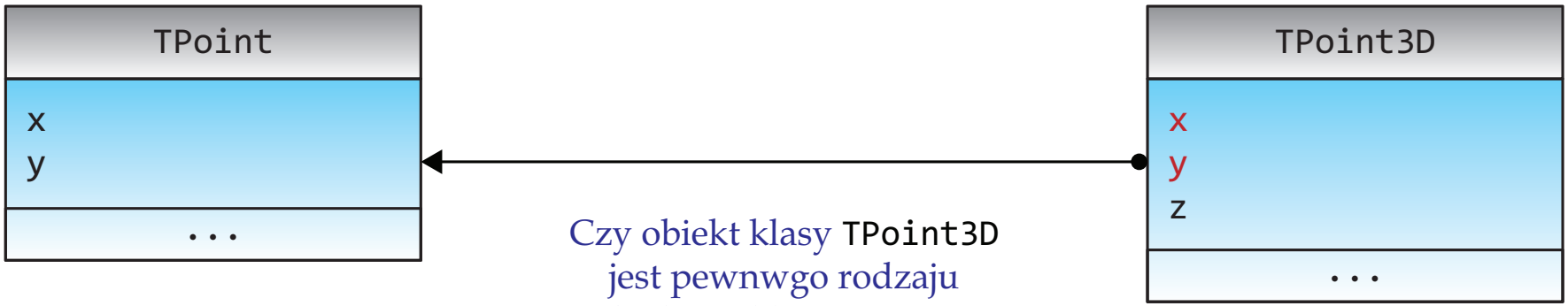
Czy klasa `TSamochodKomis` jest klasą pochodną klasy `TSamochod`:

- czy obiekt klasy `TSamochodKomis` jest obiektem klasy `TSamochod`? $\Leftarrow$ **zawsze**
- czy obiekt klasy `TSamochod` jest obiektem klasy `TSamochodKomis`? $\Leftarrow$ **czasami**

Dziedziczenie w praktyce







Czy obiekt klasy `TPoint3D` jest pewnwgo rodzaju obiektem klasy `TPoint`?

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;
};
```

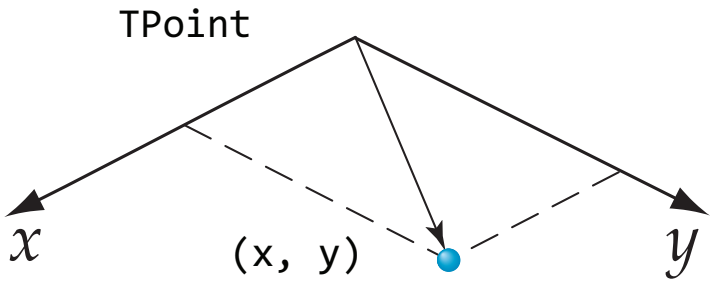
Dziedziczenie w **trybie publicznym** (`public TPoint`) zachowuje widoczność pól określoną w klasie bazowej.

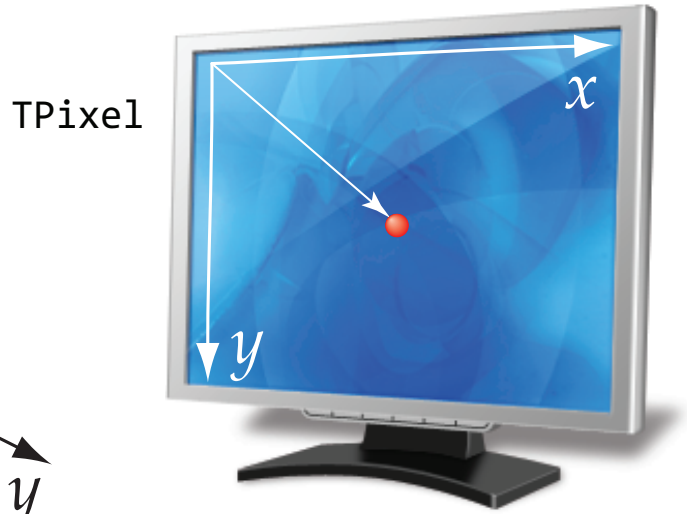
Konstruktor **klasy pochodnej** (`TPoint3D()`) aktywuje konstruktor **klasy bazowej** (`TPoint()`) umieszczony na liście inicjalizacyjnej, odbywa się to przed wywołaniem ciała konstruktora klasy pochodnej.

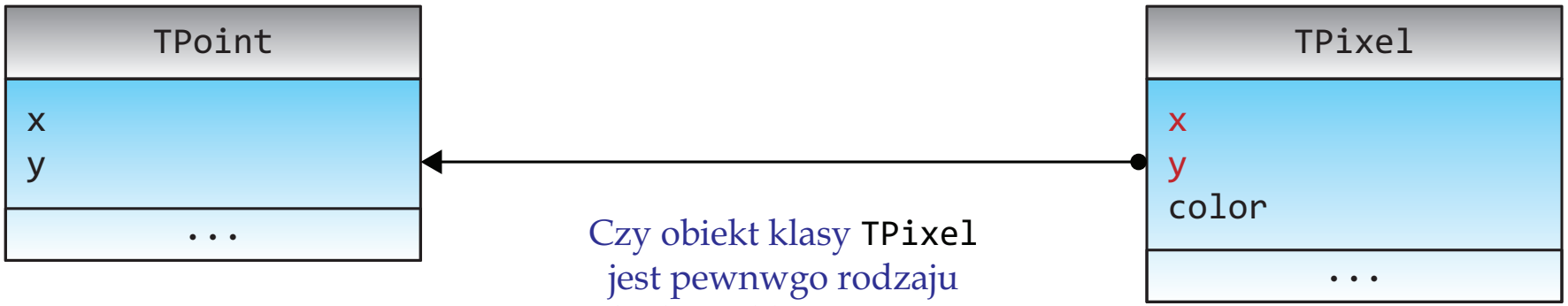
W klasie pochodnej definiujemy metody do obsługi **nowych pól** (`setZ()` i `getZ()`), obsługę **pól odziedziczonych** realizujemy z wykorzystaniem metod odziedziczonych.

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

Kolejna klasa pochodna







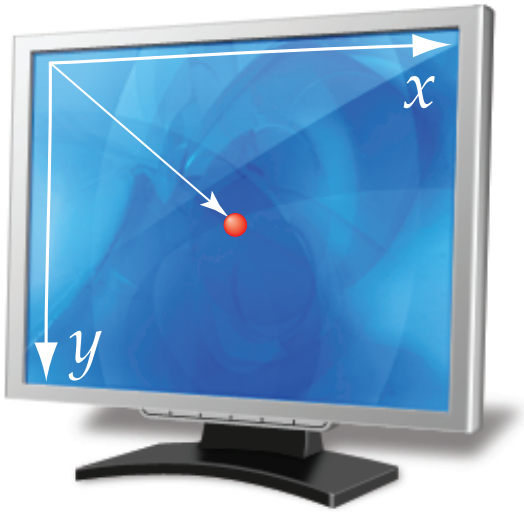
Czy obiekt klasy `TPixel` jest pewnwgo rodzaju obiektem klasy `TPoint`?

Przykład klasy pochodnej TPixel:

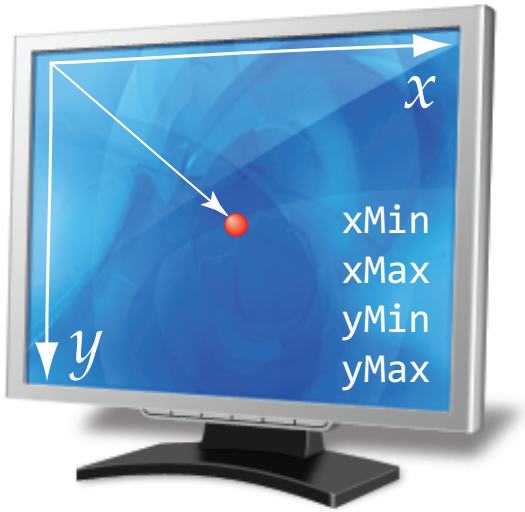
```
class TPixel : public TPoint
{
  public:
    TPixel(): TPoint(), color(0) {}
    TPixel(int x, int y, int color): TPoint(x, y), color(color) {}

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

    int getColor() const {return color;}

    void put() const
    {
      putpixel(getX(), getY(), color); // funkcja z dod. bib. graf.
    }

  private:
    int color;
};
```

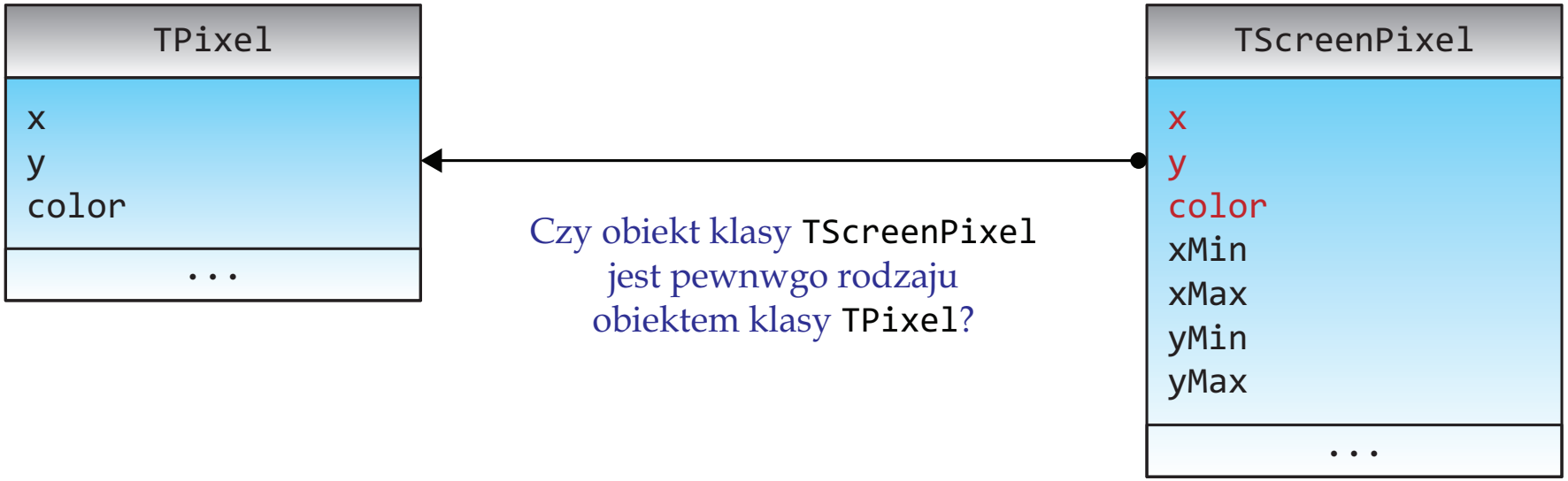
I jeszcze jedna klasa pochodna

TPixel



TScreenPixel





Czy obiekt klasy `TScreenPixel` jest pewnwgo rodzaju obiektem klasy `TPixel`?

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 do x i y
    }
};
```

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).

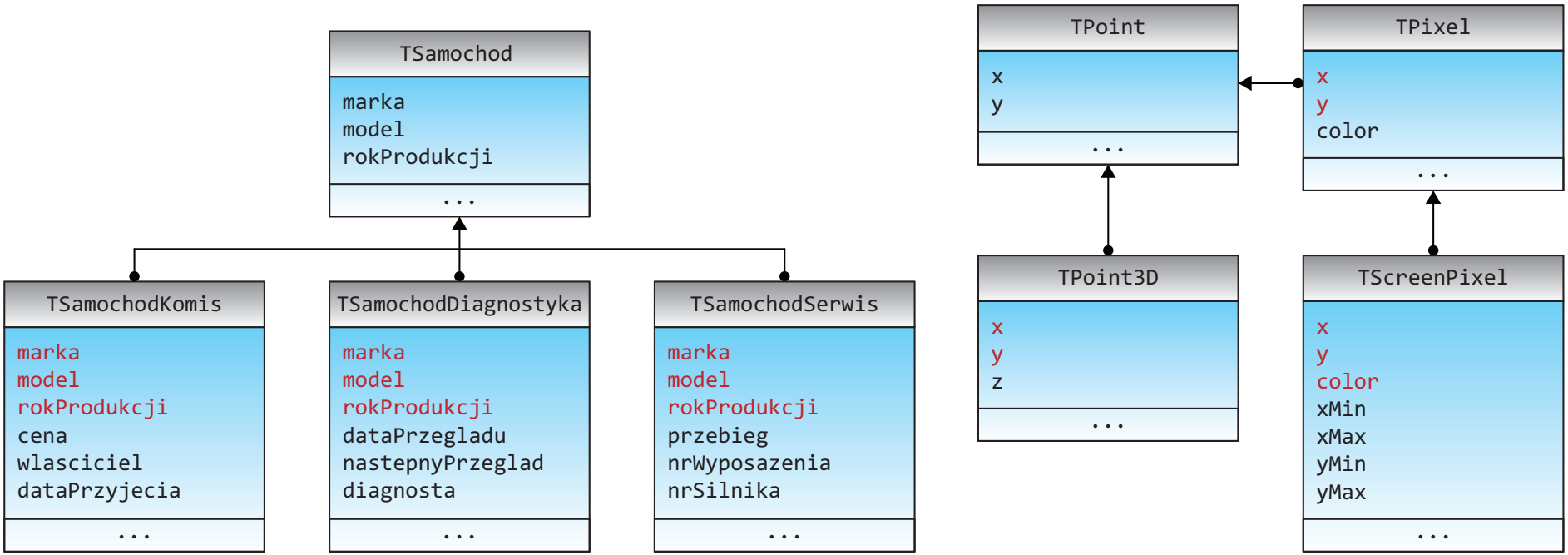
To jaka ma być **wykonana akcja**, ustalane jest na **etapie wykonania** programu, a wykonywane akcje mogą być różne.

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.

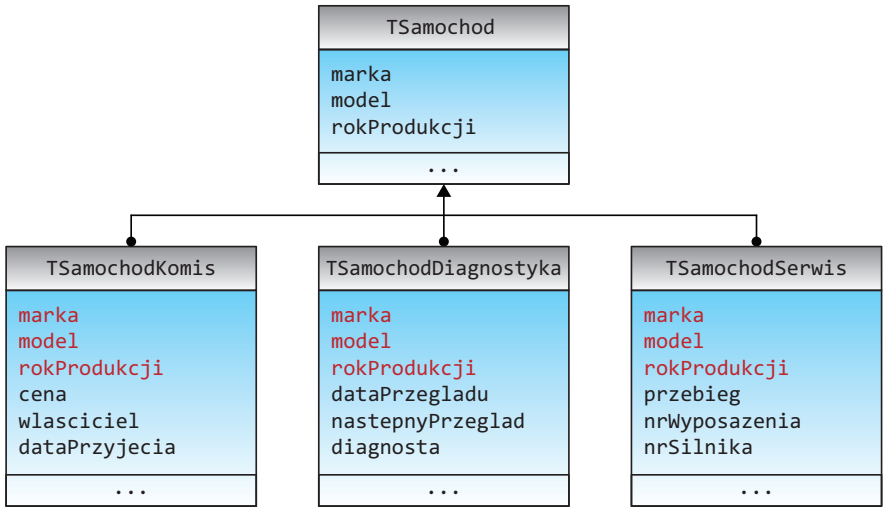
Obiekty takich klas są często do siebie podobne, ale nie jednakowe. Problem stanowi spójne i wygodne zarządzanie grupami obiektów różnych klas należących do tej samej hierarchii, zdefiniowanej dziedziczeniem.

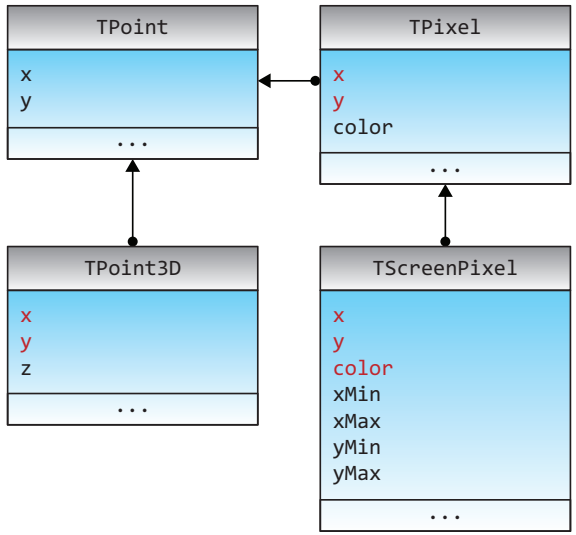


Obiekty spowinowacone mogą być traktowane w specjalny sposób.

```
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);
```

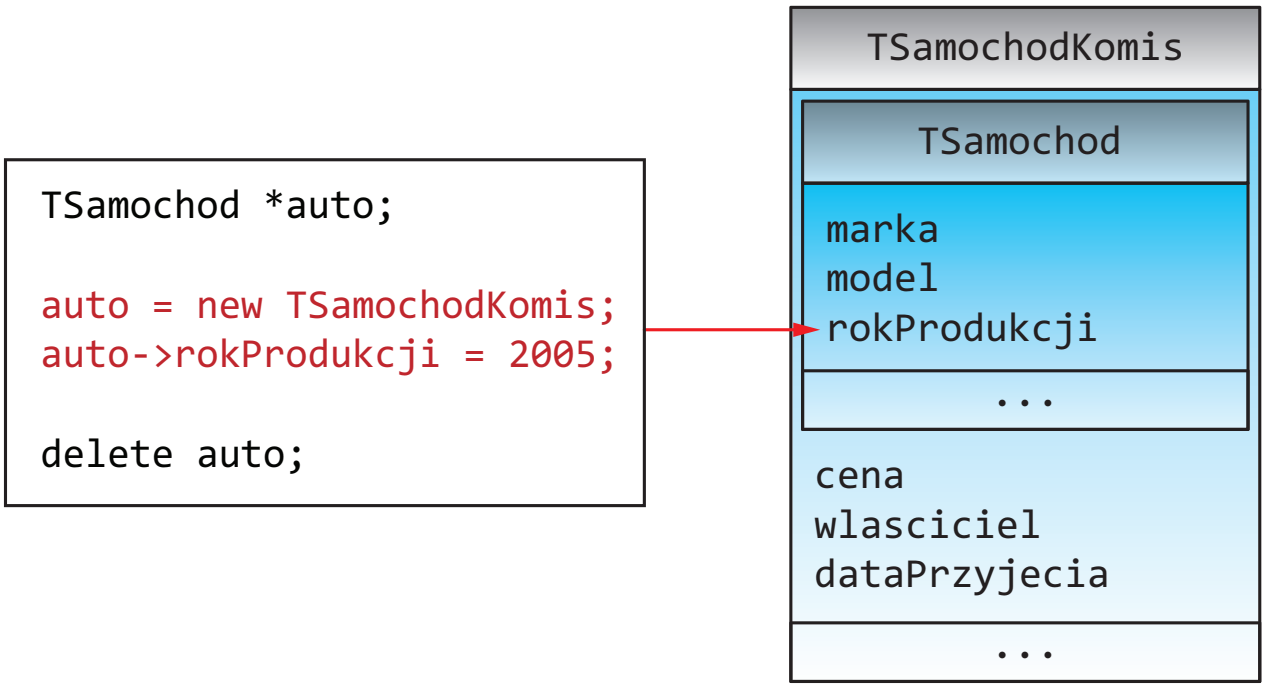




W języku C++ wolno do **wskaźnika klasy bazowej** przypisać **wskazanie obiektu klasy pochodnej**. Analogicznie jest w przypadku referencji.

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.

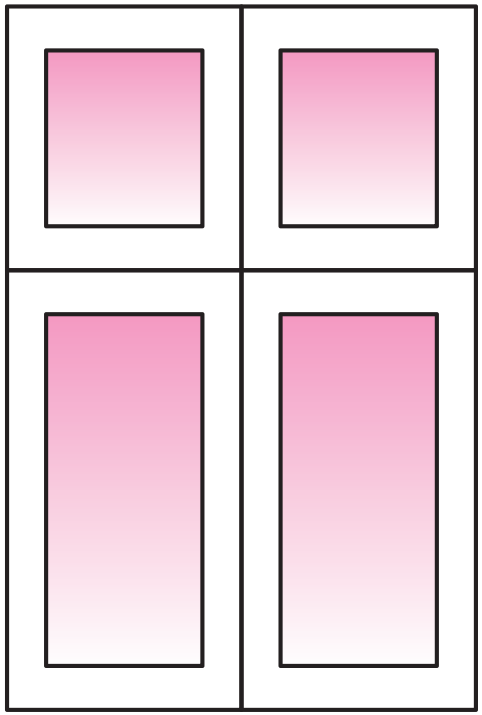


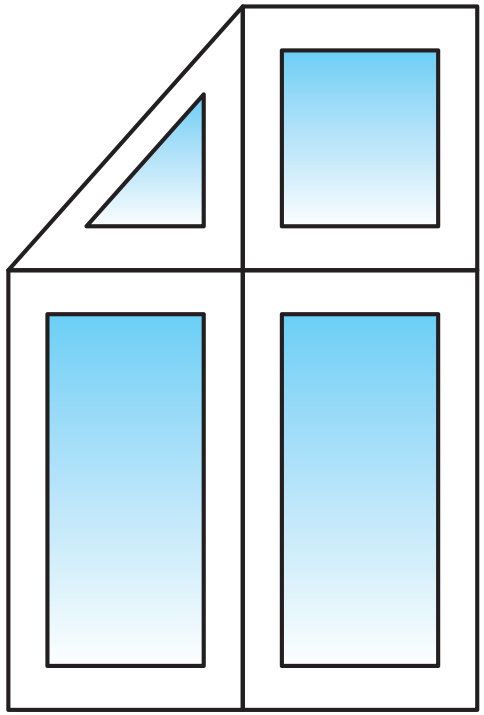
Polimorfizm w praktyce

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;}
```

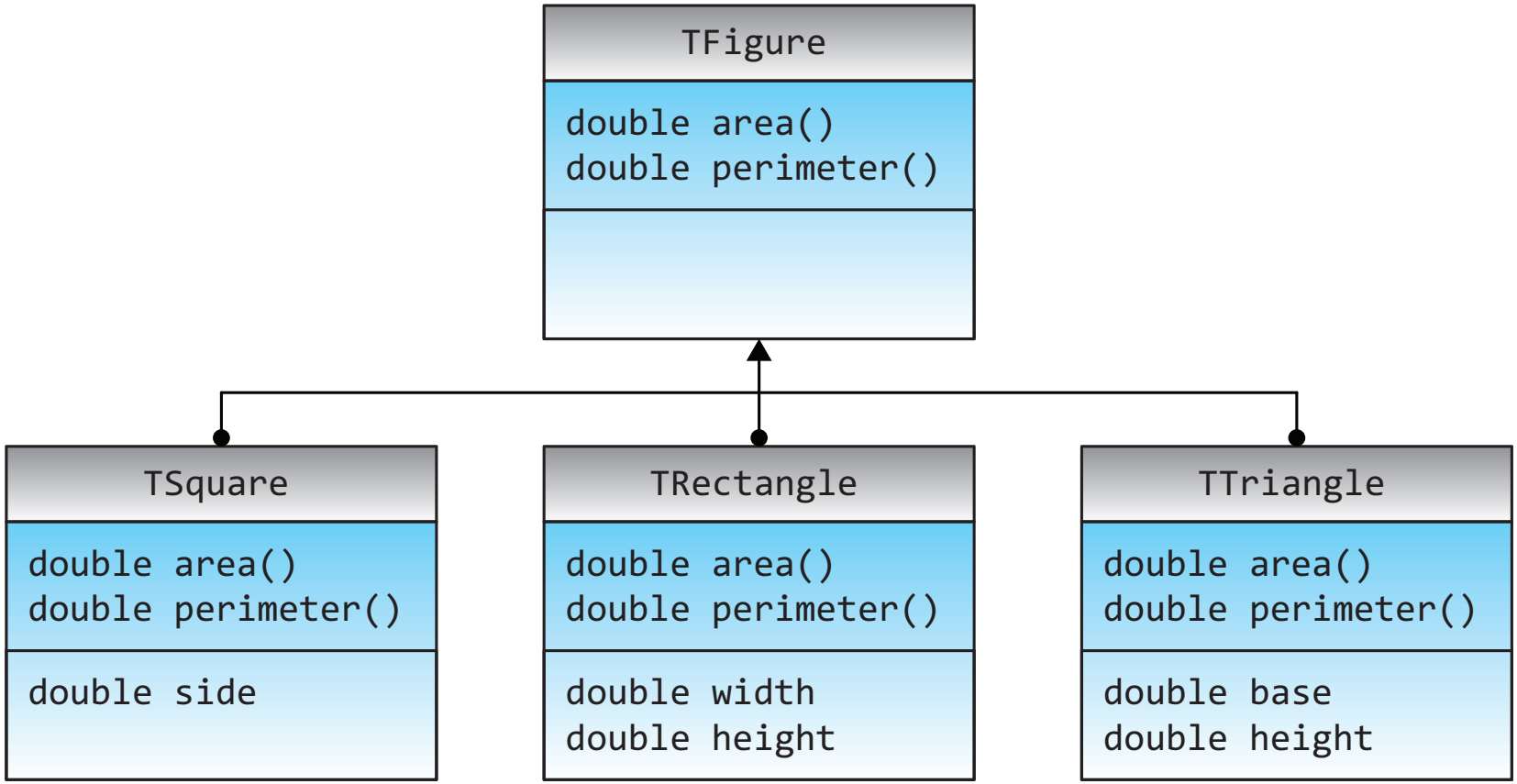
Klasa reprezentująca **trójkat** będzie klasą pochodną w stosunku do klasy `TFigure`:

```
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.

```
fig = &s;

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

Ale nie można (dlaczego?):

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

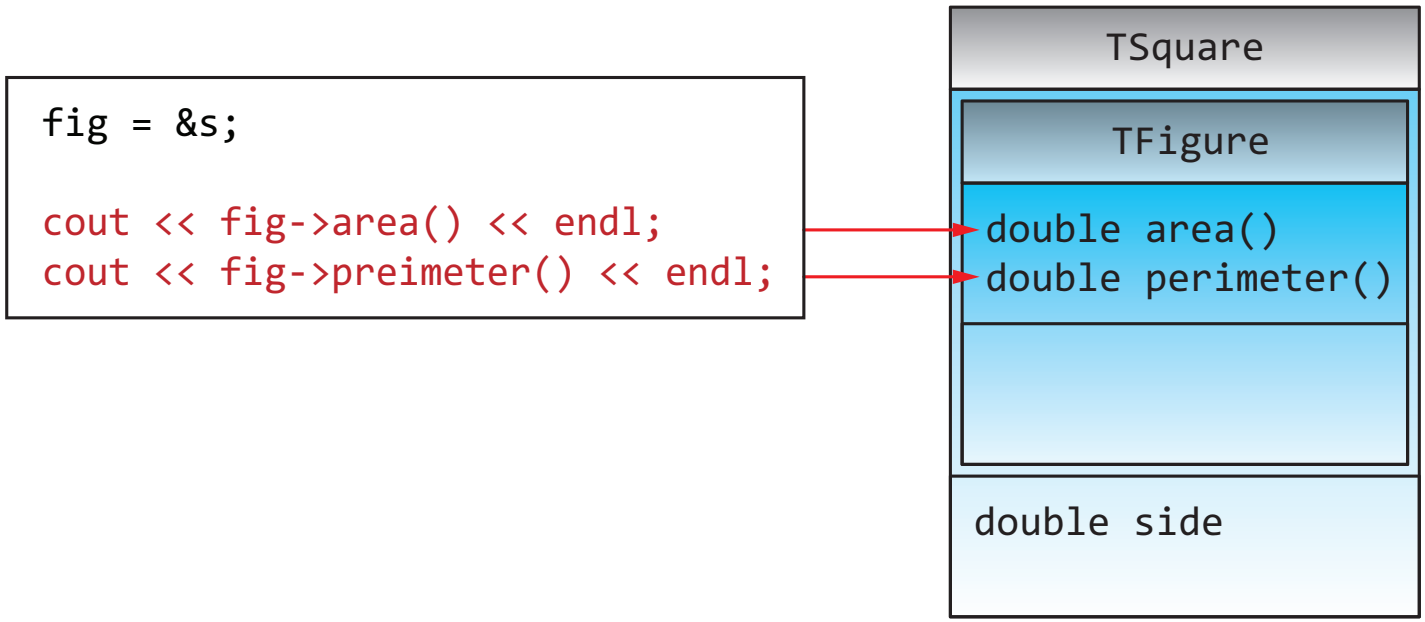
Choć można dokonać *wymuszenia* stosując brutalne rzutowanie typów (niezalecane):

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

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

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.



Rozbudujmy klasy o dodatkową funkcję określającą nazwę figury:

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

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

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

```
class TTriangle : public TFigure
{
  public:
    char *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);
```

- w końcu wywołujemy `figureInfo()`:

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

- otrzymujemy...

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:

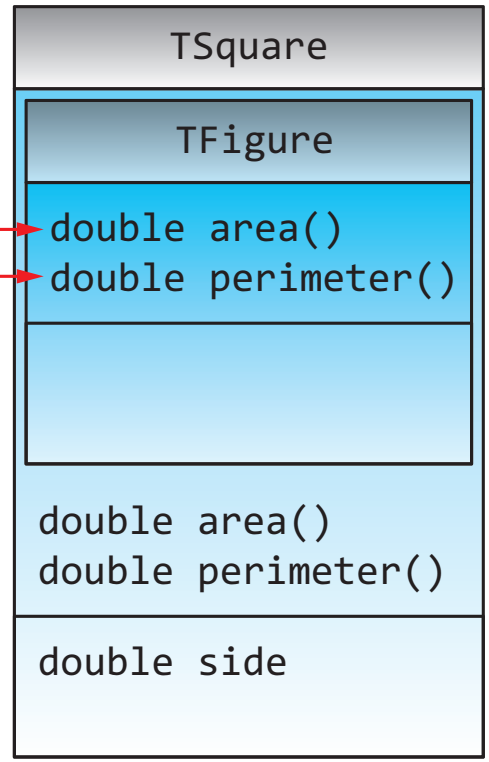
```
class TFigure
{
  char *getName() const {return "Figura";}
  double area() const {return 0;}
  double perimeter() const {return 0;}
};
```

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 char *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**!

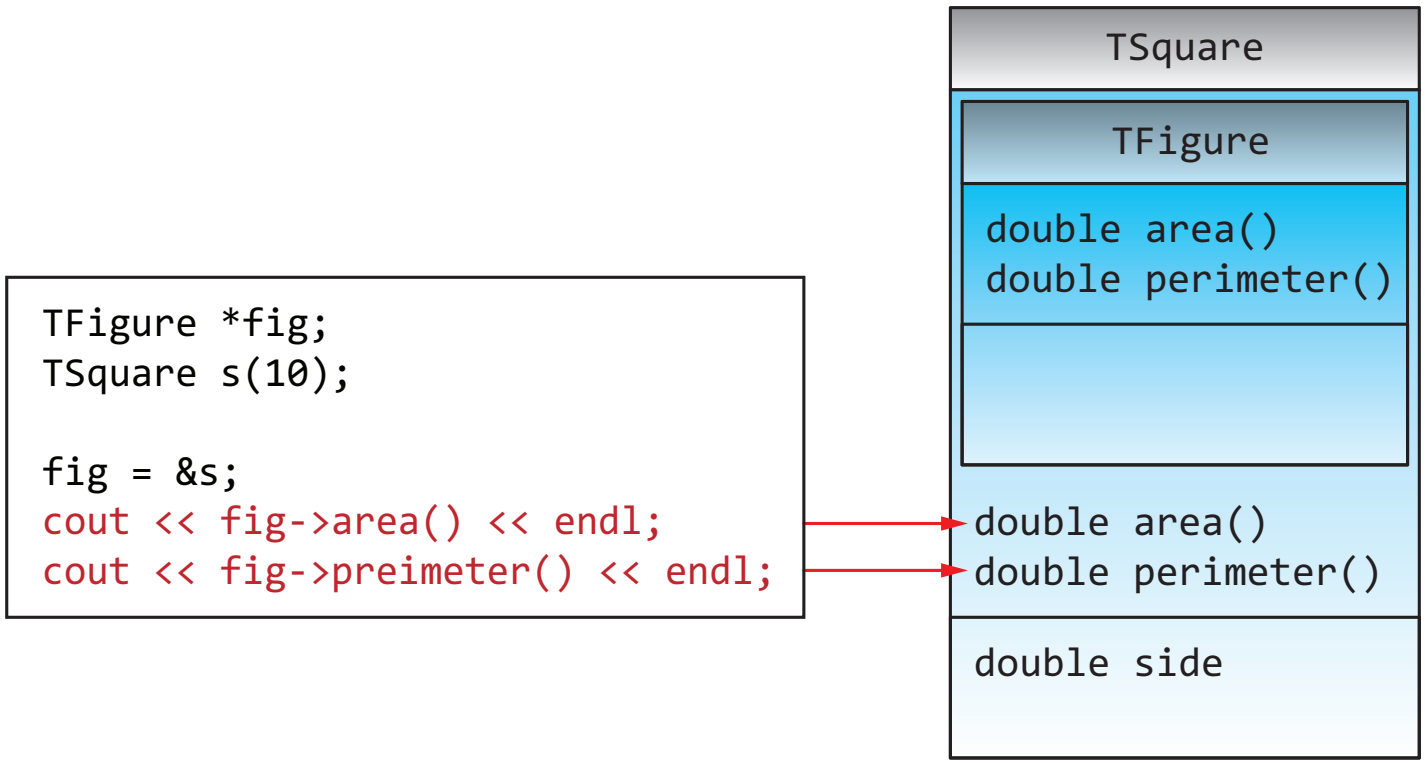
Wiązanie dynamiczne jest realizowane w języku C++ za pomocą funkcji **wirtualnych**. Funkcja wirtualna posiada w swojej deklaracji słowo kluczowe `virtual`.

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.



Klasa abstrakcyjna

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

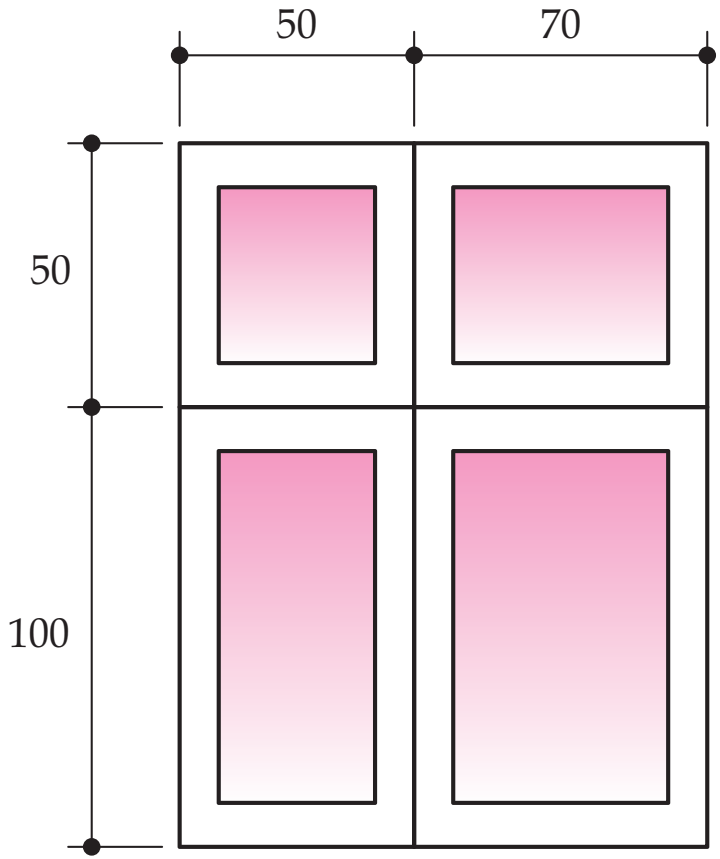
W efekcie klasa taka staje się **klasą abstrakcyjną**. Oznacza to, że iż nie jest możliwe stworzenie obiektu tej klasy.

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

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

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

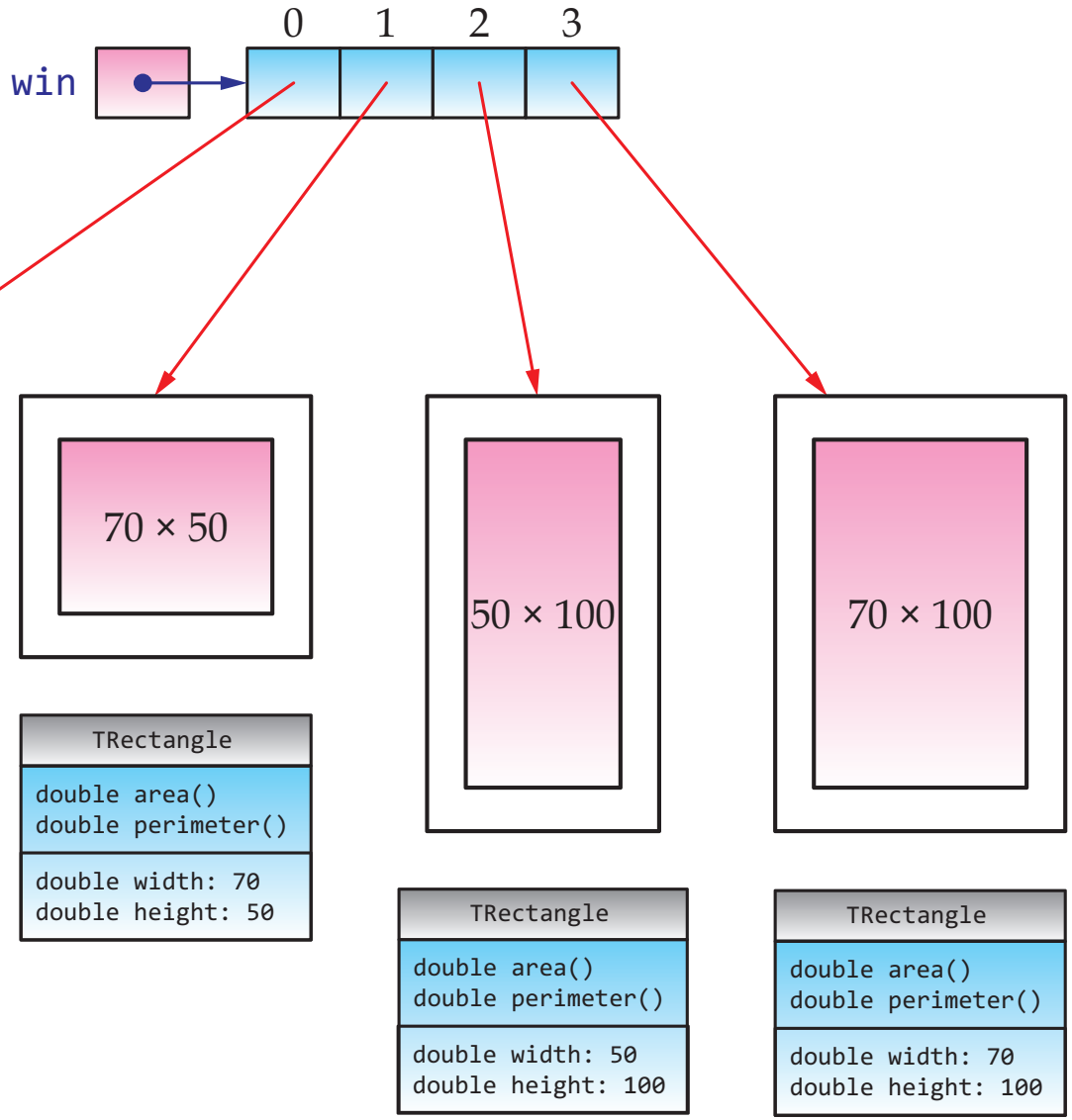
Wykorzystanie późnego wiązania

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

```
const int winNo = 4;

TFigure *win[winNo];
```



Definiowanie elementów, obliczenia, zwalnianie pamięci

```
// okreslenie liczby elementow
const int winNo = 4;

// tablica wkaznikow na elementy okna
TFigure *win[winNo];

// przydzial pamieci dla elementow okna
win[0] = new TSquare(50);
win[1] = new TRectangle(50, 70);
win[2] = new TRectangle(50, 100);
win[3] = new TRectangle(70, 100);

// obliczenia i drukowanie wynikow
calcAndInfo(win, winNo);

// zwolnianie pamieci przydzielonej elementom okna
for(int i = 0; i < winNo; delete win[i++]);
```

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`.

W przypadku języka C, funkcje przydzielające pamięć nie są częścią języka, pochodzą z dodatkowej biblioteki zarządzania pamięcią.

Funkcje przedzielające pamięć traktowane są jako amorficzne bloki, o liczonym w bajtach rozmiarze zdefiniowanym przez programistę. Typ przedzielanych obszarów nie jest funkcją znany, programista musi panować nad rozmiarami przydzielonych bloków.

W języku C++ zarządzanie pamięcią zostało włączone do języka, operatory `new` i `delete` są znane kompilatorowi.

Stosowanie operatorów `new` i `delete` zapewnia aktywowanie konstruktorów dla inicjalizacji tworzonych obiektów oraz destruktorów dla czynności kończących przy usuwaniu obiektu z pamięci.

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 stddef.h.

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.h. 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 <cstdlib>
#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 EXIT_SUCCESS;
}
```

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, czyli 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: 3.14
```

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 mozliwosci 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) - przek. 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 przekształcenia
  ...               // obiektu myInt do typu int
};

myInt i;

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

printData(i); // zgodna z printData(int) - wykorzystanie
              // przekształcenia do int z klasy myInt
```

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 newVal);
    void setImag(double newVal);

  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 newReal)
{
  real = newReal;
}

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

Użycie klasy

Kreowanie obiektów.

```
TComplex a; // konstruktor TComplex() - domyslne parametry
TComplex b(2); // konstruktor TComplex(2); parametr domyslny
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`.

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 cont 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 kopiujacy

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 newReal);
    void setImag(double newImag);

    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 = newReal;}
void TComplex::setImag(double newImag) {imag = newImag;}

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;
}
```

Wersja elegancka – bez dodatkowych zmiennych pomocniczych z wykorzystaniem konstruktora.

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

Odejmowanie liczb zespolonych

```
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

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

a = -b;
```

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

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

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

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

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.

Operatory, które można przeciążać:

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

Operatory, których przeciążać nie wolno:

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

Operacje na plikach, strumienie

Język C nie zawiera żadnego wbudowanego typu plikowego. Operacje na plikach nie są częścią języka.

Przetwarzanie plików realizowane jest zwykle przez funkcje z biblioteki obsługi standardowego wejścia i wyjścia (`stdio.h`).

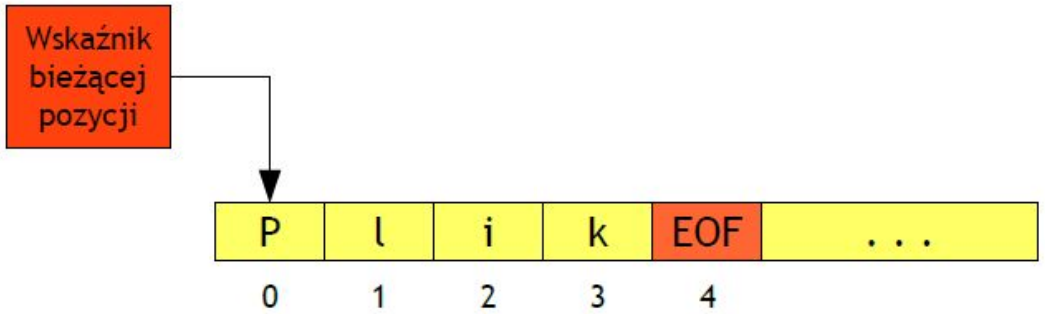
Można też korzystać z funkcji niższego poziomu (np. `io.h`) lub napisać własne funkcje.

Plik jest reprezentowany przez strumień znaków (bajtów) o zmiennej długości. Koniec strumienia identyfikowany jest znacznikiem końca pliku EOF.

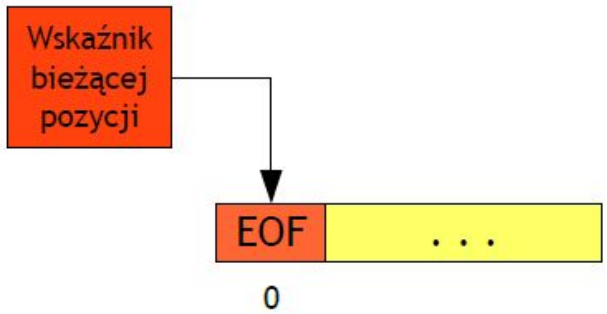
Z każdym strumieniem związany jest wskaźnik bieżącej pozycji, od której realizowane będzie czytanie lub pisanie.

Każdy zapis i odczyt zmienia wskaźnik bieżącej pozycji.

Z każdym strumieniem związany jest znacznik osiągnięcia końca pliku oraz znacznik błędu.



Plik pięcioelementowy



Plik pusty

Strumienie mogą być otwierane w trybie:

- **binarnym** – strumień jest ciągiem jednakowo traktowanych bajtów, każdy zapis i odczyt realizowany jest bez żadnych konwersji,
- **tekstowym** – strumień jest ciągiem linii tekstu zakończonych znacznikiem końca linii `\n`; w trakcie odczytu i zapisu do takiego strumienia mogą zachodzić konwersje spowodowane np. różną fizyczną reprezentacją znacznika końca wiersza (`\r\n` w DOS/Windows, pojedynczy znak `\n` w systemach UNIX/Linux).

Aby rozpocząć operacje na plikach należy zadeklarować w programie zmienną stanowiącą uchwyt do takiego pliku. W przypadku obsługi standardowych strumieni deklaruje się zmienną wskaźnikową.

Typem wskazywanym jest `FILE`, jest to zdefiniowany w pliku nagłówkowym `stdio.h` typ rekordowy, zawierający informacje o otwartym dojściu do pliku.

```
#include <stdio.h> // <cstdio> w C++

FILE *fp = NULL;
```

Wykorzystanie pliku rozpoczyna operacja jego otwarcia, realizowana zwykle przez funkcję `fopen()`.

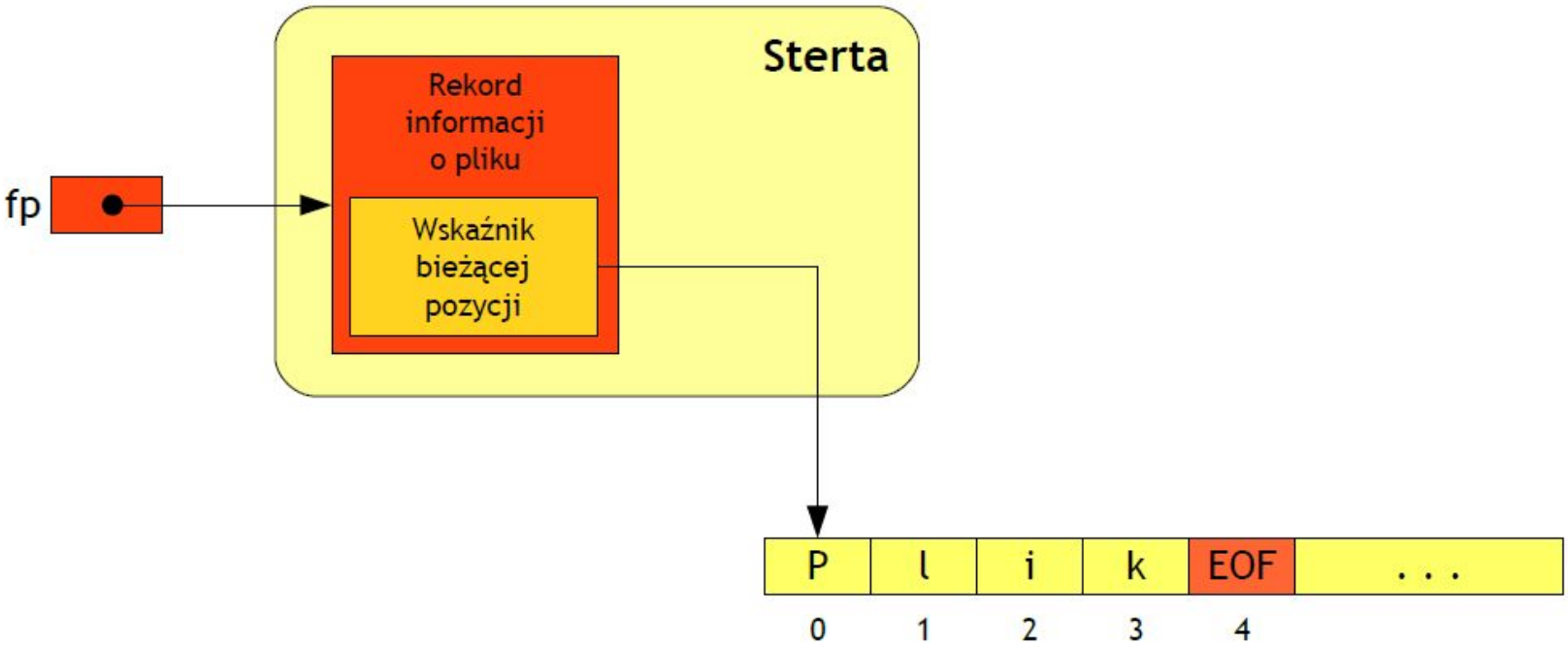
```
FILE * fopen(const char *filename, const char *mode);
```

Funkcja otwiera strumień związany z plikiem o nazwie przekazanej parametrem `filename`. Nazwa może zawierać ścieżkę dostępu do pliku. Strumień otwierany jest w trybie `mode`.

Jeżeli otwarcie zakończyło się sukcesem, funkcja udostępnia wskaźnik do dynamicznie alokowanej struktury typu `FILE`, stanowiącej programową reprezentacje fizycznego pliku.

Jeżeli pliku nie udało się otworzyć, rezultatem funkcji jest `NULL`.

```c
#include <stdio.h>

FILE *fp = NULL;
fp = fopen("file.txt", "rt");

if(fp != NULL)
// otwarcie OK
else
// otwarcie nieudane
```



Specyfikacja sposobu otwarcia pliku:

- `t` – otwarcie w trybie tekstowym,
- `b` – otwarcie w trybie binarnym.

Tryb otwarcia:

- `r` – otwarcie istniejącego pliku wyłącznie do odczytu,
- `r+` – otwarcie istniejącego pliku do odczytu i zapisu,
- `a` – zapis do istniejącego pliku lub utworzenie nowego pliku, ustawienie wskaźnika pliku na pozycji końcowej,
- `a+` – odczyt i zapis do istniejącego pliku lub utworzenie nowego pliku, ustawienie wskaźnika pliku na pozycji końcowej,
- `w` – utworzenie pliku wyłącznie do zapisu, jeżeli plik istnieje jest nadpisywany,
- `a` – utworzenie pliku do odczytu i zapisu, jeżeli plik istnieje jest nadpisywany.

Znak + w trybie otwarcia oznacza aktualizację – możliwość czytania i pisania do otwartego strumienia.

Zapis i odczyt nie mogą po sobie następować bezpośrednio. Należy użyć funkcji do **wymiatania** bufora `fflush()` lub jednej z funkcji pozycjonowania wskaźnika pozycji: `fseek()`, `fsetpos()`, `rewind()`.

Jeżeli informacja o trybie otwarcia nie występuje, przyjmowany jest tryb zgodnie z wartością globalnej zmiennej `_fmode`.

Jeśli zmienna `_fmode` ma wartość `O_BINARY`, plik jest otwierany w trybie binarnym, wartość `O_TEXT` powoduje otwarcie pliku w trybie tekstowym.

Symbole `O_BINARY` i `O_TEXT` zdefiniowane są w pliku `<fcntl.h>`. Domyślna wartość `_fmode` to `O_TEXT`.

Otwieranie i zamykanie plików

Otwarcie pliku data.txt jako pliku **tekstowego** do **odczytu**:

```
fp = fopen("data.txt", "rt");
```

Otwarcie pliku image.bmp jako pliku **binarnego** do **odczytu**:

```
fp = fopen("image.bmp", "rb");
```

Otwarcie pliku raport.doc jako pliku tekstowego do **zapisu**, wskaźnik pliku ustawiany jest w pozycji końcowej, jeżeli nie istnieje, tworzony jest nowy plik.

```
fp = fopen("raport.doc", "at");
```

Otwarcie pliku image.jpg jako pliku binarnego, do **zapisu** i **odczytu**, jeżeli plik istnieje zostanie nadpisany.

```
fp = fopen("image.jpg", "w+b");
```

```
int fclose(FILE *stream);
```

Funkcja zamyka strumień stream i zapisuje wszystkie bufory.

Rezultat EOF oznacza błąd zamykania, rezultat równy **zero** oznacza poprawne zamknięcie pliku.

Pamięć przydzielona strukturze wskazywanej przez wskaźnik `stream` jest zwalniana.

```
#include <stdio.h>

FILE *fp = NULL;
fp = fopen("file.txt", "rt");

if(fp != NULL)
{
// otwarcie OK, operacje na pliku
  fclose(fp);
}
```

Odczyt i zapis pojedynczych znaków

```
int fgetc(FILE *stream);
```

Funkcja pobiera następny znak ze strumienia `stream` i **uaktualnia wskaźnik** bieżącej pozycji w pliku. Znak pobierany jest jako `unsigned char` i przekształcany jest do typu `int`.

W przypadku napotkania końca strumienia, rezultatem jest wartość `EOF` oraz ustawiany jest **znacznik napotkania końca strumienia**.

W przypadku wystąpienia błędu odczytu, rezultatem funkcji jest wartość `EOF` oraz ustawiany jest znacznik **błędu strumienia**.

Przykładowe wykorzystanie – odczyt znaku z uprzednio otwartego pliku fp:

```
#include <stdio.h>

FILE *fp = NULL;

if((fp = fopen("data.txt", "rt")) != NULL)
{
  int c;
  c = fgetc(fp);
  printf("Przeczytano znak %c", c);
  fclose(fp);
}
```

```
int fputc(int c, FILE *stream);
```

Funkcja wyprowadza znak `c` do strumienia `stream` zgodnie ze wskaźnikiem bieżącej pozycji w pliku.

W przypadku, gdy funkcja zakończyła swoje działanie bez błędu, rezultatem funkcji jest znak `c`. W przeciwnym wypadku wartość `EOF`.

Przykładowe wykorzystanie – zapis znaku do uprzednio otwartego pliku `fp`:

```
#include <stdio.h>

FILE * fp = NULL;

if((fp = fopen("data.txt", "wt")) != NULL)
{
  fputc('C', fp);
  fclose(fp);
}
```

Przetwarzanie sekwencyjne

```
int feof(FILE *stream);
```

Rezultatem funkcji jest wartość **różna od zera**, jeżeli strumień jest w pozycji końcowej, **zero** w przeciwnym wypadku.

Strumień jest w **pozycji końcowej**, jeżeli w wyniku ostatnio przeprowadzonej operacji **odczytano znacznik końca pliku**.

Przykładowe wykorzystanie – wypisywanie do strumienia wyjściowego zawartości pliku i obliczanie liczby znaków:

```
FILE *fp;
long int counter = 0;

if((fp = fopen("data.txt", "rt")) != NULL)
{

  while(!feof(fp))
  {
    putchar(fgetc(fp));
    counter++;
  }

  fclose(fp);
  printf("\nLiczba znakow w pliku: %ld", counter - 1);
}
```

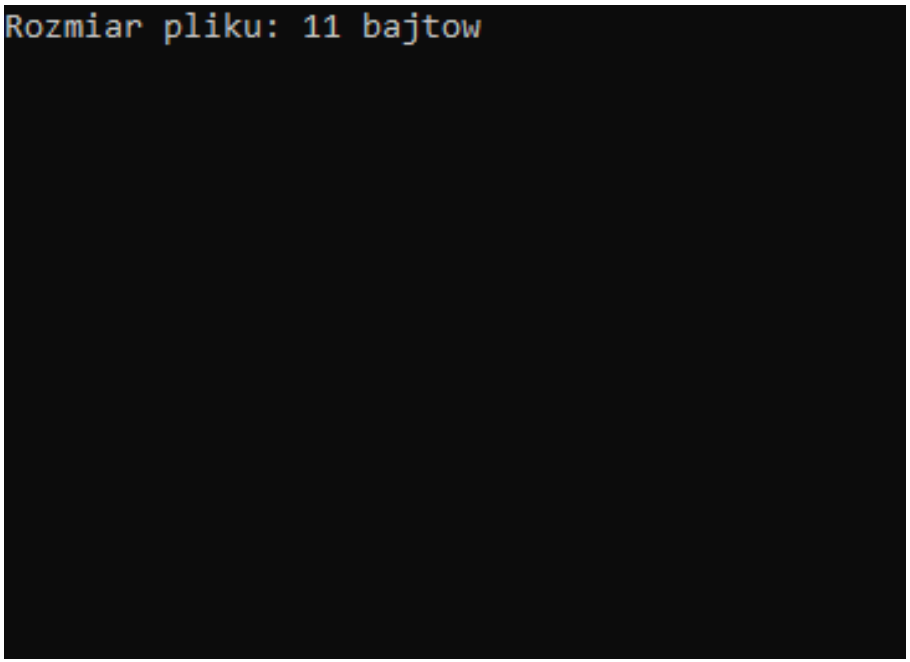
Przykładowe wykorzystanie – bez wykorzystania `feof()`:

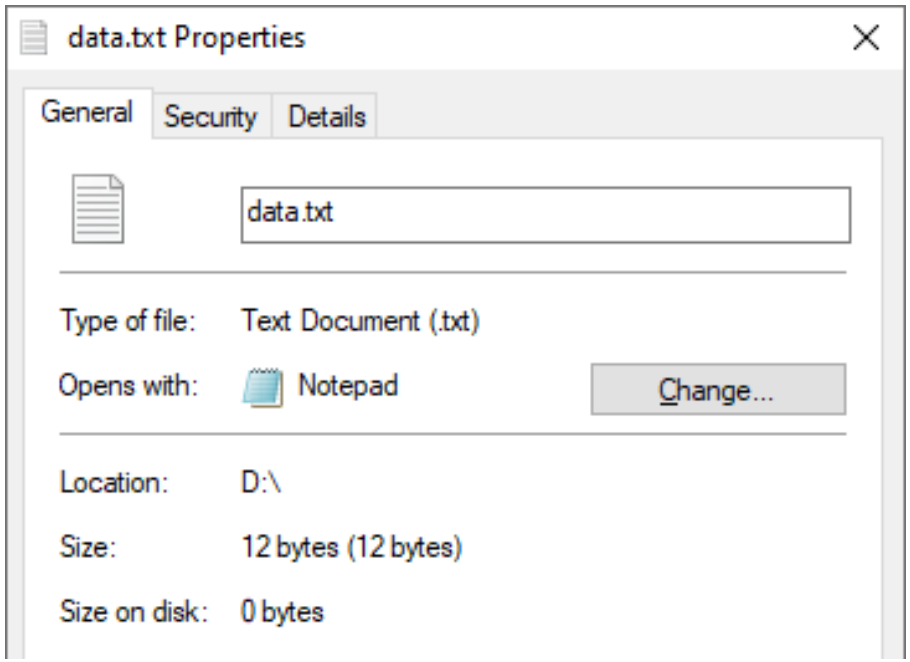
```
FILE *fp;
long int counter = 0;

int c;
if((fp = fopen("data.txt", "rt")) != NULL)
{

  while((c = fgetc(fp)) != EOF)
  {
    putchar(c);
    counter++;
  }

fclose(fp);
printf("\nLiczba znakow w pliku: %ld", counter);
}
```

Rozmiar pliku, znak końca linii i problemy

Wyznaczanie rozmiaru pliku (trochę inaczej):

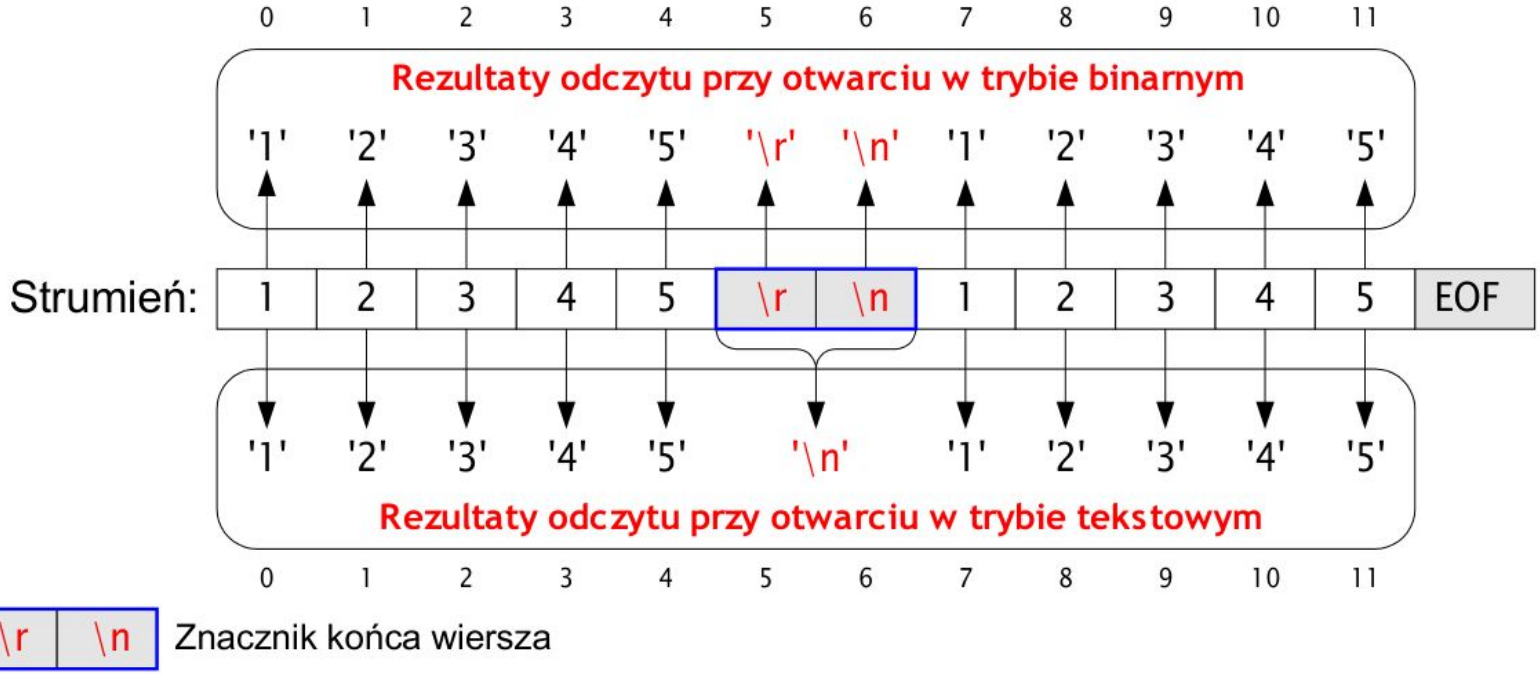
```
if((fp = fopen("data.txt", "rt")) != NULL)
{
  for(counter = 0; fgetc(fp) != EOF; counter++);
  fclose(fp);
  printf("Rozmiar pliku: %ld bajtow", counter);
}
```





Przyczyną wadliwego działania programu są konwersje znaczników końca linii w trybie tekstowym.

W systemach DOS/Windows znacznik końca linii to para `\r` i `\n` (czyli `CR` i `LF`). W trakcie odczytu w trybie tekstowym, każda para `\r\n` zamieniana jest na pojedynczy znak `\n`.



Konwersje nie zachodzą przy otwieraniu pliku w trybie binarnym. W drugim parametrze wywołania `fopen()` należy użyć litery `b`, oznaczającej otwarcie w trybie binarnym.

```
if((fp = fopen("data.txt", "rb")) != NULL)
{
  for(counter = 0; fgetc(fp) != EOF; counter++);
  fclose(fp);
  printf("Rozmiar pliku: %ld bajtow", counter);
}
```

Przetwarzanie plików linia po linii

Pliki tekstowe można **przetwarzać wierszami**, od strony programu separatorem wierszy jest znak \n.

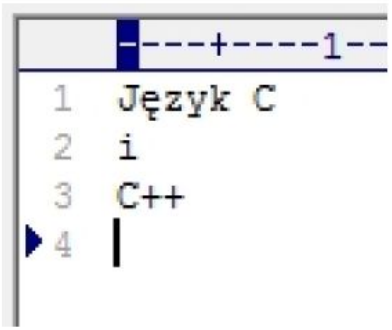
Do przetwarzania pliku tekstowego linia po linii służą funkcje odczytu/zapisu linii – buforem linii są **tablice znakowe**.

Zawartość pliku:

J	ę	z	y	k		C	\n	i	\n	C	+	+	\n	EOF

Przy przetwarzaniu linia po linii:

J	ę	z	y	k		C	\n
i	\n						
C	+	+	\n				
EOF							

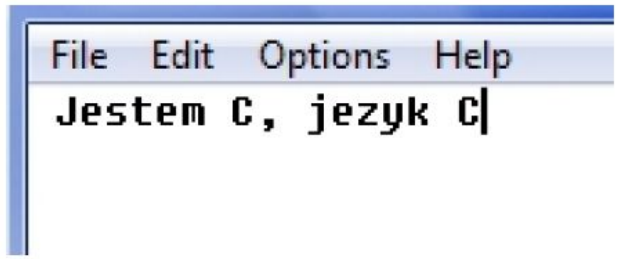


Odczyt linii lekstu z pliku realizuje znana już funkcja `fgets()`, a zapis np. `fputs()`, `fprintf()`.

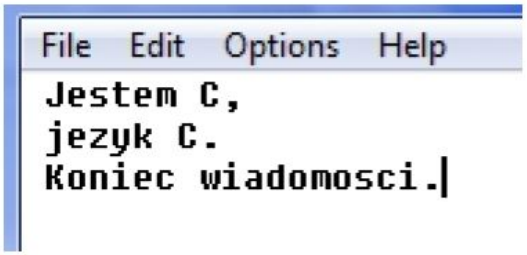
```
int fputs(const char *s, FILE *stream);
```

Funkcja `fputs()` wyprowadza napis `s` do pliku `stream`, nie dopisuje znaczników końca wiersza ani końca napisu.

Rezultatem funkcji jest **ostatni zapisany znak**, w przypadku gdy zapis zakończył się sukcesem lub `EOF`, gdy wystąpił błąd.




```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  FILE *fp = NULL;

 if((fp = fopen( "data.txt", "wt")) != NULL)
  {
    fputs("Jestem C", fp);
    fputs(", jezyk C", fp);
    fclose(fp);
  }

  return EXIT_SUCCESS;
}
```




```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  FILE *fp = NULL;

  if((fp = fopen("date.txt", "wt")) != NULL)
  {
    fputs("Jestem C", fp);
    fputs(",\njezyk C.\nKoniec wiadomosci.", fp);
    fclose(fp);
  }

  return EXIT_SUCCESS;
}
```

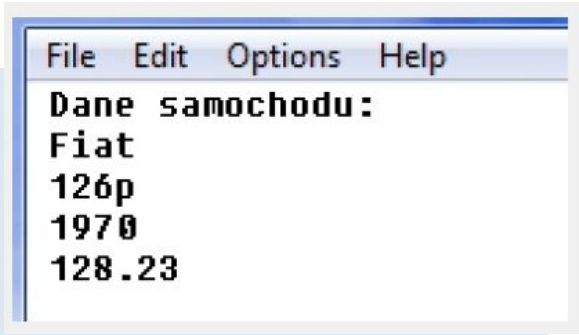
```
int fprintf(FILE *stream, const char *format [, argument, ...]);
```

Funkcja `fprintf()` wyprowadza do pliku `stream` napis `format` oraz opcjonalne argumenty, w postaci określonej przez sekwencje formatujące zapisane w napisie `format`.

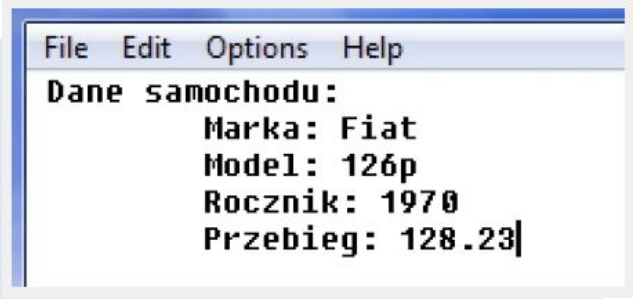
Rezultatem funkcji jest **liczba wyprowadzonych bajtów**, w przypadku gdy zapis zakończył się sukcesem lub `EOF`, gdy wystąpił błąd.

Wszystkie zasady formatowania znane z wykorzystania funkcji `printf()` obowiązują dla funkcji `fprintf()`.

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  FILE *fp = NULL;

  if((fp = fopen("data.txt", "wt")) != NULL)
  {
    char brand[80] = "Fiat";
    char model[80] = "126p";
    int year = 1970;
    float mileage = 128.23;

    fprintf(fp, "Dane samochodu:\n%s\n%s\n%d\n%g", brand, model,
      year, mileage);

    fclose(fp);
  }

  return EXIT_SUCCESS;
}
```



```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  FILE *fp = NULL;

  if((fp = fopen("data.txt", "wt")) != NULL)
  {
    char brand[80] = "Fiat";
    char model[80] = "126p";
    int year = 1970;
    float mileage = 128.23;

    fprintf(fp, "Dane samochodu:\n\tMarka: %s\n\tModel: %s\n", brand,
      model);
    fprintf(fp, "\tRocznik: %d\n\tPrzebieg: %g", year, mileage);

    fclose(fp);
  }

  return EXIT_SUCCESS;
}
```



```
char * fgets(char *s, int n, FILE *stream );
```

Funkcja `fgets()` wczytuje dane do bufora `s` ze strumienia (pliku) `stream`.

Parametr `n` określa maksymalną pojemność bufora, uwzględniającą miejsce na znacznik końca napisu.

Działanie funkcji kończy się gdy funkcja odczyta $n - 1$ znaków lub wcześniej zostanie odczytany znak nowego wiersza. Znacznik końca napisu dopisywany jest na jego końcu.

```
Dane samochodu:
        Marka: Fiat
        Model: 126p
        Rocznik: 1970
        Przebieg: 128.23
```

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_LINE 256

int main()
{
  FILE *fp = NULL;

  if((fp = fopen( "data.txt", "rt")) != NULL)
  {
    char line[MAX_LINE];

    while(fgets(line, MAX_LINE, fp) != NULL)
      printf(line);

    fclose(fp);
  }

  return EXIT_SUCCESS;
}
```

Przed znacznikiem końca pliku `EOF` może nie być znacznika końca wiersza `\n`. Funkcja `fgets()` go nie przeczyta.

Zawartość pliku:

J	ę	z	y	k		C	\n	i	\n	C	+	+	EOF

Przy przetwarzaniu linia po linii:

J	ę	z	y	k		C	\n
i	\n						
C	+	+	EOF