Podstawy programowania

dr inż. Sławomir Koczubiej

Politechnika Świętokrzyska
Wydział Zarządzania i Modelowania Komputerowego
Katedra Technologii Informatycznych

(29 października 2026)

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

Kontakt

Budynek C, pokój 3.26
sk@tu.kielce.pl

Materiały do pobrania, aktualności, terminy zaliczeń

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

Organizacja wykładów

- Program wykładów obejmuje zagadnienia wykraczające poza ćwiczenia laboratoryjne.
- Uczestnictwo w wykładach pozwala zdobyć wiedzę fundamentalną, która nie jest omawiana szczegółowo na laboratoriach.
- Regularne uczestnictwo w wykładach jest najlepszym sposobem na zdanie egzaminu.
- Czasem sprawdzam listę obecności.

Warunki zaliczenia wykładu

Egzamin zaliczeniowy po zakończeniu wykładów.

Organizacja laboratoriów

- Zajęcia laboratoryjne są obowiązkowe.
- Dopuszcza się jedną nieobecność.
- Większa liczba nieobecności powoduje zmniejszenie oceny do niedostatecznej włącznie (30% lub więcej).
- W przypadku usprawiedliwionej nieobecności zajęcia można odrobić z inną grupą (jeśli istnieje taka możliwość).
- Zajęcia praktyczne stanowią jedynie wycinek kursu.

Warunki zaliczenia laboratoriów

Wykonanie ćwiczeń i zaliczenie sprawdzianów kontrolnych.

Tematyka wykładów

- Pojęcie algorytmu, języki zapisu algorytmów. Przykłady algorytmów rozwiązujących wybrane zadania.
- Paradygmaty programowania. Język programowania, jednostki leksykalne.
- Podstawowe struktury sterujące. Syntaktyka i semantyka instrukcji. Operatory i wyrażenia.
- Typy danych, zmienne, stałe. Złożone typy danych: tablice, łańcuchy, struktury, typy wyliczeniowe. Przykłady użycia poszczególnych typów.
- Funkcje i procedury, definicja i deklaracja. Przekazywanie parametrów do podprogramów. Zmienne lokalne i globalne. Biblioteki.
- Pliki, rodzaje, praca z plikami i napisami.

Tematyka laboratoriów

- Struktura programu w języku C. Rola plików nagłówkowych.
- Operacje wejścia-wyjścia.
- Operatory w programie, definiowanie wyrażeń operatorowych. Własności i priorytety operatorów.
- Instrukcje warunkowa i przełączające, pętle. Algorytmy przetwarzania iteracyjnego.
- Tablice, definiowanie tablic. Przetwarzanie tablic, w szczególności iteracyjne.
- Definiowanie funkcji; parametry formalne i aktualne, przekazywanie parametrów w programie.
- Rodzaje błędów i ich diagnozowanie. Testowanie programu.
- Obsługa plików.

Tematyka projektu

- Środowiska pracy programisty. Praca w systemach z rodziny GNU i Windows.
- Wyrażenia regularne, wykorzystanie praktyczne.
- Narzędzia wspomagające pracę z plikami, wyszukiwanie plików, przeszukiwanie plików.
- Narzędzia tekstowe wspomagające pracę programisty.
- Zarządzanie projektami programistycznymi i wersjami.
- Praca w grupach.

Literatura

- Dokumentacja poszczególnych narzędzi programistycznych: `MSYS2`, `grep`, `sed`, `awk`, `bash`, `findutils`, `diffutils`.
- Dougherty D., Robbins A. *sed & awk.* Wydawnictwo Helion, Gliwice 2016.
- Friedl J. E. F. *Wyrażenia regularne.* Wydawnictwo Helion, Gliwice 2001.
- Kernighan B., Ritchie D. *Język ANSI C.* Wydawnictwo Naukowo Techniczne, Warszawa 2004.
- Prata S. *Szkoła programowania. Język C.* Wydawnictwo Helion, Gliwice 2006.

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

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**.

Java (podobnie jak C#) jest kompilowana nie do kodu maszynowego, lecz do **kodu bajtowego** (*bytecode*) maszyny wirtualnej. Na kod maszynowy tłumaczy go dopiero w trakcie wykonania kompilator **JIT** (*just-in-time*).

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, w latach 90. powstały narzędzia **RAD** (*Rapid Application Development*), np. Delphi czy Visual Basic, umożliwiające budowanie programów z gotowych komponentów. Wizualne projektowanie interfejsu jest dziś standardową częścią środowisk programistycznych, a idea RAD wróciła pod nazwą platform **low-code** – z tymi samymi zaletami (szybki start) i wadami (ograniczenia gotowych klocków, uzależnienie od dostawcy).

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 mucioszą!” 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 kompilacji, np. użycie niezadeklarowanej zmiennej lub niezgodność typów, lecz inne mogą ujawnić się dopiero w trakcie wykonywania.

Błędy

W trakcie pisania kodu nie da się uniknąć błędów. Ze względu na moment, w którym się ujawniają, można je podzielić na:

- **błędy kompilacji** – **syntaktyczne** (składniowe), np. brak średnika, nawiasu, przecinka, oraz **semantyczne** wykrywane przez kompilator, np. niezadeklarowana zmienna, niezgodność typów, zła liczba argumentów funkcji,
- **błędy konsolidacji** – kod się skompilował, ale konsolidator nie znalazł definicji użytej funkcji lub zmiennej (`undefined reference`, `unresolved external symbol`) albo znalazł ich kilka (`multiple definition`),
- **błędy wykonania** – ujawniają się dopiero po uruchomieniu, np. dzielenie całkowite przez zero, odwołanie do nieistniejącego elementu tablicy, otwarcie nieistniejącego pliku, brak pamięci,
- **błędy logiczne** – program działa, ale robi coś innego niż zamierzono.

Im wcześniej błąd zostanie wykryty, tym łatwiej go poprawić. Błąd kompilacji wskazuje konkretny wiersz, błąd logiczny nie wskazuje niczego.

Oprócz błędów kompilator zgłasza **ostrzeżenia** (*warnings*). Nie przerywają one kompilacji, ale bardzo często wskazują prawdziwy błąd: użycie niezainicjowanej zmiennej, `=` zamiast `==` w warunku, format w `printf()` niezgodny z typem argumentu.

Ostrzeżenia warto włączyć możliwie szeroko (`gcc -Wall -Wextra`, w MSVC poziom `/W4`) i traktować poważnie. Dobry program kompiluje się bez ostrzeżeń.

W języku C szczególnie groźne jest **niezdefiniowane zachowanie** (*undefined behavior*). Standard nie określa, co się stanie, np. przy wyjściu poza tablicę, odczycie niezainicjowanej zmiennej czy przepełnieniu typu `int`. Program może działać poprawnie, zachowywać się inaczej po zmianie kompilatora lub opcji optymalizacji albo zakończyć się błędem ochrony pamięci (`segmentation fault`, `access violation`).

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

Przykładem jest liczenie średniej `sum / n`, gdy obie zmienne są typu `int`. Wynik zostanie obcięty do liczby całkowitej, choć kompilator nie zgłosi żadnego zastrzeżenia.

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

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

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**. Duża liczba użytkowników ceni **Go** i **Rust**.

Analitycy często używają języków 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://survey.stackoverflow.co** – ankiety serwisu społecznościowego programistów i informatyków.

Zwykle podczas pracy programiści używają (zintegrowanych) **środowisk programistycznych** (*IDE*, *integrated development environment*). To program lub grupa programów (środowisko) służących do tworzenia, modyfikowania, testowania i konserwacji oprogramowania.

Środowiska programistyczne charakteryzują się tym, że udostępniają złożoną, wieloraką funkcjonalność obejmującą edycję kodu źródłowego, kompilowanie kodu źródłowego, tworzenie zasobów programu (szablonów, okien dialogowych, menu, raportów, elementów graficznych jak ikony, obrazy), tworzenie baz danych, komponentów i innych.

Środowisko programistyczne może występować jako osobny pakiet oprogramowania. Przykładami są:

- Visual Studio (Microsoft, Windows; C/C++, C#).
- Delphi (Embarcadero, Windows; Object Pascal) i C++ Builder (Embarcadero, Windows; C++)
- Eclipse i NetBeans (darmowe, Windows/Linux; Java).
- IntelliJ IDEA (JetBrains, Windows/Linux; Java).
- Anjuta (darmowe, Linux; C/C++, GTK).
- Code::Blocks (darmowe, Windows/Linux; C/C++).
- KDevelop (darmowe, Windows/Linux; C/C++, Qt).
- Lazarus (darmowe, Windows/Linux; Free Pascal, Qt, GTK).
- Qt Creator (darmowe, Windows/Linux; C/C++, Qt).

Środowisko programistyczne może być zintegrowane z systemem operacyjnym, jak w przypadku systemów z rodziny **Unix** lub **maszyn lispowych**.

Wartym wspomnienia jest też środowisko **MSYS2** – zbiór narzędzi i bibliotek znanych z systemu GNU/Linux zapewniających łatwe w użyciu środowisko do budowania, instalacji i uruchamiania natywnych aplikacji dla systemu operacyjnego Windows.

W skład wchodzą terminal wiersza poleceń **mintty**, **bash**, systemy kontroli wersji takie jak **git** i **subversion**, narzędzia takie jak **sed**, **awk**, **grep** czy system kompilacji **autotools**.

Pomimo że niektóre z narzędzi oparte są na bibliotece **Cygwin**, głównym celem MSYS2 jest zapewnienie środowiska budowy dla natywnego oprogramowania Windows i utrzymywanie Cygwina na minimalnym poziomie.

MSYS2 zapewnia aktualne natywne kompilacje m.in.: **GCC**, **mingw-w64**, **CPython**, **CMake**, **Meson**, **OpenSSL**, **FFmpeg**, **Rust**, **Ruby**.

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

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).

Do tej rodziny należy też **programowanie funkcyjne**, którego elementy przejęła większość współczesnych języków (np. lambdy). Przykłady języków: **SQL**, **XSLT**, **Prolog**, **Haskell**, **Lisp**.

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 budowanie programu z trzech struktur sterujących: **sekwencji**, **wyboru** (instrukcje warunkowe) i **iteracji** (pętle), hierarchicznie w siebie zagnieżdżanych. Każdy blok ma jeden punkt wejścia i jeden punkt wyjścia.

Chodzi przede wszystkim o rezygnację z instrukcji skoku `goto`. Wykazano (Böhm, Jacopini, 1966), że każdy algorytm da się zapisać za pomocą samych tych trzech struktur.

Ścisłą strukturalność zakłócają instrukcje opuszczające blok przed jego końcem: `break`, `continue`, `return` w środku funkcji, a także przechodzenie między etykietami `case` instrukcji `switch` bez `break`. W niektórych przypadkach znacząco podnoszą one jednak 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. Tak przynajmniej twierdzili jego propagatorzy w latach 90. 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 kolejność zdarzeń, a więc i przepływ sterowania w programie, nie jest znana z góry.

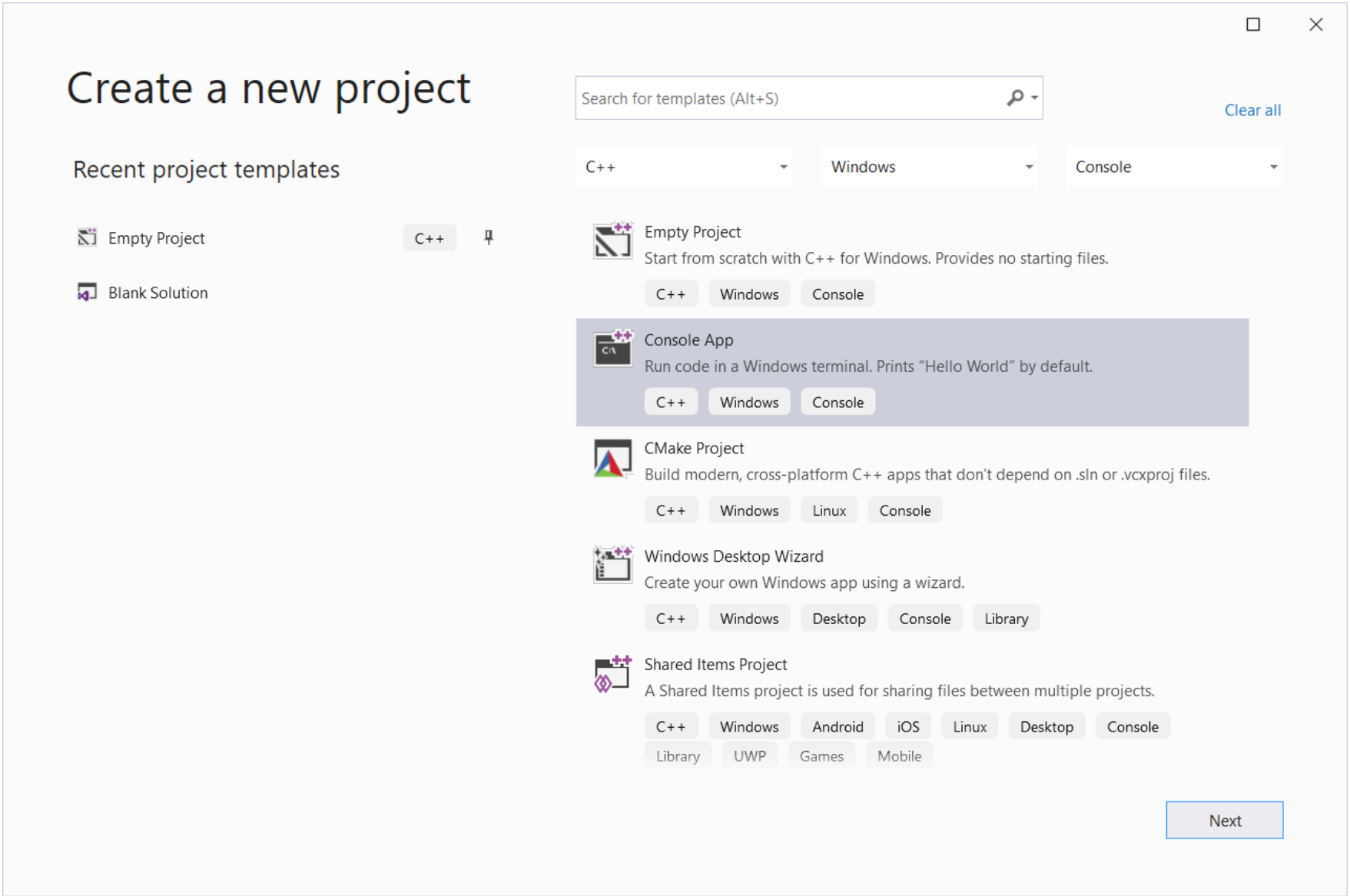
Programowanie zdarzeniowe jest dominującym typem programowania związanego z **graficznym interfejsem użytkownika** (*Graphical User Interface*) – zdarzenia to naciśnięcia myszy, klawiszy, żądania odświeżenia przez system okienkowy, różne zdarzenia sieciowe i inne.

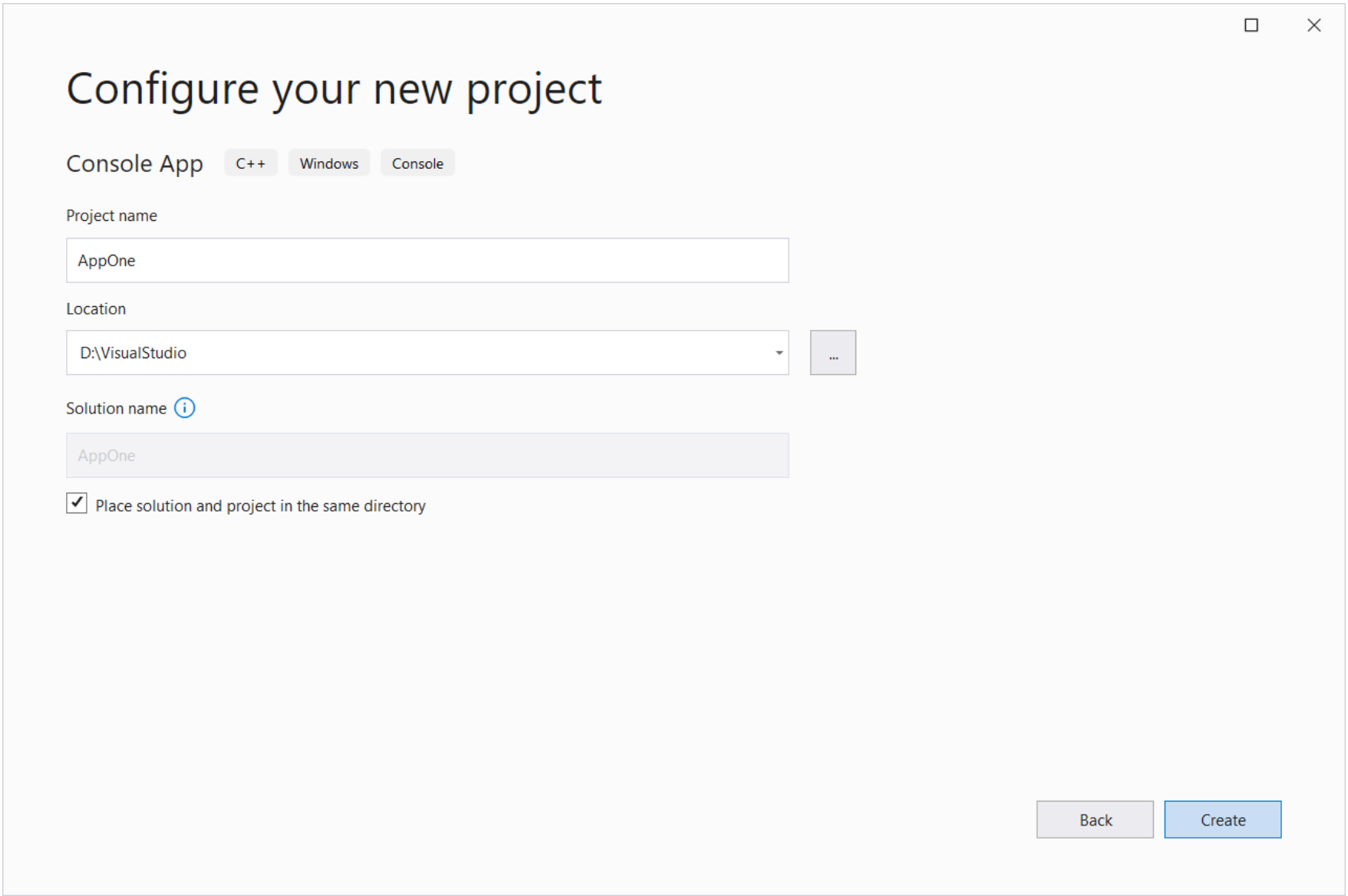
W programowaniu zdarzeniowym ważne jest, żeby nie obsługiwać zbyt długo danego zdarzenia, bo blokuje się w ten sposób obsługę innych. W przypadku serwerów obniżyłoby to znacznie wydajność, w przypadku GUI program zbyt wolno odpowiadałby na akcje użytkownika.

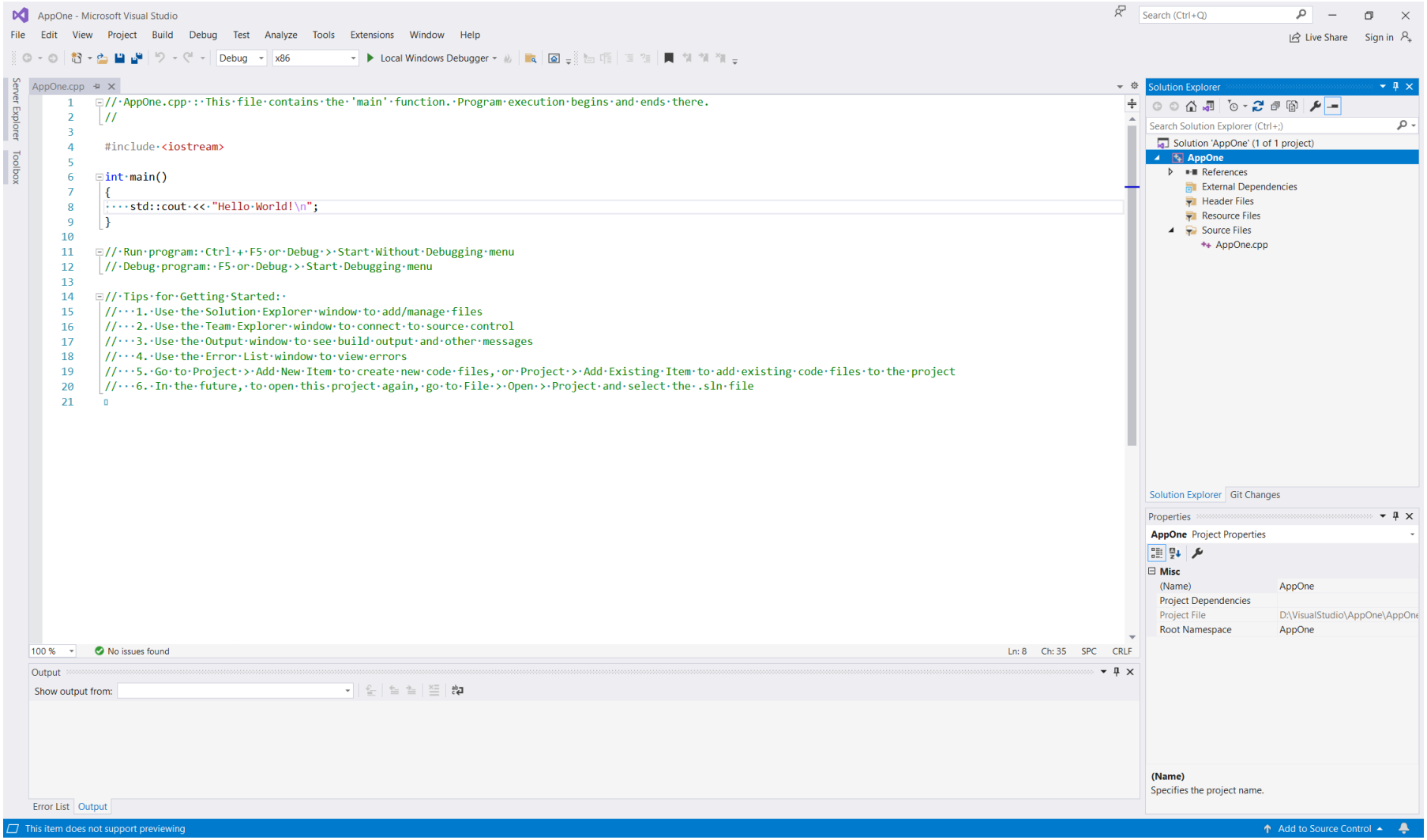
Można to osiągnąć za pomocą asynchronicznego I/O, wielowątkowości, rozbijania zdarzenia na podzdarzenia i wielu innych mechanizmów.

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

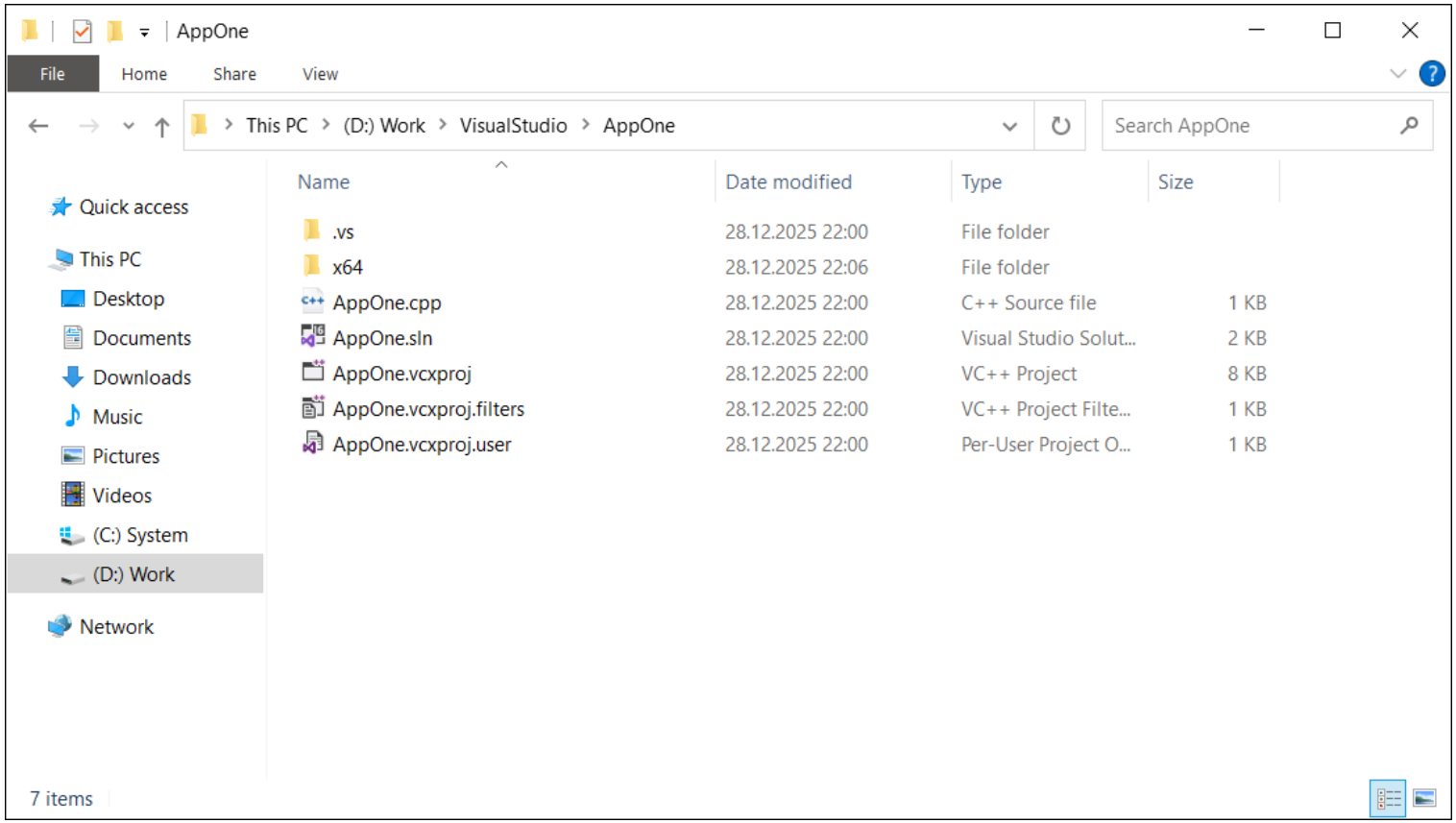
Środowisko programowania Visual Studio – Windows

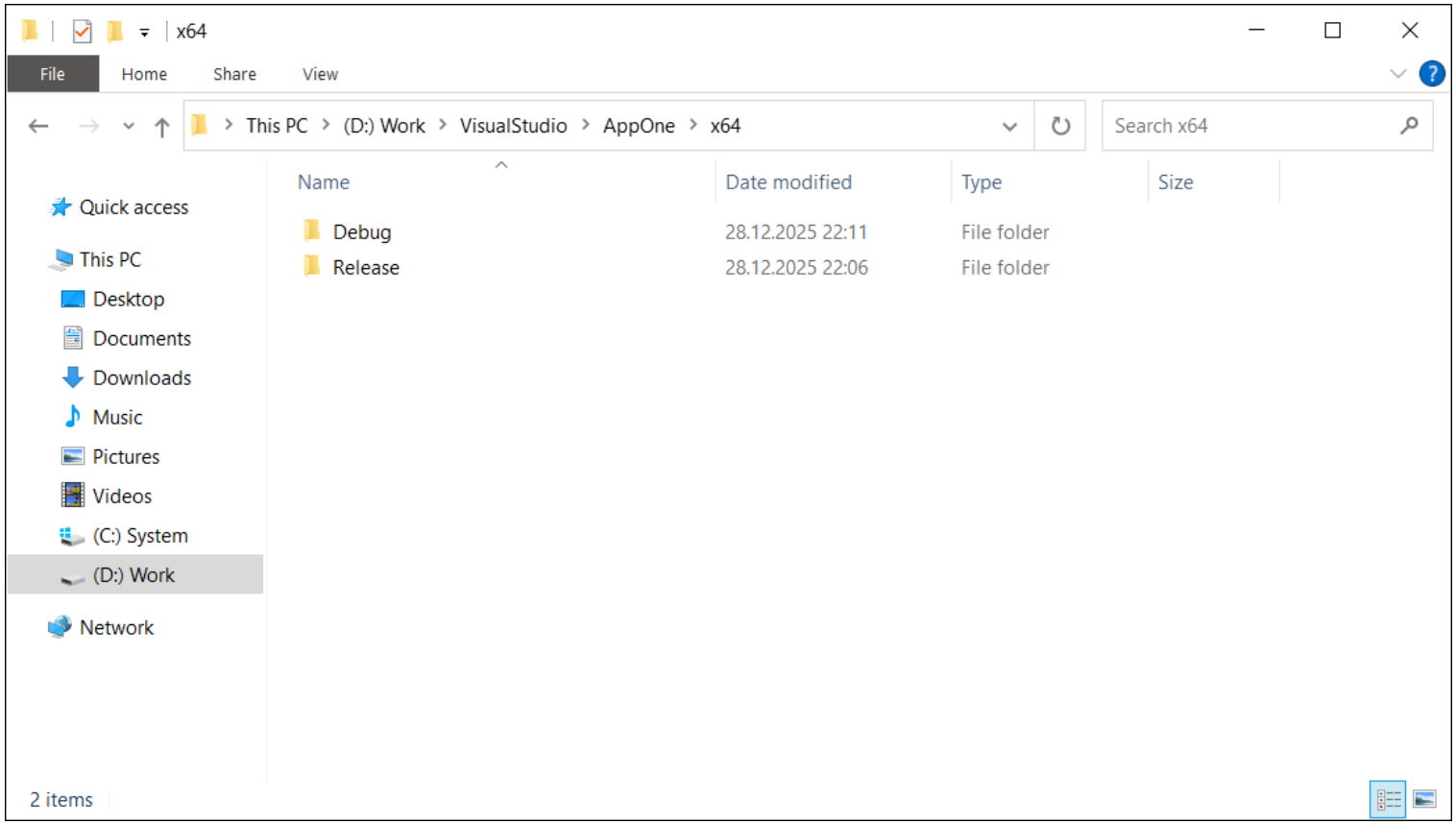


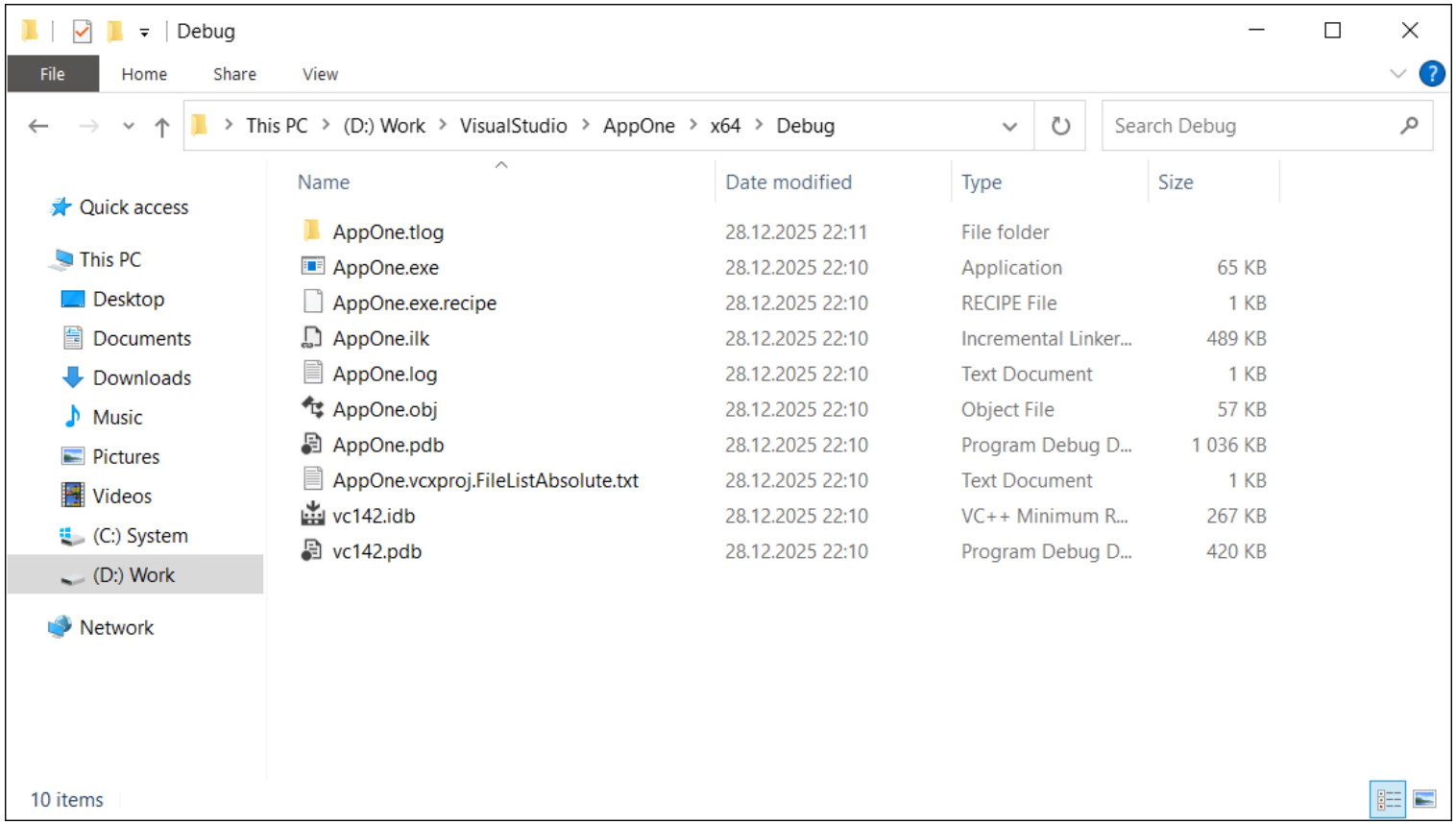

Configure your new project
Console App
C++
Windows
Console
Project name
AppOne
Location
D:\VisualStudio
...
Solution name
AppOne
Place solution and project in the same directory
Back
Create


AppOne - Microsoft Visual Studio
File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Live Share
Sign in
Debug
x86
Local Windows Debugger
AppOne.cpp
//·AppOne.cpp·:·This·file·contains·the·'main'·function.·Program·execution·begins·and·ends·there.
//
#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·to·the·project
//···6.·In·the·future,·to·open·this·project·again,·go·to·File·>·Open·>·Project·and·select·the·.sln·file
100 %
No issues found
Ln: 8 Ch: 35 SPC CRLF
Output
Show output from:
Error List
Output
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'AppOne' (1 of 1 project)
AppOne
References
External Dependencies
Header Files
Resource Files
Source Files
AppOne.cpp
Solution Explorer
Git Changes
Properties
AppOne Project Properties
Misc
(Name) AppOne
Project Dependencies
Project File D:\VisualStudio\AppOne\AppOne
Root Namespace AppOne
(Name)
Specifies the project name.
This item does not support previewing
Add to Source Control

Pliki projektu

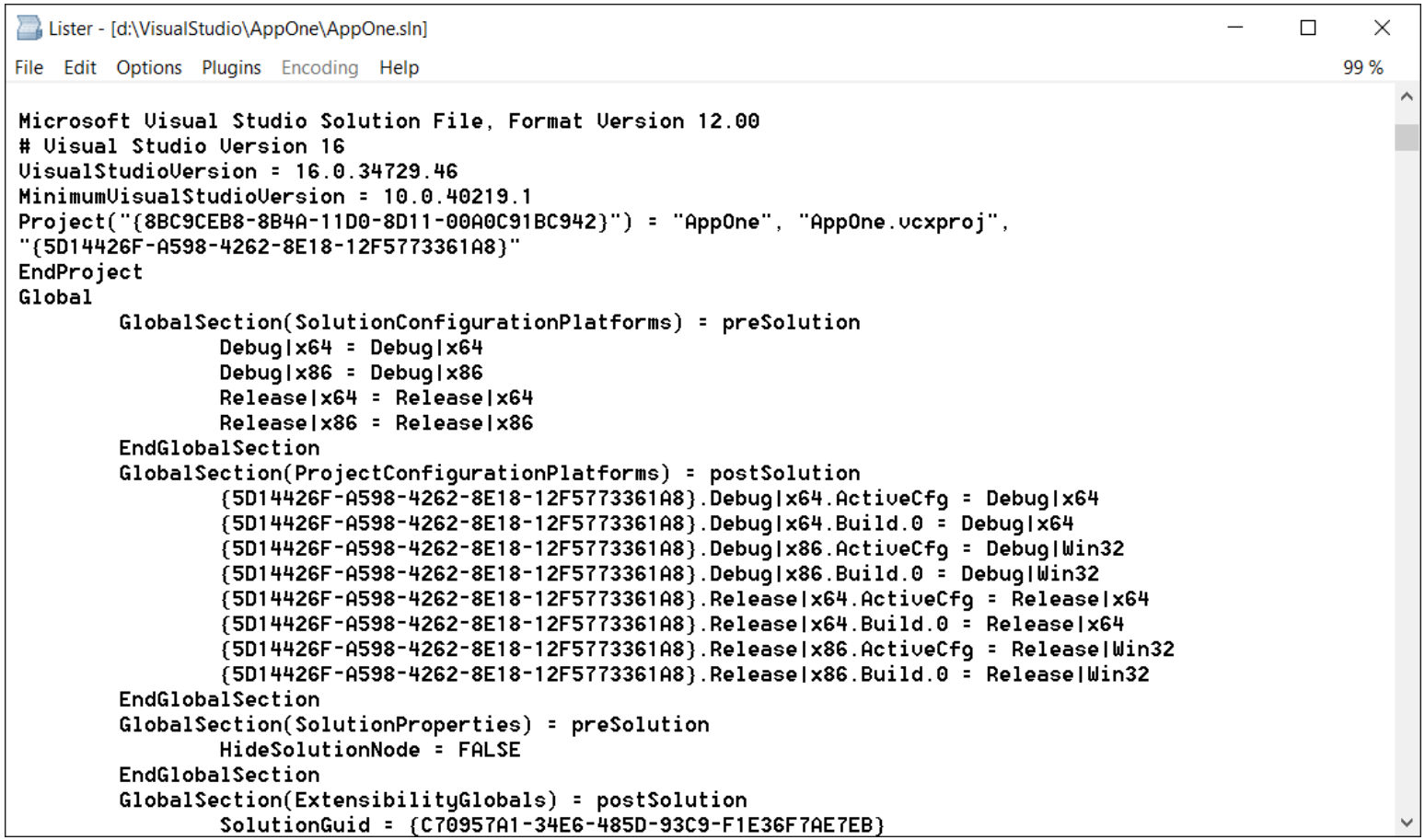



x64
File
Home
Share
View
This PC
(D:) Work
VisualStudio
AppOne
x64
Search x64
Quick access
This PC
Desktop
Documents
Downloads
Music
Pictures
Videos
(C:) System
(D:) Work
Network
Name
Date modified
Type
Size
Debug
28.12.2025 22:11
File folder
Release
28.12.2025 22:06
File folder
2 items


Debug
File
Home
Share
View
This PC
(D:) Work
VisualStudio
AppOne
x64
Debug
Search Debug
Quick access
This PC
Desktop
Documents
Downloads
Music
Pictures
Videos
(C:) System
(D:) Work
Network
Name
Date modified
Type
Size
AppOne.tlog 28.12.2025 22:11 File folder
AppOne.exe 28.12.2025 22:10 Application 65 KB
AppOne.exe.recipe 28.12.2025 22:10 RECIPE File 1 KB
AppOne.ilk 28.12.2025 22:10 Incremental Linker... 489 KB
AppOne.log 28.12.2025 22:10 Text Document 1 KB
AppOne.obj 28.12.2025 22:10 Object File 57 KB
AppOne.pdb 28.12.2025 22:10 Program Debug D... 1 036 KB
AppOne.vcxproj.FileListAbsolute.txt 28.12.2025 22:10 Text Document 1 KB
vc142.idb 28.12.2025 22:10 VC++ Minimum R... 267 KB
vc142.pdb 28.12.2025 22:10 Program Debug D... 420 KB
10 items

Struktura domyślnego projektu

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



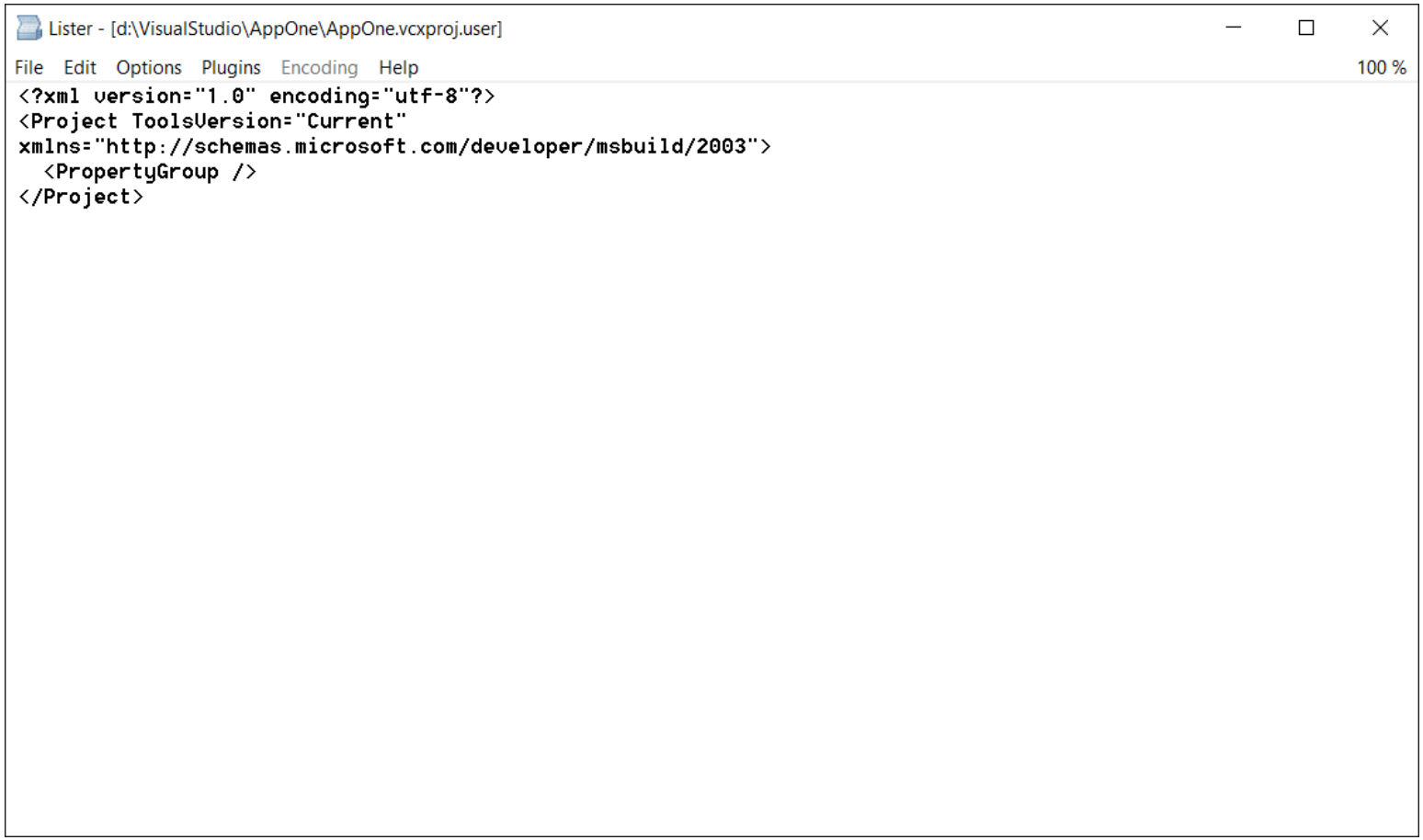
Plik `*.vcxproj` – plik *projektu*, przechowuje informacje specyficzne dla każdego projektu.

Lister - [d:\VisualStudio\AppOne\AppOne.vcxproj]

File Edit Options Plugins Encoding Help 18 %

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build"
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>16.0</VCProjectVersion>
    <Keyword>Win32Proj</Keyword>
    <ProjectGuid>{5d14426f-a598-4262-8e18-12f5773361a8}</ProjectGuid>
    <RootNamespace>AppOne</RootNamespace>
    <WindowsTargetPlatformVersion>10.0</WindowsTargetPlatformVersion>
  </PropertyGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'"
```

Plik `*.vcxproj.user` – plik *ustawień użytkownika*, przechowuje ustawienia użytkownika (np. debugowania).



W projektach starszych wersji środowiska mogą się jeszcze znajdować pliki `pch.cpp` i `pch.h` – pliki *prekompilowanych nagłówków*.

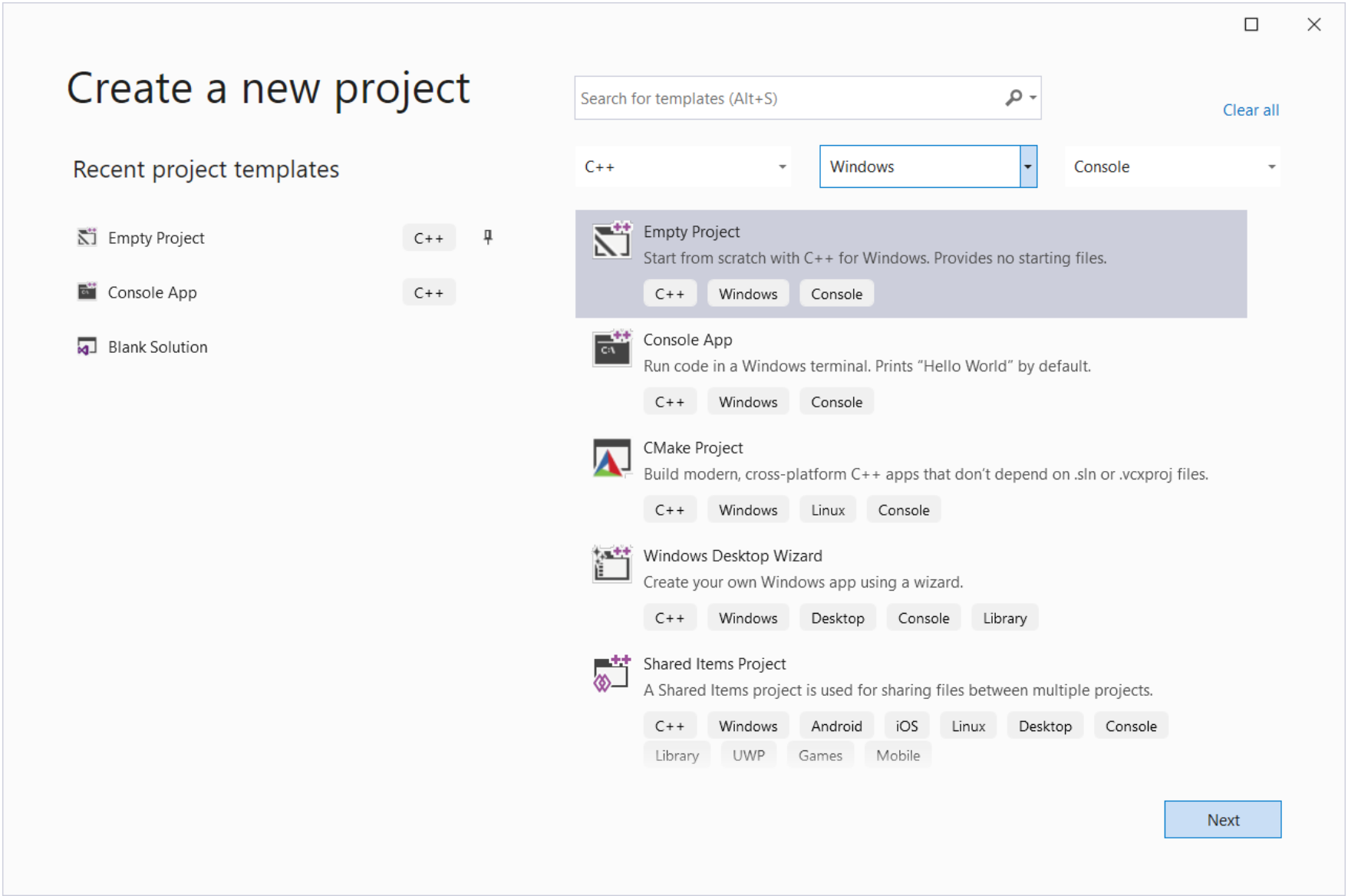
Plik `*.cpp` – plik *źródłowy*, zawiera kod źródłowy programu z funkcją `main()`.

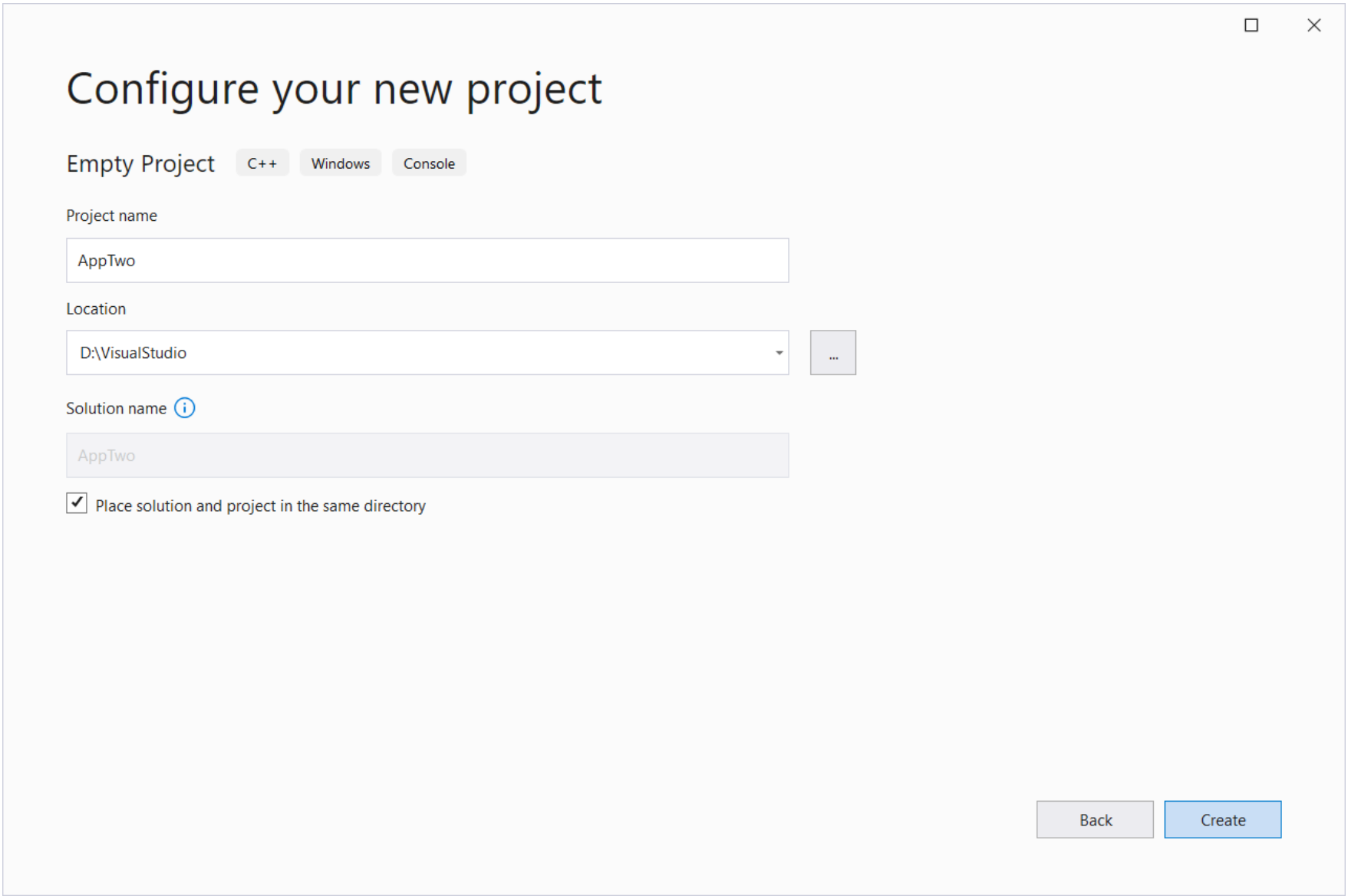
Lister - [d:\VisualStudio\AppOne\AppOne.cpp]

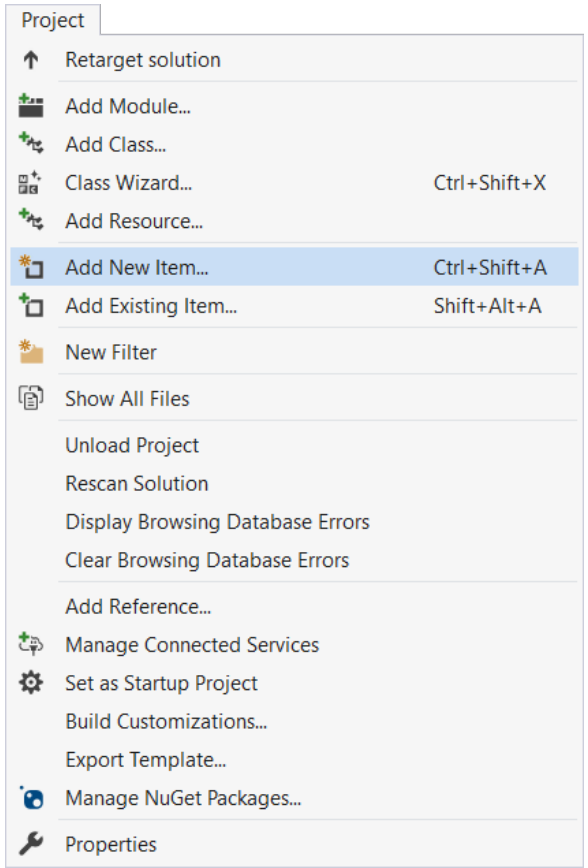
File Edit Options Plugins Encoding Help 100 %

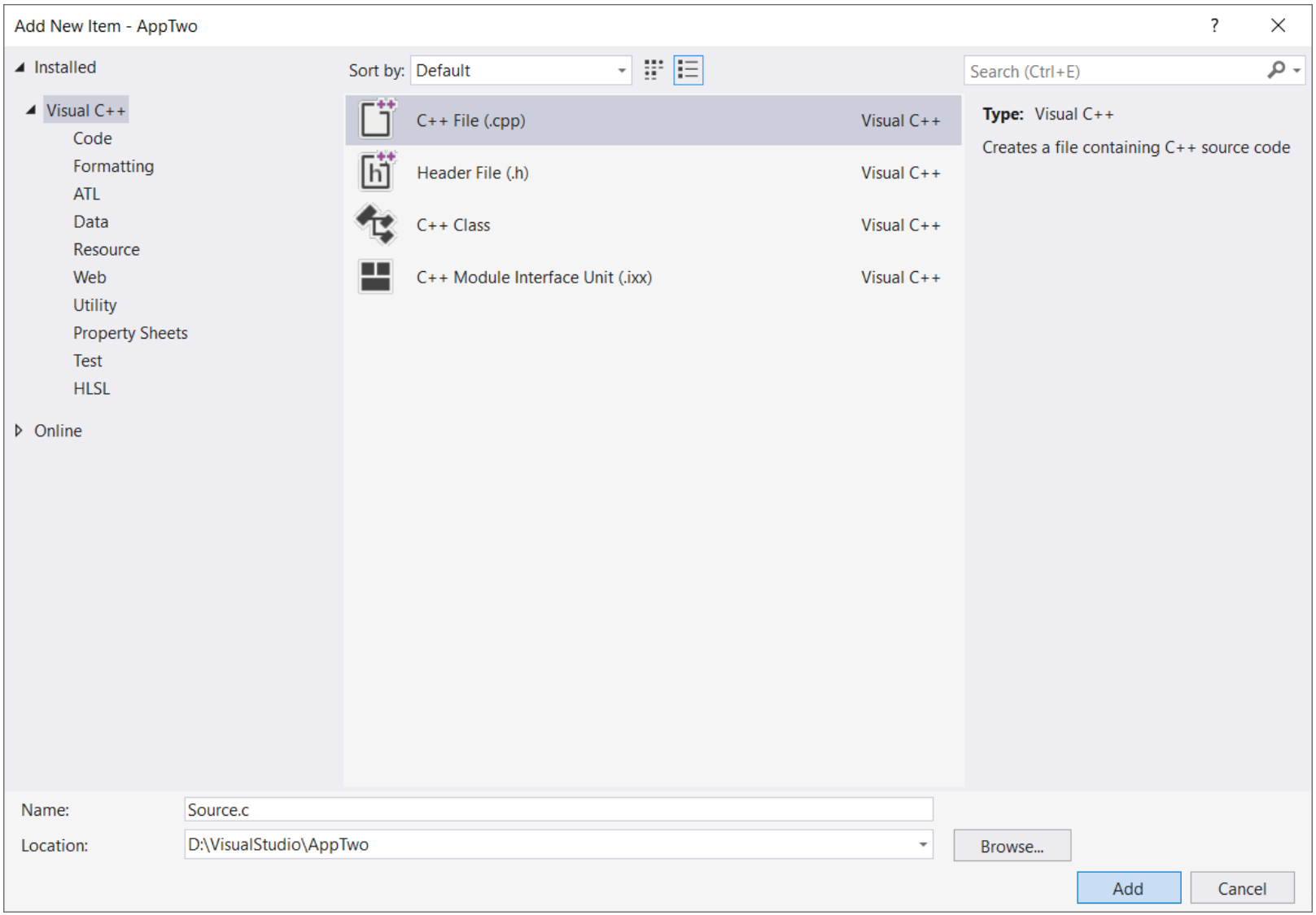
```
// AppOne.cpp : This file contains the 'main' function. Program execution begins
and ends there.
//

#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 to the project
//   6. In the future, to open this project again, go to File > Open > Project
and select the .sln file
```

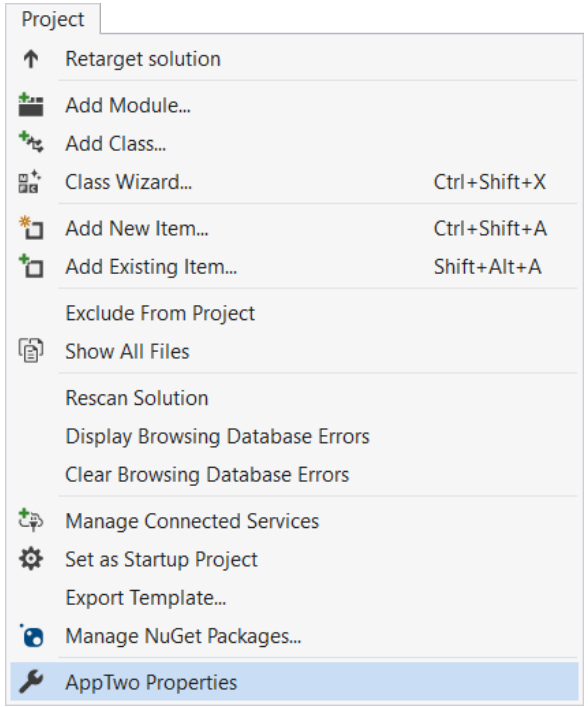
Projekt w języku C

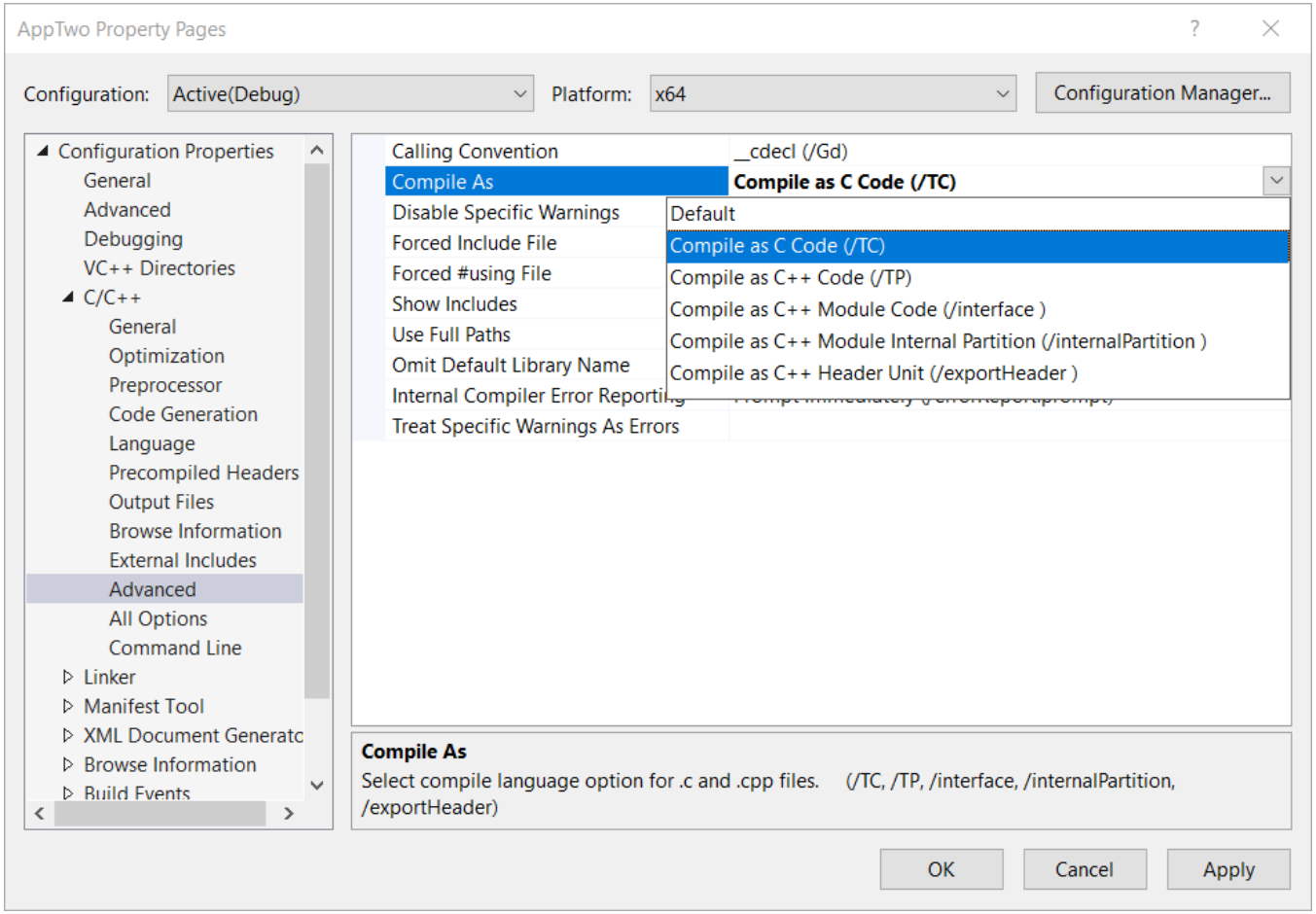


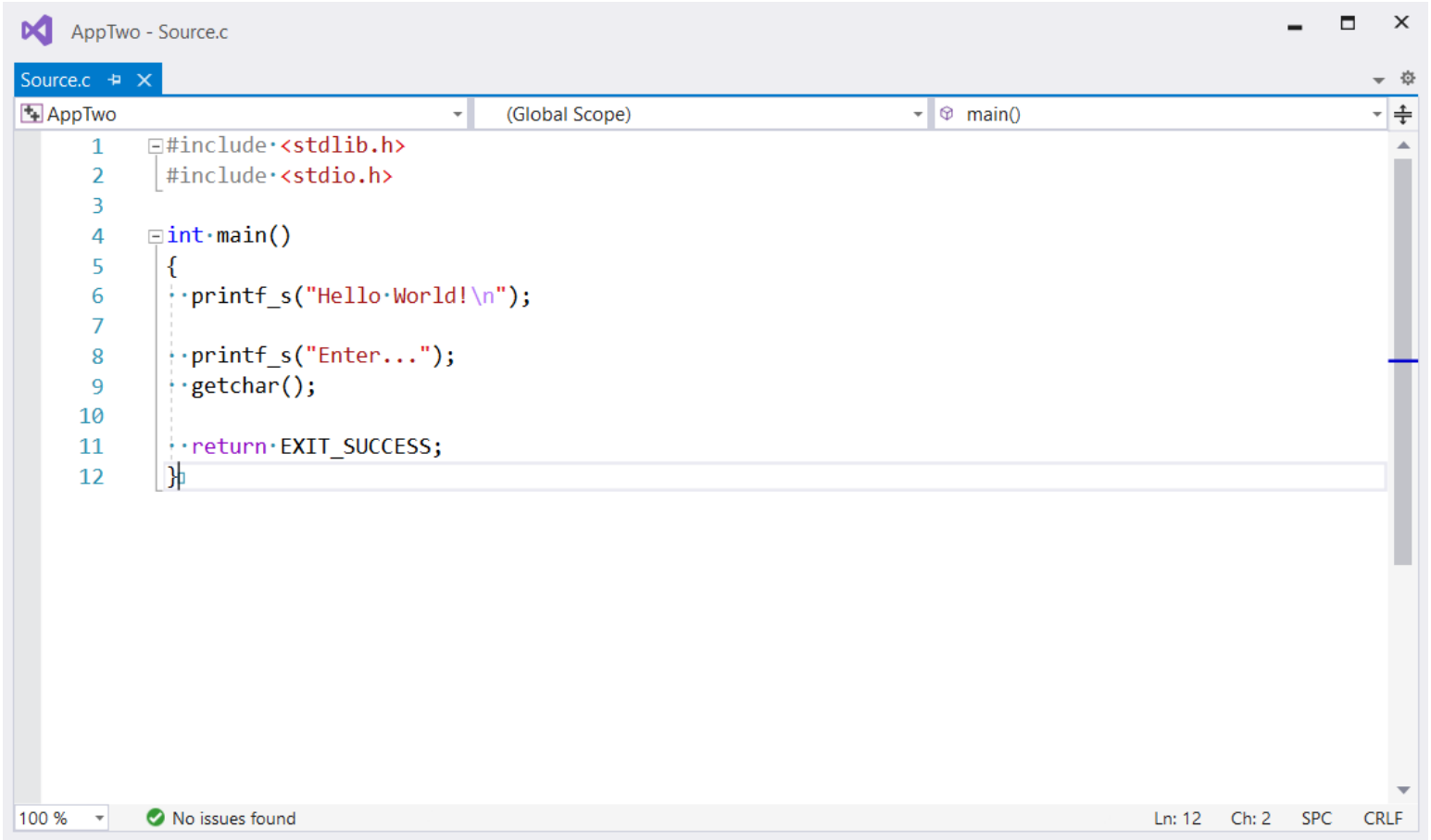

Configure your new project
Empty Project
C++
Windows
Console
Project name
AppTwo
Location
D:\VisualStudio
...
Solution name
AppTwo
Place solution and project in the same directory
Back
Create


Project
Retarget solution
Add Module...
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...
Manage Connected Services
Set as Startup Project
Build Customizations...
Export Template...
Manage NuGet Packages...
Properties


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

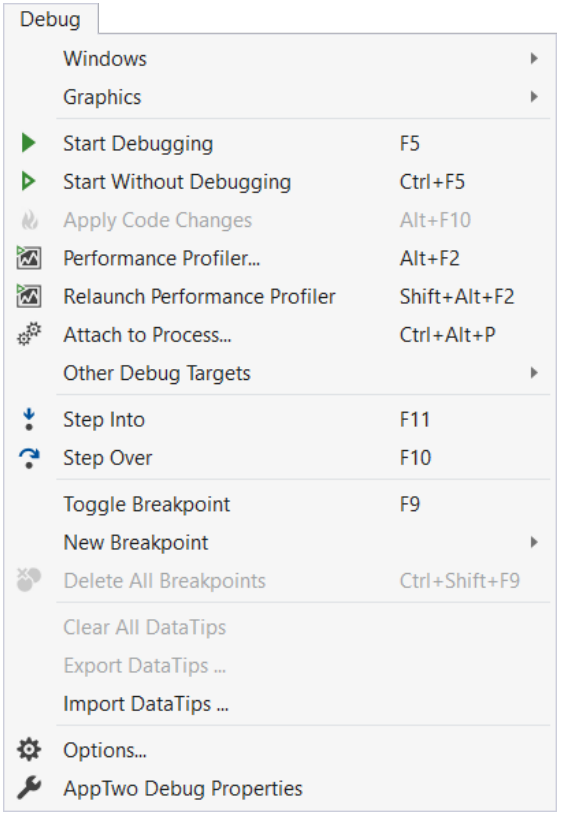

Project
Retarget solution
Add Module...
Add Class...
Class Wizard... Ctrl+Shift+X
Add New Item... Ctrl+Shift+A
Add Existing Item... Shift+Alt+A
Exclude From Project
Show All Files
Rescan Solution
Display Browsing Database Errors
Clear Browsing Database Errors
Manage Connected Services
Set as Startup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

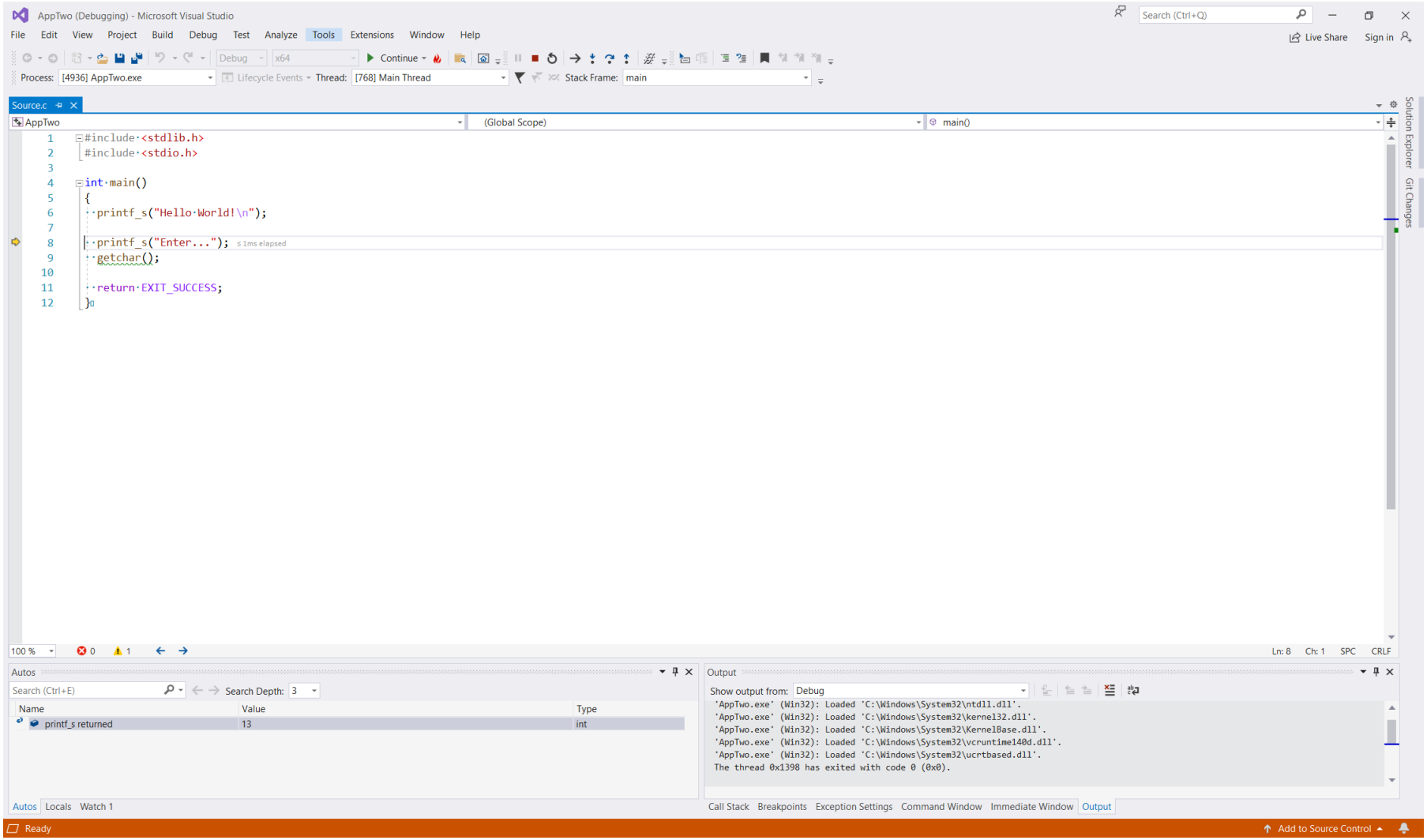

AppTwo Property Pages
Configuration: Active(Debug)
Platform: x64
Configuration Manager...
Configuration Properties
General
Advanced
Debugging
VC++ Directories
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
External Includes
Advanced
All Options
Command Line
Linker
Manifest Tool
XML Document Generatc
Browse Information
Build Events
Calling Convention
__cdecl (/Gd)
Compile As
Compile as C Code (/TC)
Disable Specific Warnings
Forced Include File
Forced #using File
Show Includes
Use Full Paths
Omit Default Library Name
Internal Compiler Error Reporting
Treat Specific Warnings As Errors
Default
Compile as C Code (/TC)
Compile as C++ Code (/TP)
Compile as C++ Module Code (/interface)
Compile as C++ Module Internal Partition (/internalPartition)
Compile as C++ Header Unit (/exportHeader)
Compile As
Select compile language option for .c and .cpp files. (/TC, /TP, /interface, /internalPartition, /exportHeader)
OK
Cancel
Apply

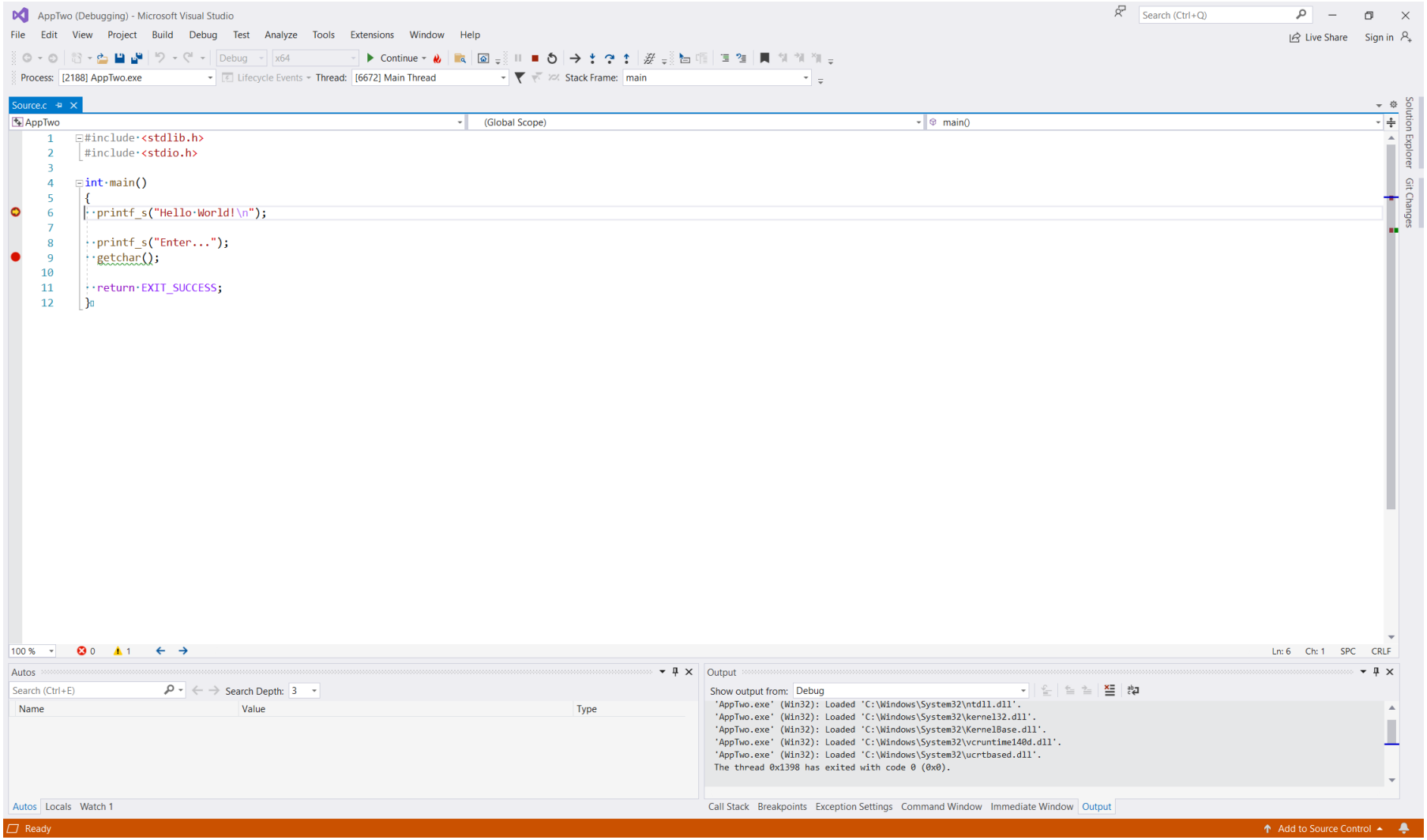

AppTwo - Source.c
Source.c
AppTwo
(Global Scope)
main()
#include <stdlib.h>
#include <stdio.h>
int main()
{
printf_s("Hello World!\n");
printf_s("Enter...");
getchar();
return EXIT_SUCCESS;
}
100 %
No issues found
Ln: 12
Ch: 2
SPC
CRLF

Kompilowanie, debugowanie, śledzenie

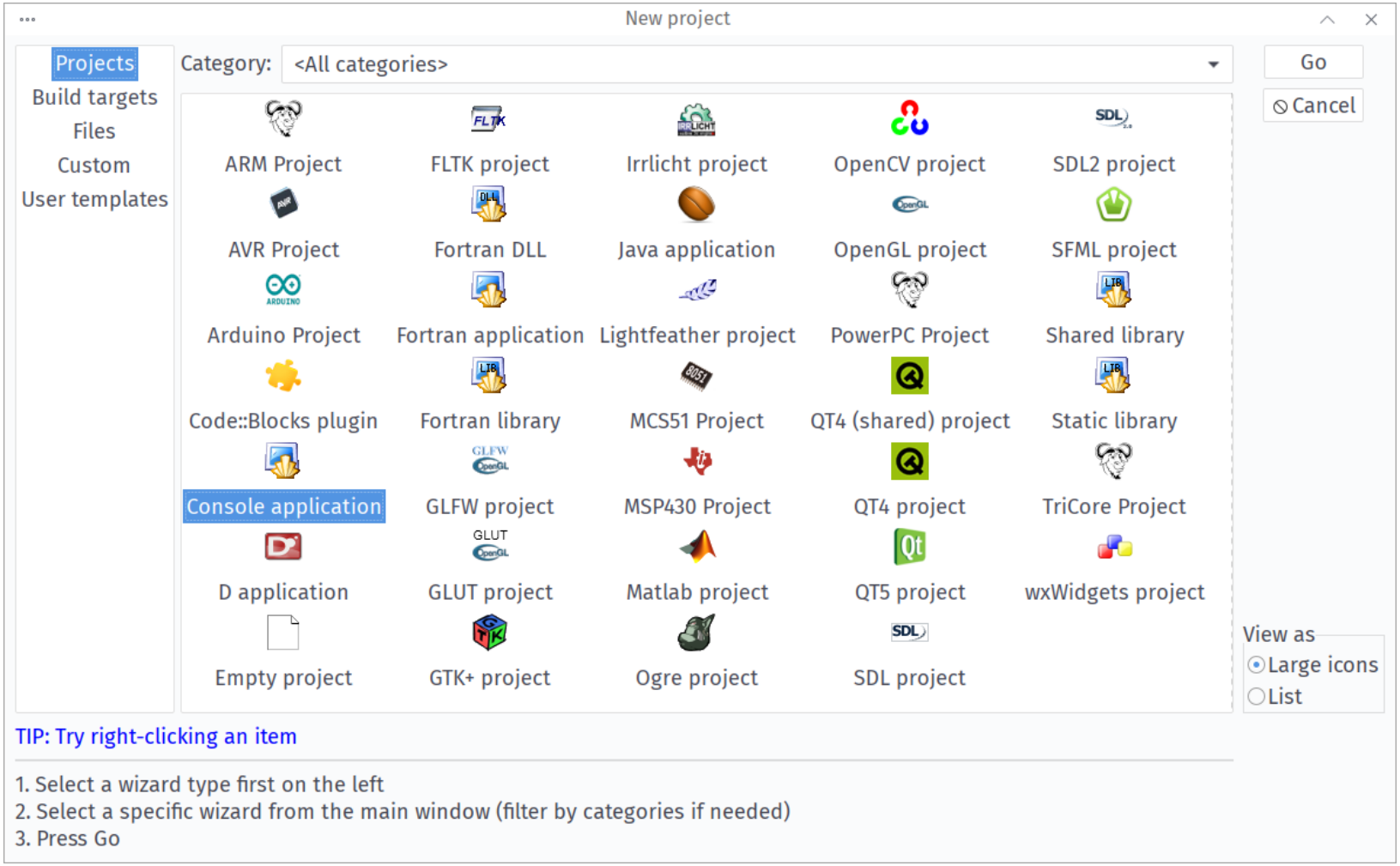
skróty klawiaturowe	
`[F5]`	start z nadzorowaniem (debugger) i kontynuacja wykonywania programu
`[CTRL+F5]`	start bez debuggera
`[F11]`	krokowe wykonywanie programu
`[F10]`	krokowe wykonywanie programu, procedury i funkcje w jednym kroku
`[F9]`	wstawianie i usuwanie punktów przerwania (*breakpoint*)

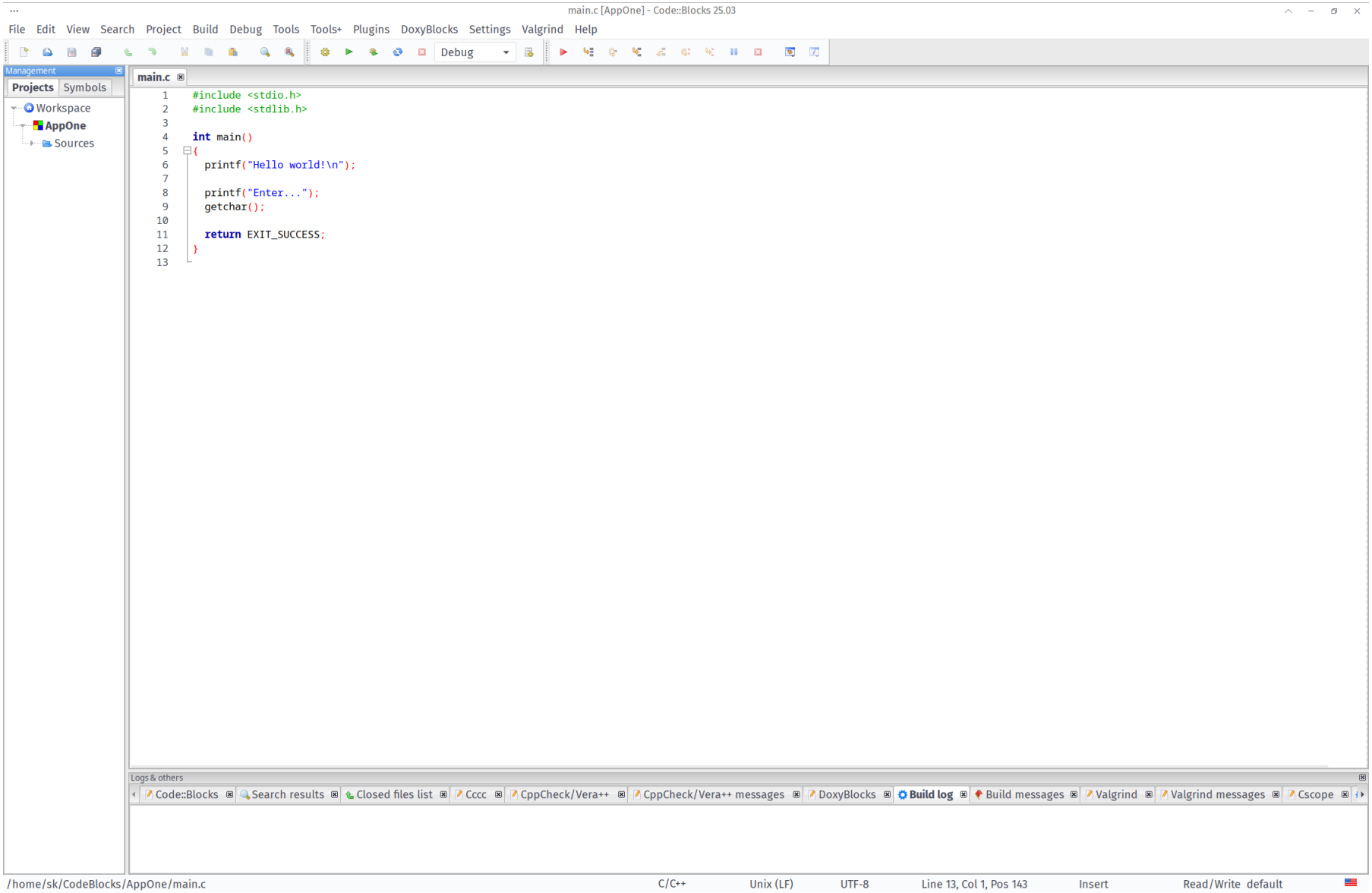



AppTwo (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Live Share
Sign in
Debug
x64
Continue
Process: [4936] AppTwo.exe
Lifecycle Events
Thread: [768] Main Thread
Stack Frame: main
Source.c
AppTwo
(Global Scope)
main()
#include <stdlib.h>
#include <stdio.h>
int main()
{
printf_s("Hello World!\n");
printf_s("Enter...");
≤ 1ms elapsed
getchar();
return EXIT_SUCCESS;
}
Solution Explorer
Git Changes
100 %
0
1
Ln: 8
Ch: 1
SPC
CRLF
Autos
Search (Ctrl+E)
Search Depth: 3
Name
Value
Type
printf_s returned
13
int
Autos
Locals
Watch 1
Output
Show output from: Debug
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\ntdll.dll'.
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\kernel32.dll'.
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\KernelBase.dll'.
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140d.dll'.
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbased.dll'.
The thread 0x1398 has exited with code 0 (0x0).
Call Stack
Breakpoints
Exception Settings
Command Window
Immediate Window
Output
Ready
Add to Source Control

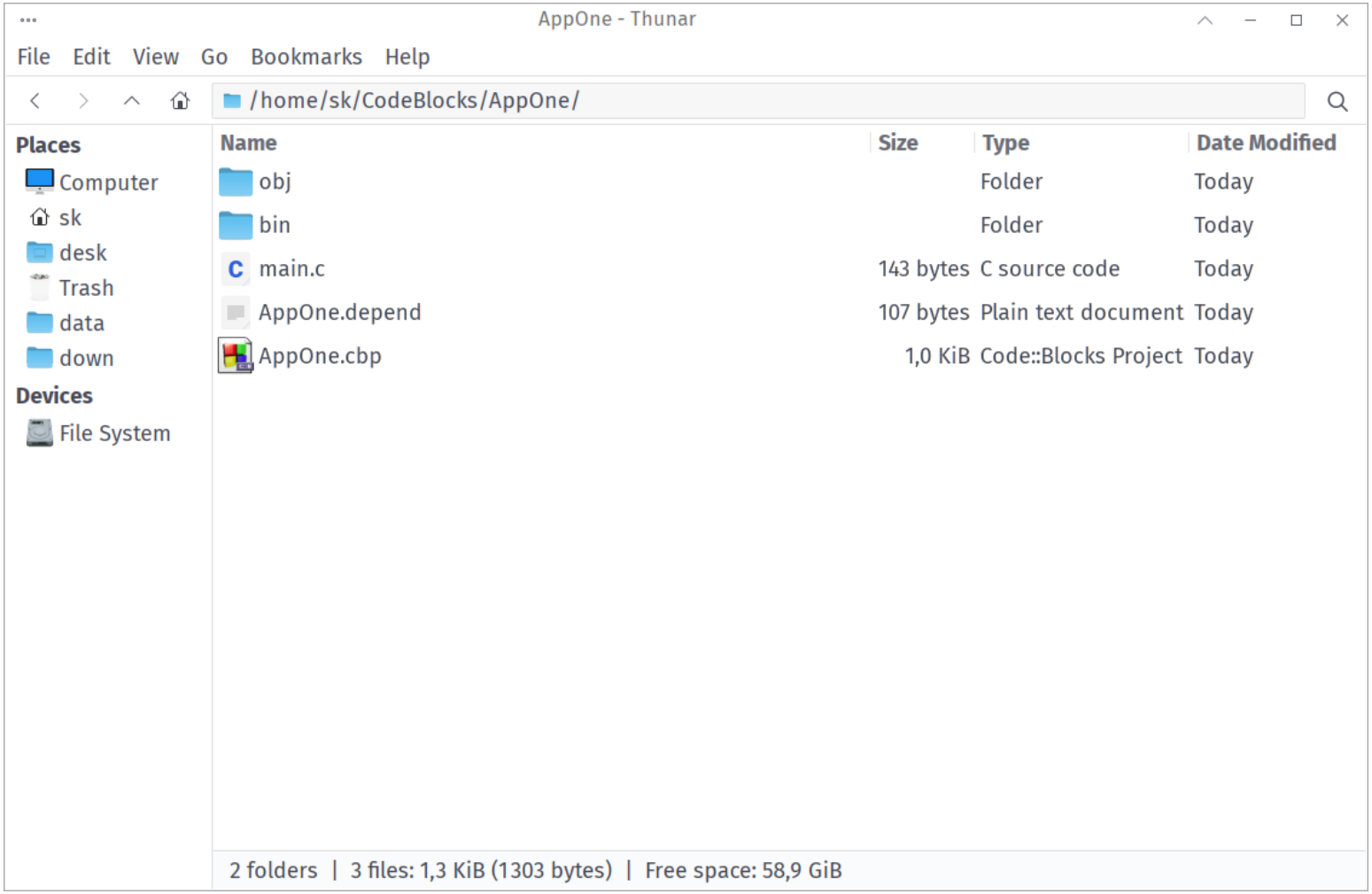

AppTwo (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Live Share
Sign in
Debug
x64
Continue
Process: [2188] AppTwo.exe
Lifecycle Events
Thread: [6672] Main Thread
Stack Frame: main
Source.c
AppTwo
(Global Scope)
main()
#include <stdlib.h>
#include <stdio.h>
int main()
{
printf_s("Hello World!\n");
printf_s("Enter...");
getchar();
return EXIT_SUCCESS;
}
100 %
0
1
Ln: 6 Ch: 1 SPC CRLF
Autos
Search (Ctrl+E)
Search Depth: 3
Name
Value
Type
Autos Locals Watch 1
Output
Show output from: Debug
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\ntdll.dll'.
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\kernel32.dll'.
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\KernelBase.dll'.
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140d.dll'.
'AppTwo.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbased.dll'.
The thread 0x1398 has exited with code 0 (0x0).
Call Stack Breakpoints Exception Settings Command Window Immediate Window Output
Ready
Add to Source Control
Solution Explorer
Git Changes

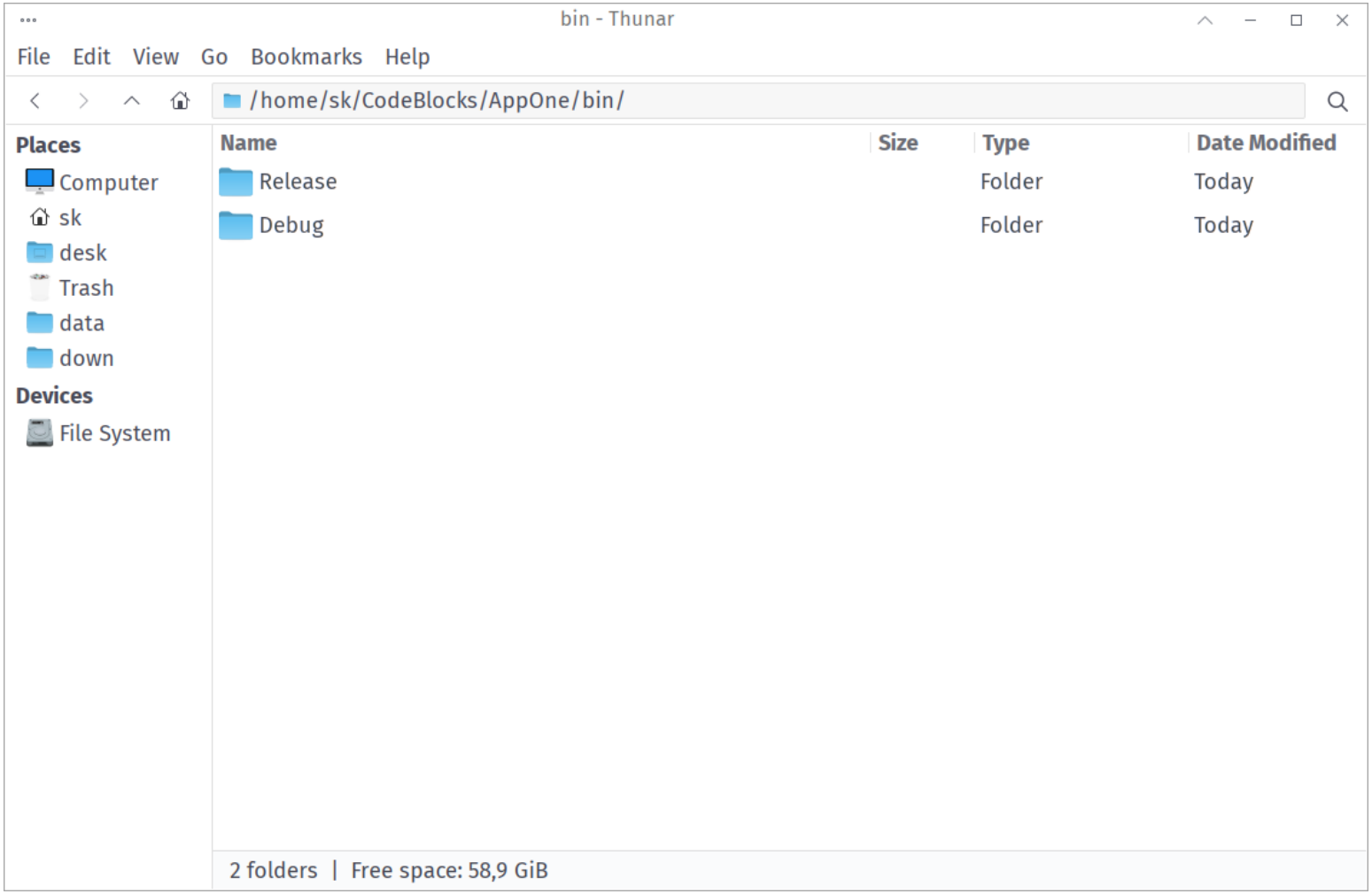
Środowisko programowania Code::Blocks – Linux

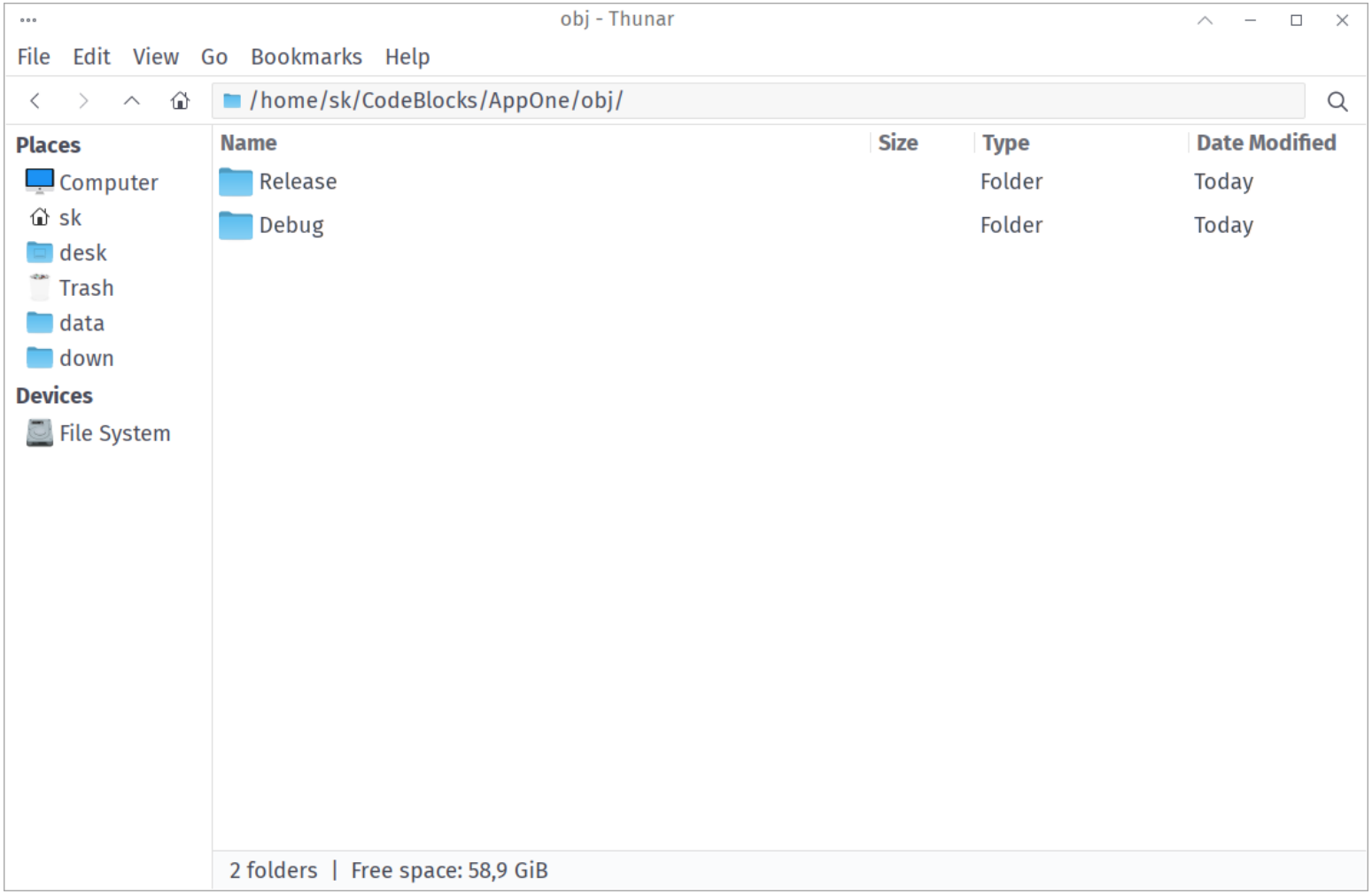


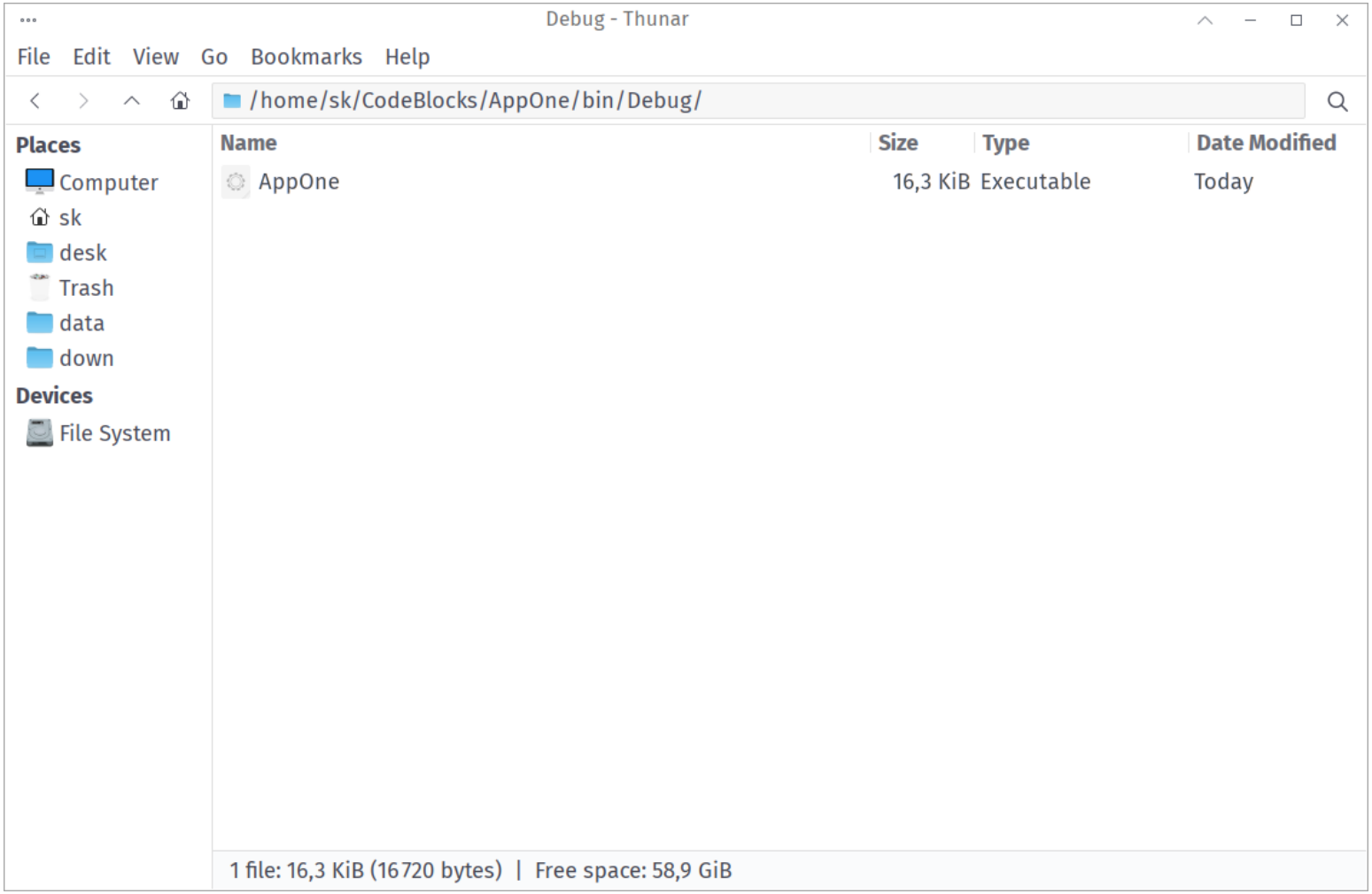

main.c [AppOne] - Code::Blocks 25.03
File Edit View Search Project Build Debug Tools Tools+ Plugins DoxyBlocks Settings Valgrind Help
Debug
Management
Projects Symbols
Workspace
AppOne
Sources
main.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("Hello world!\n");
printf("Enter...");
getchar();
return EXIT_SUCCESS;
}
Logs & others
Code::Blocks Search results Closed files list Cccc CppCheck/Vera++ CppCheck/Vera++ messages DoxyBlocks Build log Build messages Valgrind Valgrind messages Cscope
/home/sk/CodeBlocks/AppOne/main.c
C/C++ Unix (LF) UTF-8 Line 13, Col 1, Pos 143 Insert Read/Write default

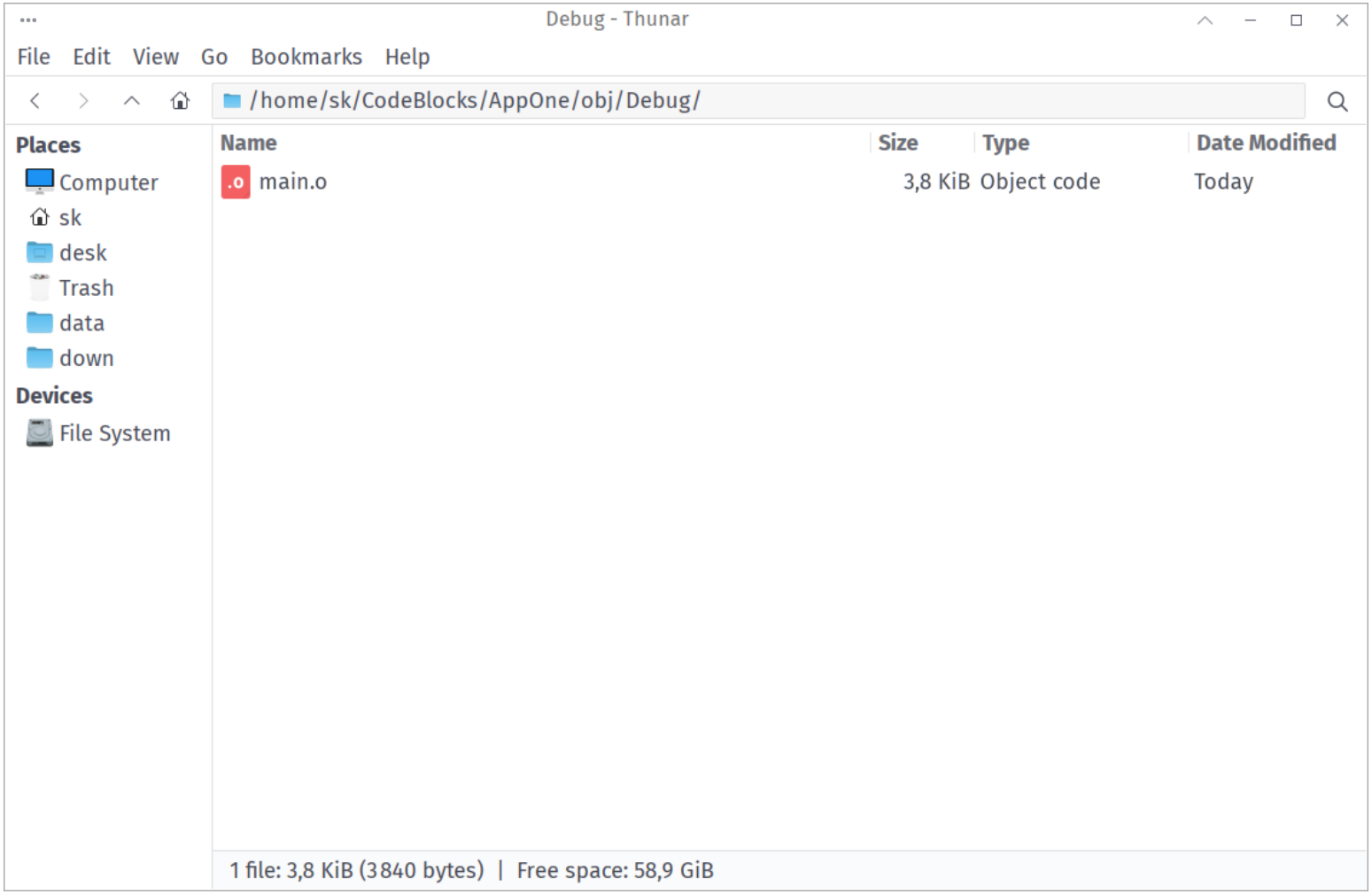
Pliki projektu




bin - Thunar
File Edit View Go Bookmarks Help
/home/sk/CodeBlocks/AppOne/bin/
Places
Computer
sk
desk
Trash
data
down
Devices
File System
Name
Size
Type
Date Modified
Release
Folder
Today
Debug
Folder
Today
2 folders | Free space: 58,9 GiB

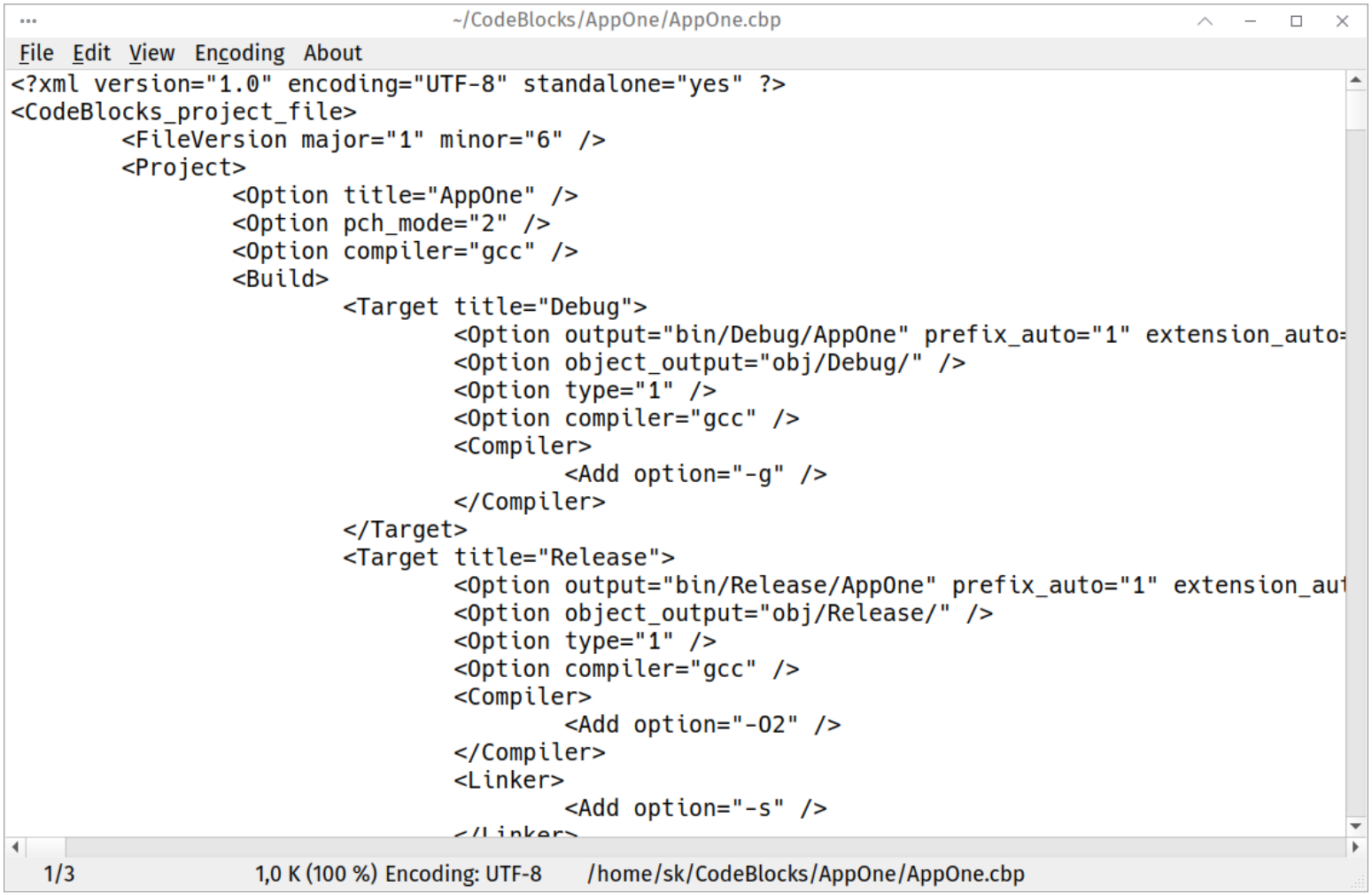

obj - Thunar
File Edit View Go Bookmarks Help
/home/sk/CodeBlocks/AppOne/obj/
Places
Computer
sk
desk
Trash
data
down
Devices
File System
Name
Size
Type
Date Modified
Release
Folder
Today
Debug
Folder
Today
2 folders | Free space: 58,9 GiB


Debug - Thunar
File Edit View Go Bookmarks Help
/home/sk/CodeBlocks/AppOne/bin/Debug/
Places
Computer
sk
desk
Trash
data
down
Devices
File System
Name
Size
Type
Date Modified
AppOne
16,3 KiB
Executable
Today
1 file: 16,3 KiB (16720 bytes) | Free space: 58,9 GiB

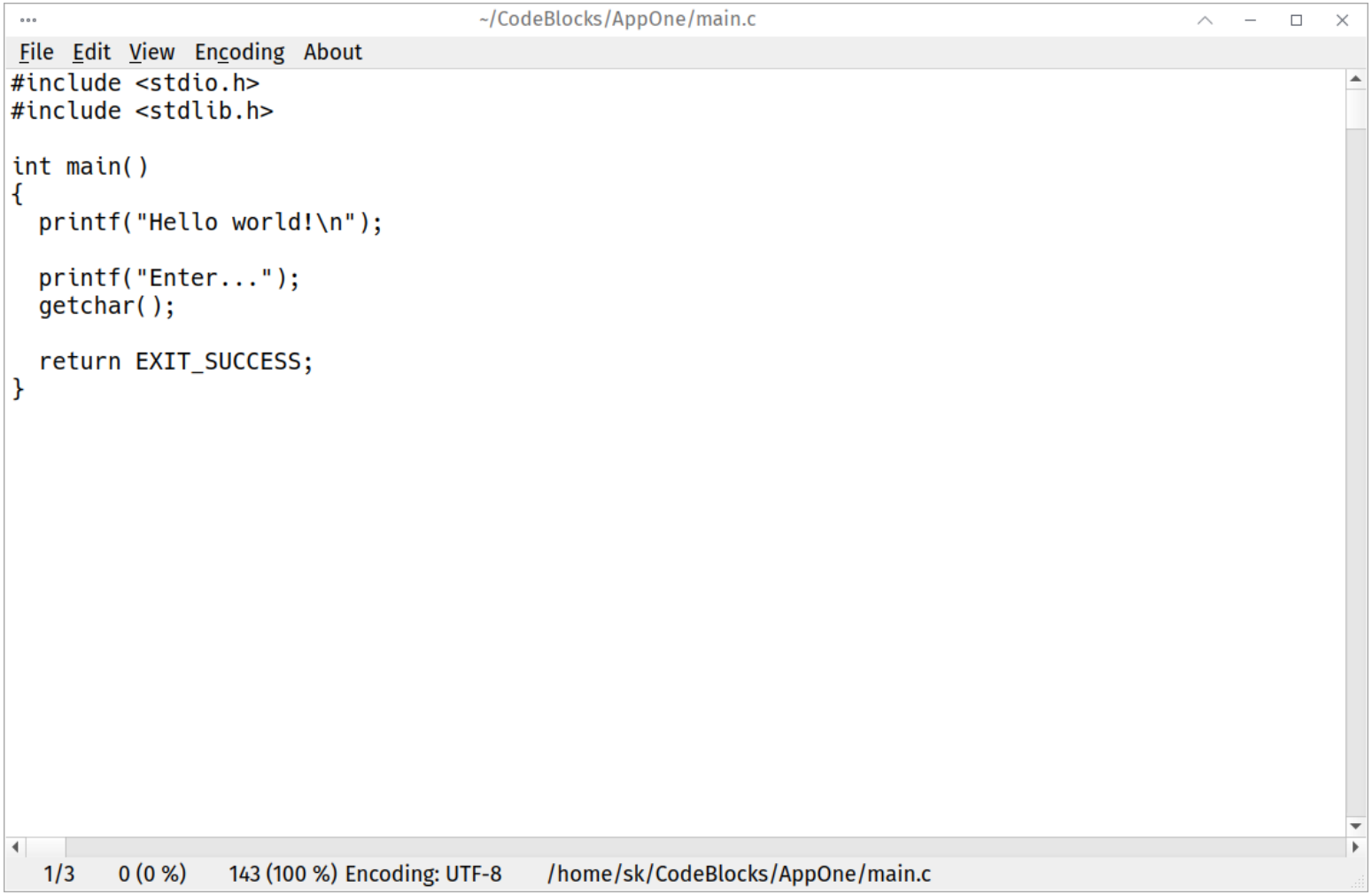

Debug - Thunar
File Edit View Go Bookmarks Help
/home/sk/CodeBlocks/AppOne/obj/Debug/
Places
Computer
sk
desk
Trash
data
down
Devices
File System
Name
Size
Type
Date Modified
main.o
3,8 KiB
Object code
Today
1 file: 3,8 KiB (3840 bytes) | Free space: 58,9 GiB

Struktura domyślnego projektu

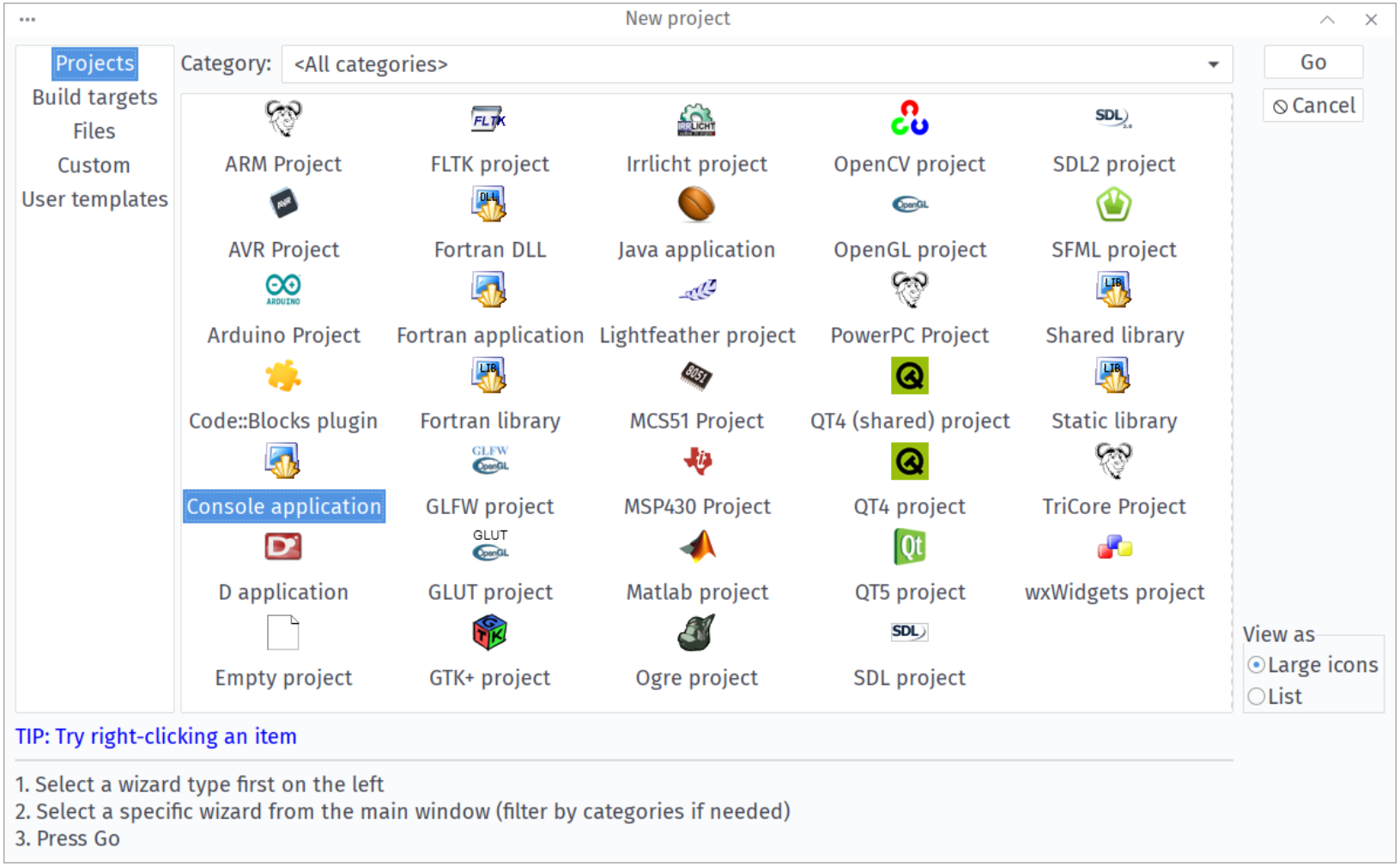
Plik `*.cbp` – plik *projektu*, przechowuje informacje specyficzne dla każdego projektu.

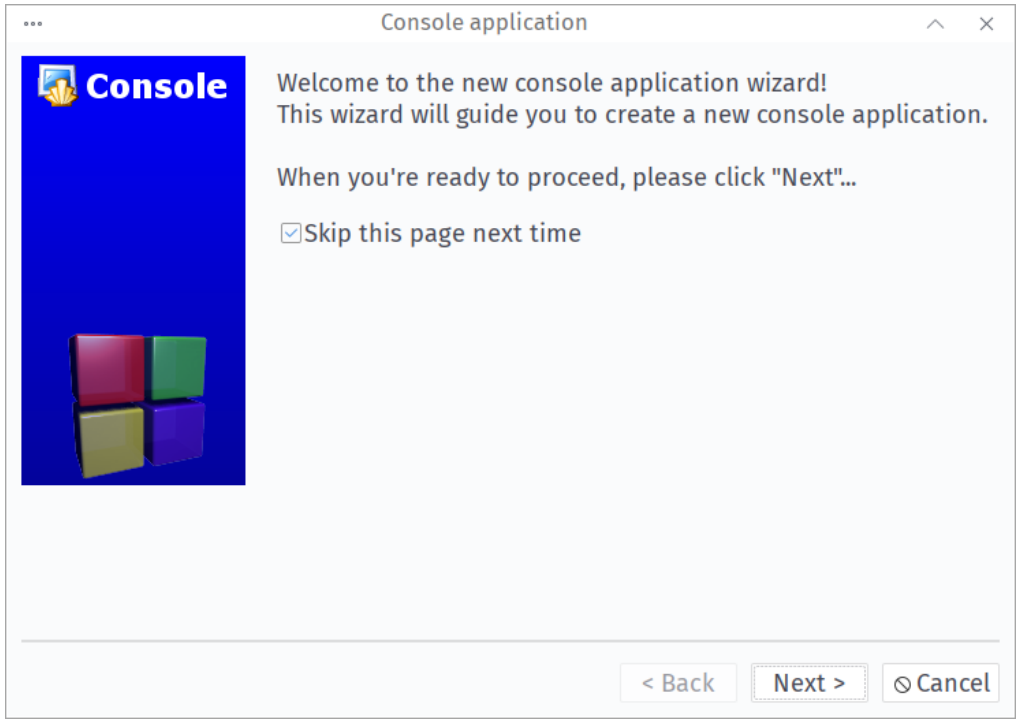


Plik `*.cpp` – plik *źródłowy*, zawiera kod źródłowy programu z funkcją `main()`.

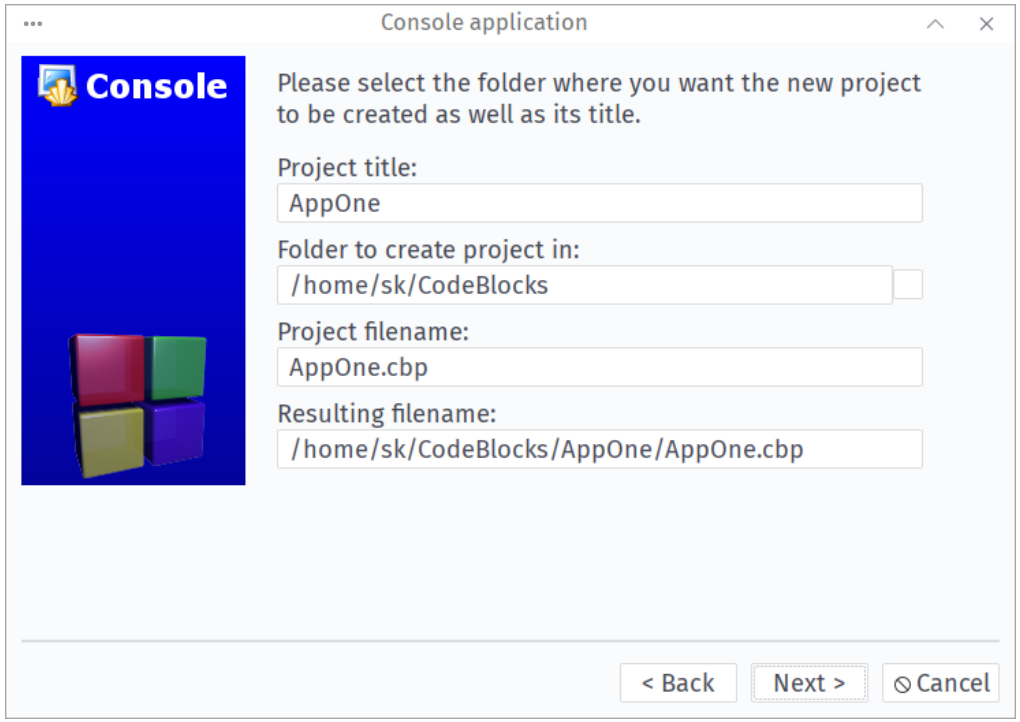


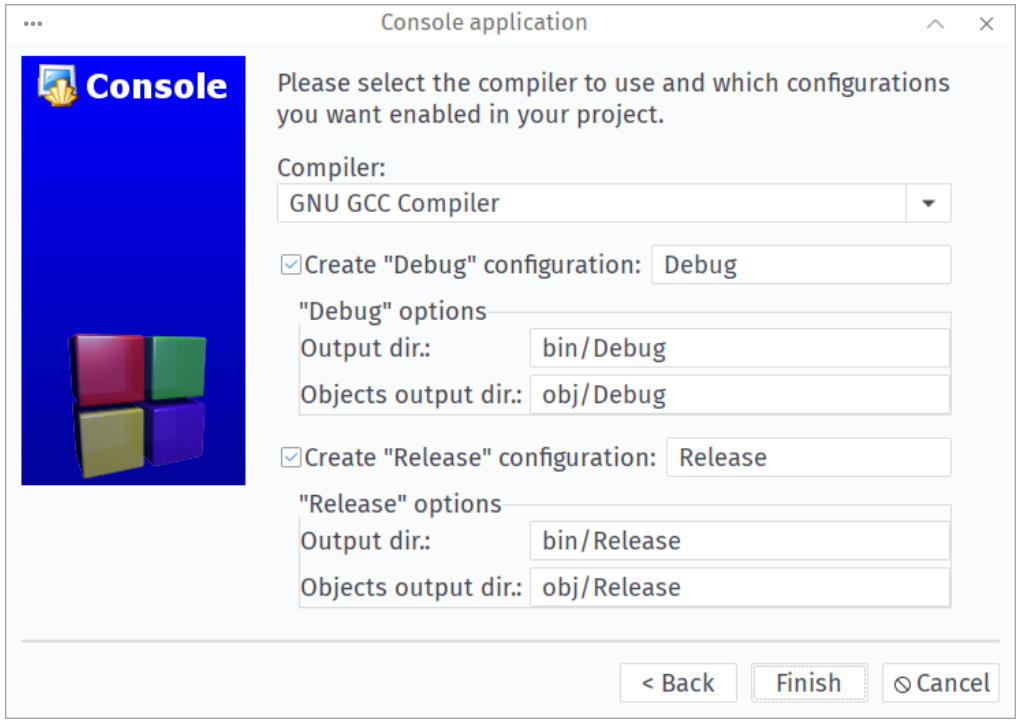
Projekt w języku C




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/sk/CodeBlocks
Project filename:
AppOne.cbp
Resulting filename:
/home/sk/CodeBlocks/AppOne/AppOne.cbp
< Back
Next >
Cancel

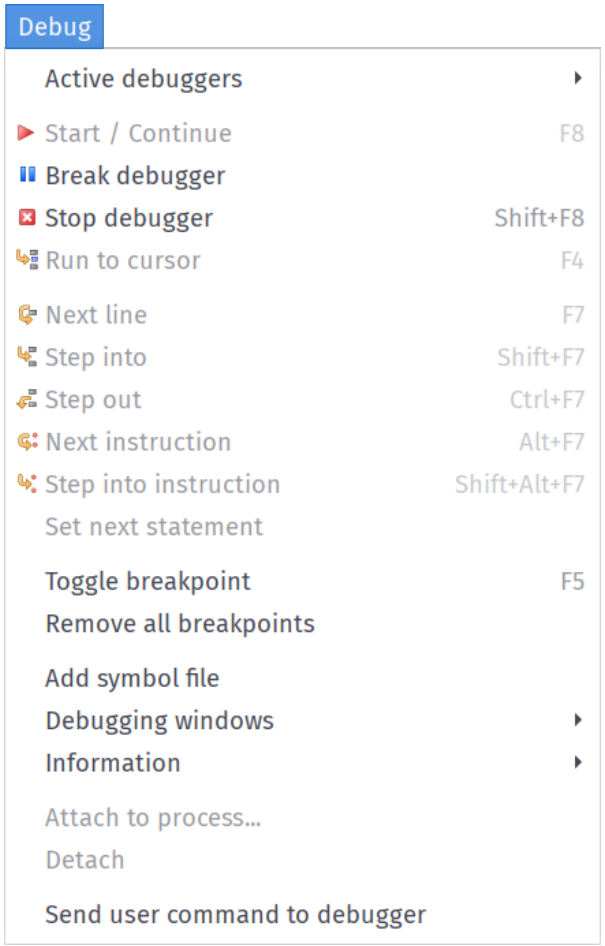

Console application
Console
Please select the compiler to use and which configurations you want enabled in your project.
Compiler:
GNU GCC Compiler
Create "Debug" configuration:
Debug
"Debug" options
Output dir.:
bin/Debug
Objects output dir.:
obj/Debug
Create "Release" configuration:
Release
"Release"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

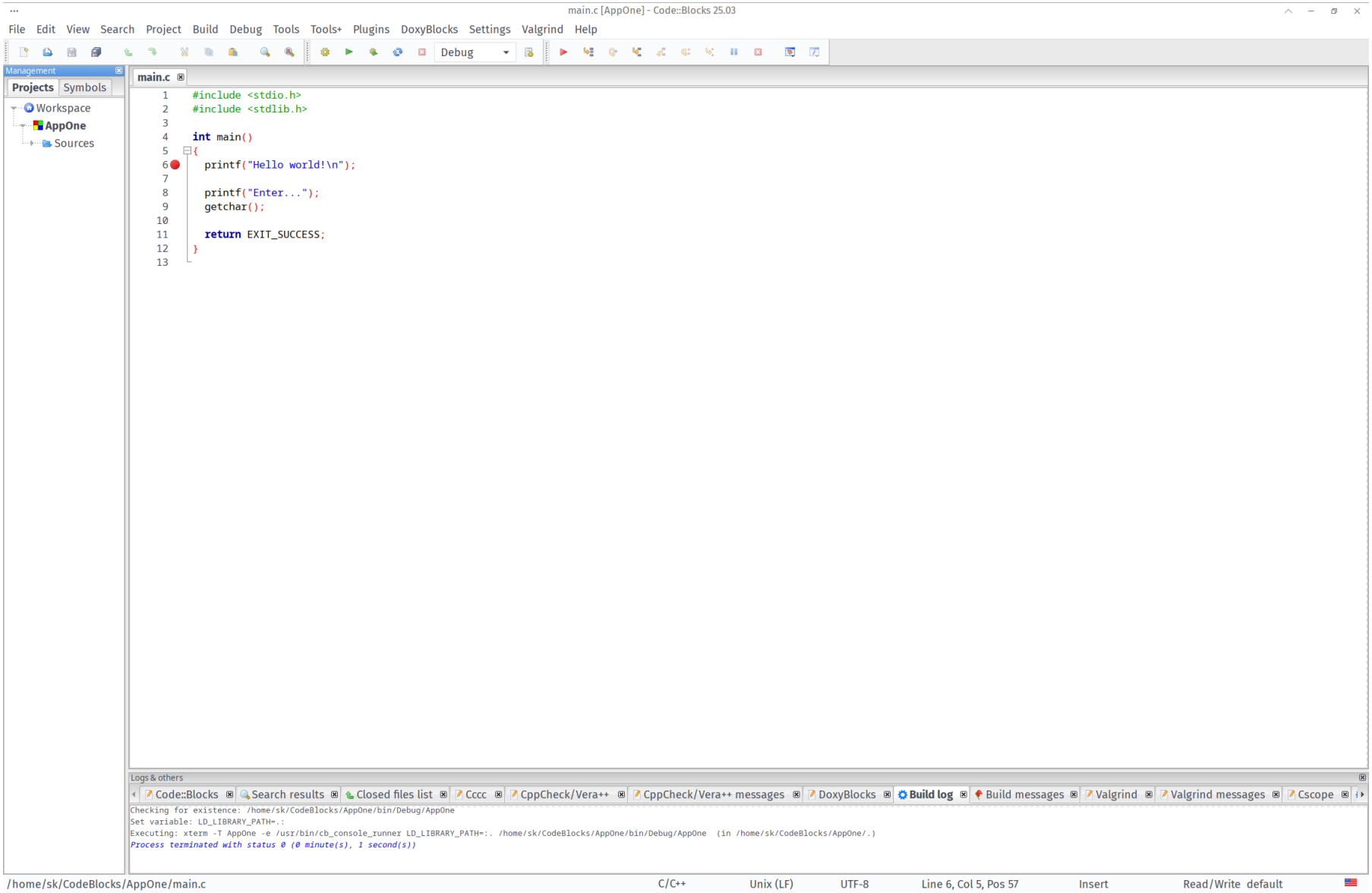
main.c

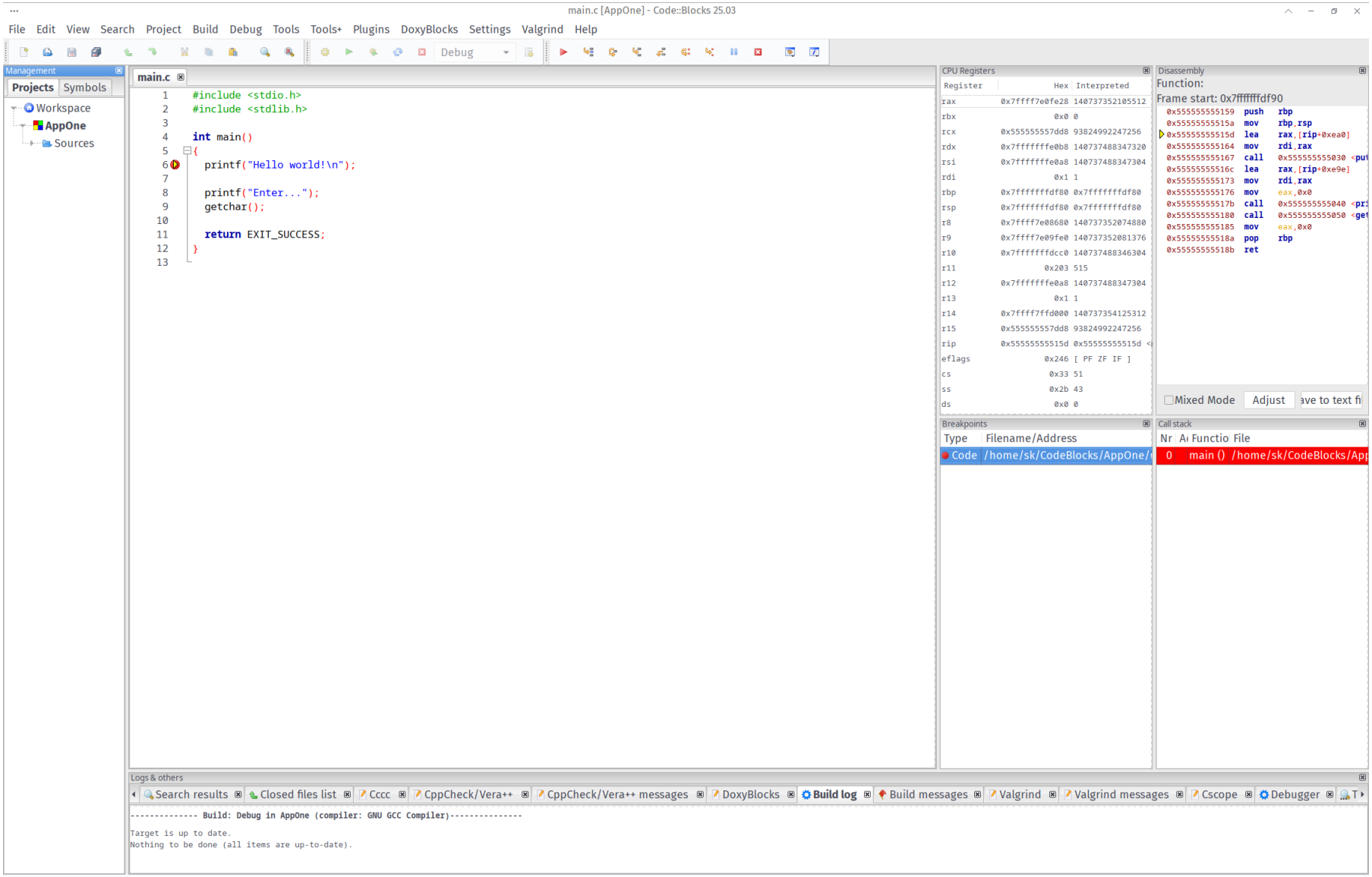
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  printf("Hello world!\n");

  printf("Enter...");
  getchar();

  return EXIT_SUCCESS;
}
```

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 25.03
File Edit View Search Project Build Debug Tools Tools+ Plugins DoxyBlocks Settings Valgrind Help
Debug
Management
Projects Symbols
Workspace
AppOne
Sources
main.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("Hello world!\n");
printf("Enter...");
getchar();
return EXIT_SUCCESS;
}
Logs & others
Code::Blocks Search results Closed files list Cccc CppCheck/Vera++ CppCheck/Vera++ messages DoxyBlocks Build log Build messages Valgrind Valgrind messages Cscope
Checking for existence: /home/sk/CodeBlocks/AppOne/bin/Debug/AppOne
Set variable: LD_LIBRARY_PATH=.:
Executing: xterm -T AppOne -e /usr/bin/cb_console_runner LD_LIBRARY_PATH=:. /home/sk/CodeBlocks/AppOne/bin/Debug/AppOne (in /home/sk/CodeBlocks/AppOne/.)
Process terminated with status 0 (0 minute(s), 1 second(s))
/home/sk/CodeBlocks/AppOne/main.c
C/C++
Unix (LF)
UTF-8
Line 6, Col 5, Pos 57
Insert
Read/Write default


main.c [AppOne] - Code::Blocks 25.03
File Edit View Search Project Build Debug Tools Tools+ Plugins DoxyBlocks Settings Valgrind Help
Debug
Management
Projects Symbols
Workspace
AppOne
Sources
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6 printf("Hello world!\n");
7
8 printf("Enter...");
9 getchar();
10
11 return EXIT_SUCCESS;
12 }
13
CPU Registers
Register Hex Interpreted
rax 0x7ffff7e0fe28 140737352105512
rbx 0x0 0
rcx 0x555555557dd8 93824992247256
rdx 0x7fffffffe0b8 140737488347320
rsi 0x7fffffffe0a8 140737488347304
rdi 0x1 1
rbp 0x7fffffffdf80 0x7fffffffdf80
rsp 0x7fffffffdf80 0x7fffffffdf80
r8 0x7ffff7e08680 140737352074880
r9 0x7ffff7e09fe0 140737352081376
r10 0x7fffffffdcc0 140737488346304
r11 0x203 515
r12 0x7fffffffe0a8 140737488347304
r13 0x1 1
r14 0x7ffff7ffd000 140737354125312
r15 0x555555557dd8 93824992247256
rip 0x55555555515d 0x55555555515d
eflags 0x246 [PF ZF IF]
cs 0x33 51
ss 0x2b 43
ds 0x0 0
Disassembly
Function:
Frame start: 0x7fffffffdf90
0x555555555159 push rbp
0x55555555515a mov rbp,rsp
0x55555555515d lea rax,[rip+0xea0]
0x555555555164 mov rdi,rax
0x555555555167 call 0x555555555030
0x55555555516c lea rax,[rip+0xe9e]
0x555555555173 mov rdi,rax
0x555555555176 mov eax,0x0
0x55555555517b call 0x555555555040
0x555555555180 call 0x555555555050
0x555555555185 mov eax,0x0
0x55555555518a pop rbp
0x55555555518b ret
Mixed Mode
Adjust
Breakpoints
Type Filename/Address
Code /home/sk/CodeBlocks/AppOne/
Call stack
Nr Functio File
0 main () /home/sk/CodeBlocks/App
Logs & others
Search results
Closed files list
Cccc
CppCheck/Vera++
CppCheck/Vera++ messages
DoxyBlocks
Build log
Build messages
Valgrind
Valgrind messages
Cscope
Debugger
---------- Build: Debug in AppOne (compiler: GNU GCC Compiler)----------
Target is up to date.
Nothing to be done (all items are up-to-date).
/home/sk/CodeBlocks/AppOne/main.c
C/C++
Unix (LF)
UTF-8
Line 6, Col 1, Pos 53
Insert
Read/Write
default

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

Budowa programu (który nic nie robi)

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

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

Budowa programu:

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

Co i jak wywołuje funkcję main?

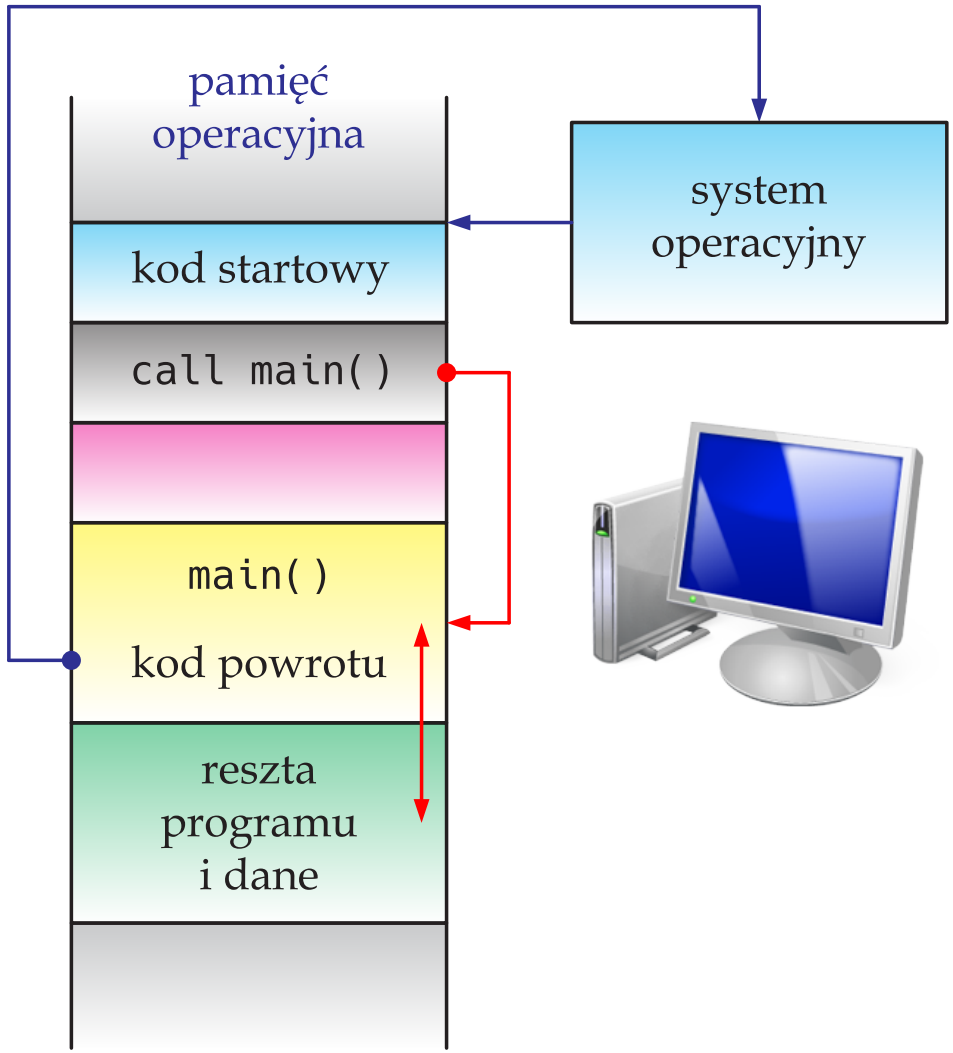
Funkcja `main()` stanowi punkt programu, od którego zaczyna się jego wykonywanie. Upraszczając, funkcja `main()` jest wywoływana przez **system operacyjny**.

Wartość, będąca rezultatem funkcji `main()`, jest przekazywana systemowi operacyjnemu (**kod wyjścia programu**). Przyjmuje się, że `0` oznacza poprawne zakończenie, a wartość niezerowa (zwykle `1`) – błąd. Biblioteka standardowa definiuje też symbole `EXIT_SUCCESS` i `EXIT_FAILURE` (`stdlib.h`), spotykane w kodzie dbającym o przenośność.

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 error

echo Exit code 0
goto end

:error
echo Exit code %errorlevel%

:end
pause
```

Przykładowy skrypt będący tzw. skryptem powłoki (*shell script*) Linuksa (`*.sh`) testujący kod wyjścia programu `foo`:

```
#!/bin/bash

./foo
code=$?

if [ $code -eq 0 ]; then

  echo "Exit code 0"

else

  echo "Exit code $code"

fi
```

Plik wykonywalny: ELF i PE

Ten sam kod w języku C skompilowany pod Linuksem i pod Windows daje pliki w dwóch różnych formatach: **ELF** (*Executable and Linkable Format*) oraz **PE** (*Portable Executable*). Kod maszynowy w środku korzysta z tego samego zestawu rozkazów procesora, ale nie jest identyczny.

ELF – Linux

- nagłówek `ELF`
  architektura, punkt wejścia
- Program Header Table
  segmenty ładowane do pamięci
- `.text`
  kod maszynowy
- `.rodata`
  stałe i literały tekstowe
- `.data` / `.bss`
  dane globalne i statyczne
- pozostałe sekcje (biblioteki, CRT)
  `.init`, `.fini`, `.plt`, `.got`, `.dynamic`, Section Header Table

PE – Windows

- nagłówek DOS `MZ` + `PE`
  relikt z lat osiemdziesiątych
- COFF + Optional Header
  architektura, `ImageBase`
- `.text`
  kod maszynowy
- `.rdata`
  stałe i literały tekstowe
- `.data` / `.bss`
  dane globalne i statyczne
- pozostałe sekcje (biblioteki, CRT)
  `.CRT`, `.idata`, `.edata`, `.reloc`

Niezależnie od systemu plik wykonywalny musi zawierać te same elementy logiczne, rozdzielone według tego, jak traktuje je system przy ładowaniu programu.

- `.text` – kod maszynowy; zestaw rozkazów jest ten sam, bo architektura jest ta sama, ale różni się **konwencja wywołań** (ABI) – rejestry, w których przekazywane są argumenty funkcji, i sposób korzystania ze stosu,
- `.rodata` w ELF, `.rdata` w PE – stałe i literały tekstowe, obszar ten jest mapowany tylko do odczytu, próba zmiany literału kończy się naruszeniem ochrony pamięci,
- `.data` – dane globalne i statyczne z podaną wartością początkową; wartości muszą być zapisane w pliku,
- `.bss` – dane globalne i statyczne bez wartości początkowej; w pliku zajmują zero bajtów, ponieważ zapisany jest jedynie wymagany rozmiar.

Tablica `int t[1000000]` zadeklarowana globalnie nie powiększa pliku wykonywalnego, a ta sama tablica z niezerowymi wartościami początkowymi powiększa go o cztery megabajty.

Ten sam fragment kodu skompilowany dla architektury x86-64 pod Linuksem i pod Windows:

```
long add3(long a, long b, long c);
long caller(void) { return add3(1, 2, 3) + 1; }
```

Linux (System V ABI)

```
push   rax
mov    edi, 1
mov    esi, 2
mov    edx, 3
call   add3
inc    rax
pop    rcx
ret
```

Windows (Microsoft x64 ABI)

```
sub    rsp, 40
mov    ecx, 1
mov    edx, 2
mov    r8d, 3
call   add3
inc    eax
add    rsp, 40
ret
```

Argumenty trafiają do innych rejestrów, Windows wymaga zarezerwowania na stosie 32 bajtów dla wywoływanej funkcji (*shadow space*), a typ `long` ma 64 bity w Linuksie i 32 bity w Windows, stąd `rax` i `eax`.

Kod startowy

Funkcja `main()` nie jest pierwszą funkcją wykonywaną po uruchomieniu programu. Poprzedza ją **kod startowy** biblioteki języka C, dołączany przez konsolidator niezależnie od tego, czy programista o nim wie.

W Linuksie punktem wejścia jest `_start` z pliku `crt1.o`, dostarczanego razem z biblioteką standardową. Wywołuje on `__libc_start_main()`, która inicjalizuje bibliotekę standardową, ustawia wskaźnik do zmiennych środowiska, wywołuje funkcje inicjalizujące i dopiero wtedy wywołuje `main()`. Po powrocie z `main()` wywoływana jest funkcja `exit()`, a ta uruchamia wszystko, co zarejestrowano przez `atexit()`.

W Windows odpowiednikiem jest `mainCRTStartup()`, a dla programów okienkowych `WinMainCRTStartup()` z biblioteki uruchomieniowej MSVCRT albo nowszej UCRT. Zadania są analogiczne: sterta, przetworzenie wiersza poleceń na `argv`, ustawienia regionalne, wywołanie `main()`.

Wartość zwrócona z `main()` nie jest zwracana systemowi bezpośrednio, tylko przejmuje ją kod startowy i dopiero on kończy proces, przekazując tę liczbę jako kod wyjścia.

Nagłówki

Plik ELF zaczyna się od czterech bajtów `\x7fELF`, po których następuje nagłówek z architekturą i adresem punktu wejścia oraz położenie dwóch tablic. **Program Header Table** opisuje segmenty, czyli bloki ładowane do pamięci przez system. **Section Header Table** opisuje sekcje, potrzebne konsolidatorowi i debugerowi, i można ją z gotowego programu usunąć.

Plik PE zaczyna się od nagłówka `MZ` z czasów systemu DOS, zawierającego mały program wypisujący komunikat o niemożności uruchomienia. Dalej jest sygnatura `PE`, nagłówek **COFF** oraz **Optional Header**, który mimo nazwy obowiązkowy jest zawsze i zawiera między innymi preferowany adres załadowania.

Nagłówki gotowego pliku ogląda się narzędziem właściwym dla formatu, w Linuksie jest to `readelf`, w Windows `dumpbin` z pakietu Visual Studio.

```
readelf -h program
```

```
dumpbin /headers program.exe
```

Biblioteki i dyrektywa `#include`

Funkcje wejścia/wyjścia, takie jak `printf()`, nie są częścią języka C, lecz jego **biblioteki standardowej**. Ich deklaracje znajdują się w pliku nagłówkowym `stdio.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.

Standardowe strumienie

Standardowe kanały komunikacji między komputerem a otoczeniem (zwykle terminalem). Występują w Linuksie 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
```

```
user@linux:~$ ./foo > output.txt
user@linux:~$ ./foo < input.txt
user@linux:~$ ./foo < input.txt > output.txt
```

Poważny program w języku C

```c
#include <stdio.h>

int main()
{
  float cm, inch;

  printf("Przeliczanie centymetrow na cale.\n");
  printf("Podaj wymiar w cm: ");

  scanf("%f", &cm);

  inch = 0.3937 * cm;

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

  return 0;
}
```

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

Podstawy obsługi wejścia/wyjścia

Funkcje obsługi wejścia i wyjścia występują w wielu wersjach – głównie wynika to z rozwoju języka. Ich implementacje są wtedy ustandaryzowane.

Część funkcji jest obligatoryjna (występują we wszystkich kompilatorach zgodnych z odpowiednim standardem), część opcjonalna. Przykładem są funkcje *bezpieczne* z przyrostkiem `_s` (aneks K standardu C11) – w praktyce dostępne tylko w MSVC, biblioteka gcc w Linuksie ich nie ma.

Część funkcji ma odmianę obsługującą tablicę znaków o typie `wchar_t` (*wide character*) zamiast `char`. Wtedy w ich nazwie pojawia się literka `w`.

Mogą zdarzać się też funkcje, które nie są częścią standardu (np. obsługujące *locale*, implementowane w kompilatorach MSVC). Funkcje niestandardowe zwykle są w jakiś sposób oznaczane, np. przez dodanie znaku podkreślenia na początku nazwy funkcji.

Funkcja `puts()` wysyła na standardowe wyjście napis `str`, a następnie znak nowej linii.

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

Funkcja `gets()` czyta linię ze standardowego wejścia i umieszcza ją w tablicy `str`, zastępując znak nowego wiersza `\n` zerem (`\0`). Nie sprawdza rozmiaru tablicy, więc zbyt długa linia nadpisuje pamięć – dlatego została **usunięta ze standardu C11**.

```
char *gets(char *str);                                    ← do C99
```

```
char *gets_s(char *str, rsize_t n);                       ← C11, aneks K
```

Przenośnym zamiennikiem jest `fgets()`, która czyta najwyżej `n - 1` znaków i **zostawia** `\n` w tablicy:

```
char *fgets(char *str, int n, FILE *stream);              ← fgets(s, sizeof(s), stdin)
```

Funkcja `printf()` formatuje tekst zgodnie z formatem i wypisuje tekst na standardowe wyjście (`stdout`).

```c
int printf(const char *format, ...);                                    ← do C99
```

```c
int printf(const char *restrict format, ...);                           ← od C99
```

```c
int printf_s(const char *restrict format, ...);                         ← od C11, aneks K
```

```c
#include <stdio.h>

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

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

  return 0;
}
```

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 z 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 6 (wartość 0 jest traktowana jak 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 precyzją z tą różnicą, że jest jeszcze poprzedzona kropką.

Rozmiar argumentu:

- dla formatów `d` i `i` można użyć jednego z modyfikatorów 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 modyfikatora 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 argumentu 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 `[-]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 `[-]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.

Funkcja `scanf()` odczytuje dane zgodnie z formatem ze standardowego wejścia (`stdin`).

```c
int scanf(const char *format, ...);                                    ← do C99

int scanf(const char *restrict format, ...);                           ← od C99

int scanf_s(const char *restrict format, ...);                         ← od C11, aneks K
```

W `scanf_s()` formaty `%s` i `%c` wymagają dodatkowego argumentu z rozmiarem tablicy: `scanf_s("%s", s, 80)`. Visual Studio domyślnie traktuje użycie `scanf()` jako błąd. Można to wyłączyć, definiując `_CRT_SECURE_NO_WARNINGS` przed dołączeniem nagłówków.

```c
#include <stdio.h>

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

  scanf("%d %f", &i, &f);
  scanf("%79s", s);          ← maks. 79 znaków + \0

  return 0;
}
```

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 niebędącego 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 sekwencję znaków niebędących 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 sekwencję 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`	’	Single quote
`\"`	`0x22`	”	Double quote
`\?`	`0x3F`	?	Question mark
`\O`			O = łańcuch cyfr ósemkowych
`\xH`			H = łańcuch cyfr szesnastkowych

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

Z czego jest zbudowany program?

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

Rozróżnia się następujące 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. Duże i małe litery są rozróżniane.

Standard wymaga rozróżniania co najmniej 63 znaków, współczesne kompilatory nie mają praktycznego limitu. Najnowsze gcc i MSVC przyjmują identyfikatory w UTF-8, zatem można stosować polskie znaki, ale **nie należy** tego robić.

Nazwy zaczynające się od dwóch podkreśleń lub od podkreślenia i dużej litery są zarezerwowane dla kompilatora i biblioteki standardowej. Pojedyncze podkreślenie na początku nazwy jest dopuszczalne tylko lokalnie – bezpieczniej go unikać.

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

```
maxvalue      mindelta      Pi_5
max_value     min_delta     Pi_Five
MaxValue      MinDelta      JW23
maxValue      minDelta      JW_23
```

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 standardu 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
```

W kolejnych wersjach pojawiły się kolejne słowa kluczowe.

```
inline          restrict        _Bool           _Complex    _Imaginary              ← C99
```

```
_Alignas        _Alignof        _Atomic         _Generic    _Noreturn               ← C11
_Static_assert  _Thread_local
```

```
alignas         alignof         bool            constexpr   false                   ← C23
nullptr         static_assert   thread_local    true        typeof
typeof_unqual   _BitInt         _Decimal128     _Decimal32  _Decimal64
```

Separatory

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

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

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

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

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

Komentarze

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

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

```
/* To jest komentarz jednoliniowy. */

/*
Ten komentarz obejmuje
kilka linii
kodu.
*/

// Komentarz jednowierszowy, bez znacznika konca.                ← od C99
```

Standard C89 (zgodny z 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 niepozwalającą 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.

W językach obiektowych dochodzą do tego poziomy dostępu do składowych klas (**prywatne**, **chronione**, **publiczne**). Zmienne mogą zmieniać swój pierwotny zasięg

na przykład poprzez importowanie/włączenie 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 oraz 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

Z typami mogą być stosowane modyfikatory: `signed`, `unsigned`, `short`, `long`.

typ	wielkość	format	wartości
`char`	1	`signed`	od `-128` do `127`
`char`	1	`unsigned`	od `0` do `255`
`short`	2	`signed`	od `-32 768` do `32 767`
`short`	2	`unsigned`	od `0` do `65 535`
`int`	4	`signed`	od `-2 147 483 648` do `2 147 483 647`
`int`	4	`unsigned`	od `0` do `4 294 967 295`
`long long`	8	`signed`	od `-9 223 372 036 854 775 808` do `9 223 372 036 854 775 807`
`long long`	8	`unsigned`	od `0` do `18 446 744 073 709 551 615`

Typ `long` ma 4 bajty w Windows i 8 bajtów w 64-bitowym Linuksie. Dawniej, w systemach 16-bitowych, typ `int` miał 2 bajty.

typ, wielkość, format, wartości			
`float`	4	**IEEE-754**	od `-3.402 823 4e+38` do `-1.175 494 4e-38`
			od `1.175 494 4e-38` do `3.402 823 4e+38`
`double`	8	**IEEE-754**	od `-1.797 693 134 862 315 7e+308`
			do `-2.225 073 858 507 201 4e-308`
			od `2.225 073 858 507 201 4e-308`
			do `1.797 693 134 862 315 7e+308`
`long double`	10 (12/16)	**x86**	od `-1.189 731 495 357 231 765 021e+4932`
			do `-3.362 103 143 112 093 506 263e-4932`
			od `3.362 103 143 112 093 506 263e-4932`
			do `1.189 731 495 357 231 765 021e+4932`
`long double`	16	**IEEE-754**	od `-1.189 731 495 357 231 765 085 759 326 628 007 016 2e+4932`
			do `-3.362 103 143 112 093 506 262 677 817 321 752 6e-4932`
			od `3.362 103 143 112 093 506 262 677 817 321 752 6e-4932`
			do `1.189 731 495 357 231 765 085 759 326 628 007 016 2e+4932`

Format x86 ma 10 bajtów danych, w pamięci zajmuje 12 (32 bity) lub 16 bajtów (64 bity). W MSVC `long double` jest tym samym co `double`.

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("%c\n", c);  ← A
printf("%c\n", d);  ← B

printf("%d\n", c);  ← 65
printf("%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`.

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 prefiksem `L`, np.
`wchar_t x = L'A';`

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 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 to, czy typ `char` ma znak, zależy od implementacji (na x86 zwykle ze znakiem, na ARM zwykle bez).

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

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:

```
sizeof(char) <= sizeof(short int) <= sizeof(int) <= 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.

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

int main()
{
  printf("         char: %d ... %d\n", CHAR_MIN, CHAR_MAX);
  printf("    short int: %hd ... %hd\n", SHRT_MIN, SHRT_MAX);
  printf("          int: %d ... %d\n", INT_MIN, INT_MAX);
  printf("     long int: %ld ... %ld\n", LONG_MIN, LONG_MAX);
  printf("long long int: %lld ... %lld\n\n", LLONG_MIN, LLONG_MAX);

  printf("         unsigned char: 0 ... %u\n", UCHAR_MAX);
  printf("    unsigned short int: 0 ... %hu\n", USHRT_MAX);
  printf("          unsigned int: 0 ... %u\n", UINT_MAX);
  printf("     unsigned long int: 0 ... %lu\n", ULONG_MAX);
  printf("unsigned long long int: 0 ... %llu\n", ULLONG_MAX);

  return 0;
}
```

```
         char: -128 ... 127
    short int: -32768 ... 32767
          int: -2147483648 ... 2147483647
     long int: -9223372036854775808 ... 9223372036854775807
long long int: -9223372036854775808 ... 9223372036854775807

         unsigned char: 0 ... 255
    unsigned short int: 0 ... 65535
          unsigned int: 0 ... 4294967295
     unsigned long int: 0 ... 18446744073709551615
unsigned long long int: 0 ... 18446744073709551615
```

Wynik z 64-bitowego Linuksa. W Windows `long int` ma taki sam zakres jak `int`.

Przybliżone zakresy zmiennych

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

- typ `char` to ok. 128 na plus i minus, `unsigned char` to 255 na plus,
- typ `short int` to ok. 32 tys. na plus i minus, `unsigned short int` to ok. 65 tys. 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` w Windows jak `int` (32 bity), w 64-bitowym Linuksie jak `long long` (64 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

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

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

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

  return 0;
}
```

```
65535
0
1

32767
-32768
-32767
```

Przekroczenie zakresu zachowuje się różnie w zależności od typu:

- typy **bez znaku** – zachowanie jest określone, arytmetyka odbywa się modulo 2^n (po wartości maksymalnej następuje 0),
- `short` **ze znakiem** – `si++` jest obliczane w typie `int`, a wynik konwertowany z powrotem do `short`; rezultat zależy od implementacji (w praktyce zawija się do wartości ujemnej),
- `int` **ze znakiem** – przepełnienie jest **niezdefiniowanym zachowaniem**.

Kompilator przy optymalizacji zakłada, że przepełnienie `int` nie występuje:

```
int check(int x) { return x + 1 > x; }

check(INT_MAX);     ← clang -O0: 0, clang -O2: 1
```

Typy pochodne typów zmiennopozycyjnych

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

- jedyną poprawną kombinacją jest `long double`,
- zapis `long float` był w przedstandardowym C (K&R) synonimem typu `double`, od standardu C89 jest błędem,
- kombinacje `short float` i `short double` nigdy nie były poprawne – typu zmiennopozycyjnego nie można skrócić.

Wybrane stałe symboliczne z pliku `float.h`:

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

int main()
{
  printf("        Rozmiar mantysy float w bitach: %d\n", FLT_MANT_DIG);
  printf("Minimalna liczba cyfr znaczacych float: %d\n\n", FLT_DIG);

  printf("      Minimalny ujemny wykladnik float: %+d\n", FLT_MIN_10_EXP);
  printf("    Maksymalny dodatni wykladnik float: %+d\n\n", FLT_MAX_10_EXP);

  printf("      Minimalna dodatnia wartosc float: %+e\n", FLT_MIN);
  printf("     Maksymalna dodatnia wartosc float: %+e\n\n", FLT_MAX);

  printf("Roznica miedzy 1.00 a najmniejsza wart. wieksza od 1.00: %+e\n", FLT_EPSILON);

  return 0;
}
```

```
        Rozmiar mantysy float w bitach: 24
Minimalna liczba cyfr znaczacych float: 6

      Minimalny ujemny wykladnik float: -37
    Maksymalny dodatni wykladnik float: +38

      Minimalna dodatnia wartosc float: +1.175494e-38
     Maksymalna dodatnia wartosc float: +3.402823e+38

Roznica miedzy 1.00 a najmniejsza wart. wieksza od 1.00: +1.192093e-07
```

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 zapisywany 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.

Od standardu C23 literały rozpoczynające się od przedrostka `0b` lub `0B` są traktowane jako dwójkowe (gcc dopuszczał je wcześniej jako rozszerzenie).

```
int i = 10;       ← stała dziesiętna
int o = 077;      ← stała ósemkowa
int h = 0xff;     ← stała szesnastkowa

int b = 0b1010;   ← stała dwójkowa, od C23
```

Literał całkowitoliczbowy może być zakończony przyrostkiem:

- `u` lub `U` – liczba **bez znaku**,
- `l` lub `L` – liczba **długa** (`long`),
- `ll` lub `LL` – liczba **bardzo długa** (`long long`, od C99).

Przyrostki można łączyć, np. `10ul`, `10ULL`.

Literał bez przyrostka ma pierwszy z typów `int`, `long`, `long long`, w którym mieści się jego wartość (dla literałów ósemkowych i szesnastkowych brane są pod uwagę także typy bez znaku).

Wartość literału nie może przekraczać zakresu największego typu całkowitego, czyli `unsigned long long`. Według standardu to błąd, choć gcc tylko ostrzega i obcina wartość. Dla 64 bitów wartość maksymalna wynosi:

```
DEC: 18 446 744 073 709 551 615, OCT: 01777777777777777777777, HEX:
0xFFFFFFFFFFFFFFFF
```

Typ wyliczeniowy

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

Typ wyliczeniowy to tak naprawdę 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 muszą być unikatowe w całym zasięgu, nie tylko w ramach danego wyliczenia – dzielą przestrzeń nazw ze zmiennymi i funkcjami. Dlatego często poprzedza się je przedrostkiem, np. `COLOR_RED`, `BODY_SEDAN`.

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 nieposiadający 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++)
```

Typ logiczny

W standardzie C23 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 z 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` albo `E` oraz wykładnika potęgi ze znakiem,
- opcjonalnego przyrostka `f` albo `F` i `l` albo `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.	1
-1.23	$-1,23$
2e-5	$2 \cdot 10^{-5}$
3E+10	$3 \cdot 10^{10}$
.09E34	$0,09 \cdot 10^{34}$

Znak minus nie jest częścią literału – zapis `-1.23` to jednoargumentowy operator `-` zastosowany do literału `1.23`.

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 nieudostępniająca rezultatu:

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

Bezparametrowa funkcja, nieudostępniająca rezultatu:

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

Rzutowanie rezultatu funkcji na typ `void`:

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

Plik nagłówkowy `stdint.h` (C99) definiuje *przenośne* typy całkowite o określonych właściwościach.

Typy o dokładnym rozmiarze (*exact width types*), np. `int16_t`, `uint16_t`, `int32_t`, `uint32_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.

Plik `inttypes.h` dołącza `stdint.h` i dodatkowo definiuje makra formatujące dla `printf()` i `scanf()`, np. `PRIu8`, `PRIu16`, `PRIu32`, `PRIu64`, `PRIuFAST8`, `PRIuFAST16`, `PRIuFAST32`, `PRIuFAST64`, itp.

```
uint32_t x = 42;
printf("x = %" PRIu32 "\n", x);
```

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.

Od standardu C11 tablice VLA są **opcjonalne**.

Kompilator MSVC nigdy nie zaimplementował tablic VLA, również w trybie C11/C17 – standard pozwala mu na to, bo od C11 są one opcjonalne. Kompilator, który ich nie obsługuje, definiuje makro `__STDC_NO_VLA__`.

```
int n;

scanf("%d", &n);
int arr[n];        ← gcc: poprawne, MSVC: błąd C2057
```

Przenośne rozwiązania:

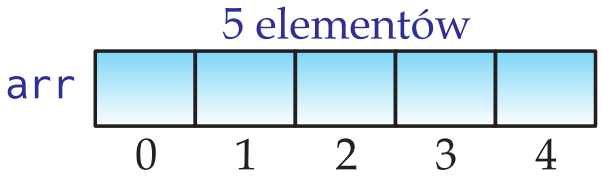
- tablica o maksymalnym rozmiarze ustalonym na etapie kompilacji (`#define MAX_N 100`) i wykorzystywanie tylko jej części,
- dynamiczna alokacja pamięci funkcją `malloc()` – omówiona w dalszej części wykładu.

W języku C++ tablic VLA nie ma wcale (g++ dopuszcza je jako rozszerzenie), co ma znaczenie przy kompilowaniu kodu C jako `*.cpp`.

Deklarowanie tablic jednowymiarowych

```
int arr[5];
```

```
char buff[80];
```



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

```
#define MAX_N 5
int arr[MAX_N];
```

```
#define MAX_BUFF 80
char buff[MAX_BUFF];
```

Zmienna z kwalifikatorem `const` w języku C nie jest **wyrażeniem stałym**, więc nie może określać rozmiaru tablicy globalnej. Wewnątrz funkcji zapis się skompiluje, ale jako tablica VLA (gcc tak, MSVC nie).

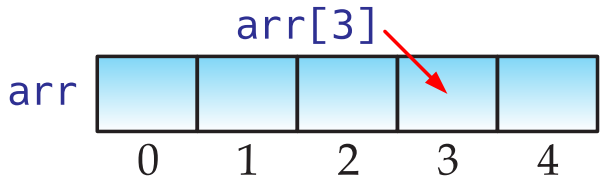
W języku C++ zmienna z kwalifikatorem `const` zainicjowana stałą jest wyrażeniem stałym i może określać rozmiar tablicy.

Standard C23 wprowadza słowo `constexpr`, które pozwala zdefiniować prawdziwą stałą – wrócimy do tego przy omawianiu stałych.

```
const int MAX_N = 5;
int arr[MAX_N];        ← C: błąd lub VLA, C++: poprawne
```

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`.



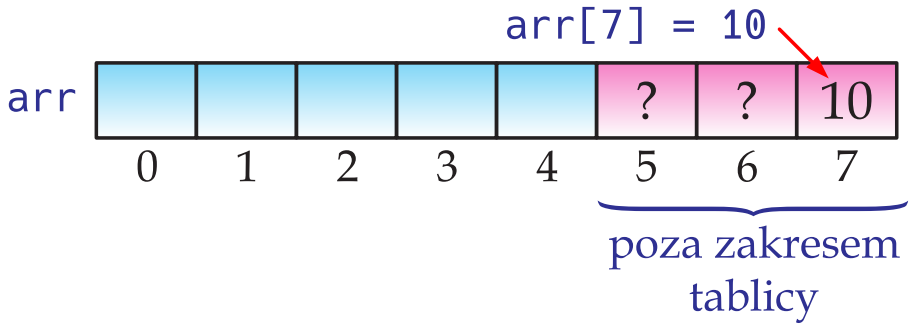

```
arr[0] = 1;
```

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

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

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

W języku 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 months = sizeof(monthDays) / sizeof(int);
```

lub

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

int months = 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++)
  arr[i] = 0;
```

lub z wykorzystaniem operatora postinkrementacji:

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

Wczytywanie danych ze strumienia wejściowego do tablicy:

```
for(i = 0; i < n; i++)
{
  printf("%d: ", i + 1);
  scanf("%d", &arr[i]);
}
```

Wyprowadzanie danych z tablicy do strumienia wyjściowego:

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

Wersja *uproszczona*:

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

Wersja wspak:

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

Zagadka:

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

i = 2;
arr[++i] = 2 * ++i;  ← arr = ?
```

Zagadka:

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

i = 2;
j = arr[++i] + arr[i];  ← j = ?
```

Rozwiązanie:

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

i = 2;
arr[++i] = 2 * ++i;  ← gcc: 1, 2, 3, 4, 8; clang: 1, 2, 3, 4, 6
```

Rozwiązanie:

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

i = 2;
j = arr[++i] + arr[i];  ← j = 8, ale tylko przypadkiem
```

Oba wyrażenia modyfikują i odczytują zmienną `i` bez ustalonej kolejności – to **niezdefiniowane zachowanie**. Kompilator może obliczyć je dowolnie, a różne kompilatory dają różne wyniki. W jednym wyrażeniu nie należy modyfikować zmiennej, której wartość jest w nim jeszcze raz używana.

Kompilatory wykrywają takie konstrukcje, jeżeli włączy się ostrzeżenia (`-Wall`):

```
gcc:   warning: operation on 'i' may be undefined [-Wsequence-point]
```

```
clang: warning: multiple unsequenced modifications to 'i' [-Wunsequenced]
clang: warning: unsequenced modification and access to 'i' [-Wunsequenced]
```

Pierwsze ostrzeżenie clang zgłasza dla wyrażenia `arr[++i] = 2 * ++i`, drugie dla `j = arr[++i] + arr[i]`. Kompilator gcc zgłasza to samo ostrzeżenie dla obu.

Sumowanie wartości elementów tablicy:

```
int sum = 0;

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

Wersja *uproszczona*:

```
int sum;

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

Zagadka:

```
int sum;

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

Suma dodatnich elementów podzielnych przez 3, o parzystych indeksach:

```
int sum;

for(i = 0, sum = 0; i < n; i += 2)
  if(arr[i] > 0)
    if(!(arr[i] % 3))
      sum += arr[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 parametrów 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 `string.h`). Funkcja `memcpy()` wymaga, aby obszary nie zachodziły na siebie, `memmove()` radzi sobie również z obszarami nakładającymi się.

```
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]));
```

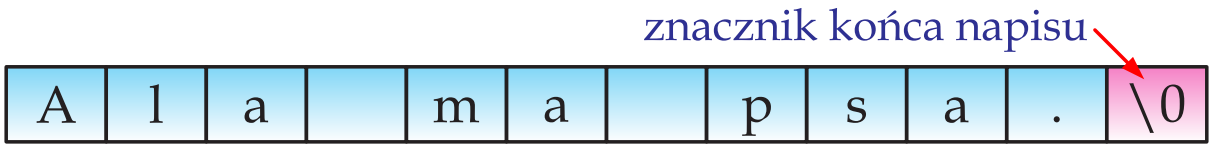
`memset()` wypełnia `n` bajtów obszaru pamięci bajtem o zadanej wartości.

Łańcuchy znakowe jako tablice znaków

Typ `char` jest wbudowanym typem języków C/C++ służącym do reprezentacji pojedynczego znaku o wielkości 1 bajtu. Do reprezentacji **łańcuchów znaków** (**napisów**) wykorzystuje się zwykłe tablice znakowe.

Tablice takie nie różnią się od innych tablic w języku C, wprowadzono jedynie *kilka udogodnień*, czyniących łatwiejszym manipulowanie takimi tablicami.

W języku C przyjęto koncepcję łańcuchów ze **znacznikiem końca** (*null terminated strings*). Znak zerowy `'\0'` jest znakiem niedrukowalnym o kodzie `0` – ma tę samą wartość co liczba `0`, ale zapis `'\0'` podkreśla, że chodzi o znak. Nie należy go mylić z cyfrą `'0'` (kod 48) ani ze wskaźnikiem `NULL`. Łańcuchy są zawsze przechowywane razem z tym znakiem. Fizyczna długość napisu jest większa o 1 znak.



"Ala ma psa."

Tablice znakowe można inicjować w zwykły sposób, przewidziany dla tablic.

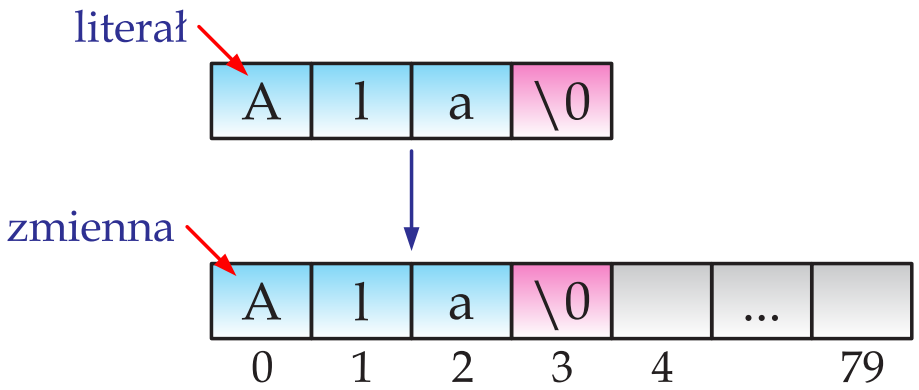
```
char name[80] = {'A', 'l', 'a'};          ← czy dobrze?
char name[80] = {'A', 'l', 'a', '\0'};    ← czy dobrze?
```

Można wykorzystywać wygodniejszą formę:

```
char name[80] = "Ala";
```

lub

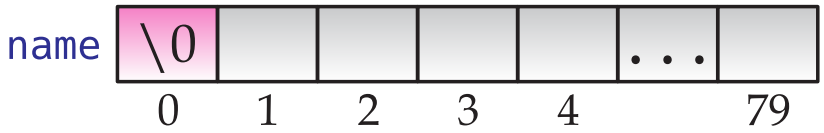
```
char name[] = "Ala";
```



Deklaracja łańcucha zainicjowanego napisem pustym:

```
char name[80] = {'\0'};
char name[80] = "";
```

Reprezentacja wewnętrzna łańcucha pustego.

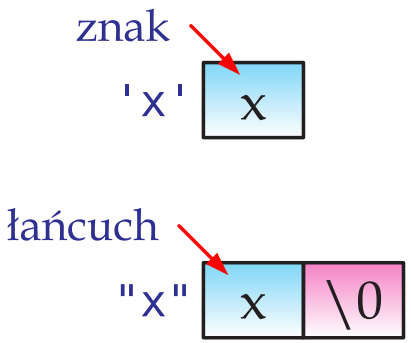


Ustawianie łańcucha pustego po deklaracji:

```
name[0] = '\0';
```

Stała łańcuchowa `"x"` nie jest tym samym co stała znakowa `'x'`. Pierwszą różnicą jest fakt, iż `'x'` jest wartością typu prostego (w C typu `int`, w C++ typu `char`), natomiast `"x"` jest typu pochodnego, tablicy opartej na typie `char`.

Druga różnica polega na tym, że `"x"` naprawdę składa się z dwóch znaków: `'x'` oraz `'\0'`.



Przetwarzanie tablic polega zwykle na *przemaszerowaniu* zmienną indeksową po tablicy, dopóki nie ma końca napisu, oznaczanego znakiem `'\0'`.

```
int i;
char str[80];

for(i = 0; str[i] != '\0'; i++)
  → operacje na każdym znaku str[i]
```

Wyprowadzanie zawartości napisu `str` do strumienia wyjściowego znak po znaku:

```
for(i = 0; str[i] != '\0'; i++)
  putchar(str[i]);
```

lub krócej

```
for(i = 0; str[i] != '\0'; putchar(str[i++]));
```

Do manipulowania tablicami znakowymi opracowano szereg funkcji bibliotecznych (plik nagłówkowy `string.h`). Wybrane funkcje do pracy z tablicami znakowymi:

- `memchr()` – szuka wystąpienia bajtu,
- `memcmp()` – porównuje bloki pamięci,
- `memcpy()` – kopiuje zawartość jednego bloku pamięci do drugiego,
- `memmove()` – kopiuje zawartość jednego bloku pamięci do drugiego (bloki mogą na siebie zachodzić),
- `memset()` – wypełnia pamięć bajtem,
- `strcat()` – scala dwa łańcuchy znaków w jeden,
- `strchr()` – szuka pierwszego wystąpienia znaku w łańcuchu znaków,

- `strcmp()` – porównuje dwa łańcuchy znaków,
- `strcoll()` – porównuje dwa łańcuchy znaków zgodnie z bieżącymi ustawieniami regionalnymi (*locale*),
- `strcpy()` – kopiuje łańcuch znaków do tablicy znaków,
- `strcspn()` – szuka pierwszego wystąpienia znaku (z puli znaków) w łańcuchu znaków,
- `strlen()` – oblicza długość łańcucha znaków,
- `strncat()` – scala dwa łańcuchy znaków w jeden; uwzględnia maksymalną liczbę znaków, jaka może zostać dopisana,
- `strncmp()` – porównuje określoną liczbę znaków dwóch łańcuchów znaków,

- `strncpy()` – kopiuje określoną liczbę znaków łańcucha,
- `strpbrk()` – szuka pierwszego wystąpienia znaku (z puli znaków bez `'\0'`) w łańcuchu znaków,
- `strrchr()` – szuka ostatniego wystąpienia znaku w łańcuchu znaków,
- `strspn()` – zwraca indeks pierwszego znaku, który nie należy do puli znaków,
- `strstr()` – szuka pierwszego wystąpienia łańcucha znaków w innym łańcuchu znaków,
- `strtok()` – dzieli łańcuch znaków na fragmenty (tokeny) rozdzielone znakami z podanej puli; modyfikuje łańcuch, wstawiając w miejsce separatorów znak `'\0'`.

Funkcje konwertujące łańcuch znaków na liczbę (`atoi()`, `atol()`, `strtol()`, `strtoul()`, ...) znajdują się w pliku nagłówkowym `stdlib.h`.

Standard nie przewiduje porównywania łańcuchów bez uwzględniania wielkości liter. W MSVC służy do tego funkcja `_stricmp()`, w systemach zgodnych z POSIX `strcasecmp()` (plik nagłówkowy `strings.h`).

Tablice dwuwymiarowe

Tablice dwuwymiarowe są rozszerzeniem tablic jednowymiarowych.

```
float rain[5][12];
```

Deklaracja składa się z dwóch części, pierwsza:

```
float rain[5][12];
```

będąca tablicą pięciu elementów, oraz druga:

```
float rain[5][12];
```

deklaracji tych elementów `float[12]`.

Zgodnie z tą logiką `rain[0]`, będący pierwszym elementem **tablicy** `rain`, jest **tablicą dwunastu** wartości typu `float`.

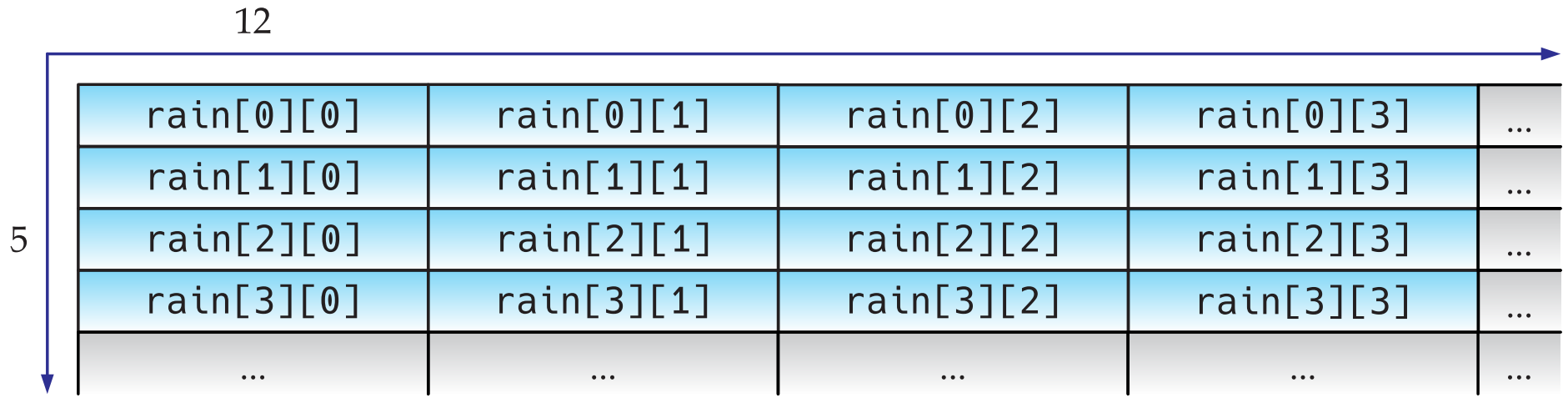
Podobnie jest w przypadku elementów `rain[1]`, `rain[2]` itd.

Skoro `rain[0]` jest tablicą, to jej pierwszym elementem jest `rain[0][0]`, drugim jest `rain[0][1]` itd.

Zatem `rain` jest tablicą pięciu elementów, które są dwunastoelementowymi tablicami typu `float`, `rain[0]` jest tablicą dwunastu wartości typu `float`, a `rain[0][0]` jest zmienną typu `float`.

Aby odwołać się do zmiennej w drugim wierszu i trzeciej kolumnie, musimy napisać `rain[1][2]`.

Tablicę `rain` można przedstawić jako dwuwymiarową tablicę składającą się z pięciu wierszy, każdy po dwanaście kolumn.



Widok dwuwymiarowy jest jedynie wygodnym sposobem przedstawienia. W rzeczywistości taka tablica przechowywana jest w ciągłym obszarze pamięci – najpierw pierwsza dwunastoelementowa tablica, potem druga dwunastoelementowa tablica i tak po kolei.

Inicjalizacja tablic dwuwymiarowych

Inicjalizacja tablic dwuwymiarowych stanowi rozwinięcie techniki inicjalizacji tablicy jednowymiarowej.

```
#define MONTHS 12
#define YEARS 5
...
float rain[YEARS][MONTHS] =
{
  {4.3, 4.3, 4.3, 3.0, 2.0, 1.2, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6},
  {8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.3},
  {9.1, 8.5, 6.7, 4.3, 2.1, 0.8, 0.2, 0.2, 1.1, 2.3, 6.1, 8.4},
  {7.2, 9.9, 8.4, 3.3, 1.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 6.2},
  {7.6, 5.6, 3.8, 2.8, 3.8, 0.2, 0.0, 0.0, 0.0, 1.3, 2.6, 5.2}
};
```

Można także pominąć wewnętrzne klamry i zachować jedynie obie zewnętrzne. Tak długo, jak długo w klamrach znajduje się właściwa liczba wartości, skutek będzie identyczny.

Jeżeli podamy zbyt mało wartości, tablica będzie wypełniana kolejno wiersz po wierszu, dopóki nie zabraknie danych. Pozostałe elementy będą inicjalizowane wartością `0`.

```
int arr[2][3] = {{5, 6}, {7, 8}};
```

5	6	0
7	8	0

```
int arr[2][3] = {5, 6, 7, 8};
```

5	6	7
8	0	0

Odwołania do elementów tablic

```
arr[0][1] = 1;
```

```
arr[n - 1][3] = 5;
```

```
a = 2 * arr[3][5];
```

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

```
int year, month;
float total, sum;

total = 0;
for(year = 0; year < YEARS; year++)
{
  sum = 0;

  for(month = 0; month < MONTHS; month++)
    sum += rain[year][month];

  total += sum;
}
```

Tablice wielowymiarowe

Wszystkie podane informacje można uogólnić na tablice, które mają więcej wymiarów.

Przykładowa deklaracja tablicy trójwymiarowej:

```
int box[10][20][30];
```

Przykładowa deklaracja tablicy pięciowymiarowej:

```
int box[10][20][30][40][50];
```

Taka tablica zajmuje $10 \cdot 20 \cdot 30 \cdot 40 \cdot 50 \cdot 4$ bajtów, czyli 48 MB. Zadeklarowana jako zmienna lokalna przepełni stos (domyślnie 1 MB w Windows, 8 MB w Linuksie).

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 języku C stałe definiuje się za pomocą:

- dyrektywy preprocesora `#define`,
- stałych wyliczeniowych (`enum`) – są prawdziwymi stałymi całkowitymi,
- zmiennych z kwalifikatorem `const` – kompilator nie pozwala ich zmieniać, ale nie są wyrażeniami stałymi,
- słowa `constexpr` (od C23) – prawdziwe stałe, również innych typów niż całkowite.

```
#define PI 3.1415
enum { MAX_N = 100 };
const double E = 2.7182;
constexpr double TWO_PI = 2 * 3.1415;                    ← od C23
```

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

```c
#include <math.h>
#define PI 3.1415

void bar(void)
{
  const float foo = 2.7182 * sin(PI / 4);  ← wartość znana dopiero w trakcie wykonania
}
```

`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.

Kilka przykładów niejawnej konwersji:

```
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 *
char *str = "foo";         ← konwersja z char[4] 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.

Literał napisowy jest w C typu `char[N]`, a nie `const char[N]` – słówko `const` pojawiło się w języku później i jego dodanie zepsułoby istniejący kod. Mimo to próba zapisu do literału jest niezdefiniowanym zachowaniem (literały leżą w obszarze tylko do odczytu), dlatego bezpieczniej jest pisać `const char *str = "foo";`. W C++ literał jest typu `const char[N]` i przypisanie go do `char*` jest błędem.

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

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

```
int i = 3;
float fi = (float)i;
```

W języku C++ dostępna jest też notacja funkcyjna (w praktyce działa tak samo jak rzutowanie z C), a zalecanym sposobem jest operator `static_cast<>`, który pozwala tylko na bezpieczne konwersje i łatwo go odnaleźć w kodzie.

```
double d = 3.14;
int id = int(d);                    ← notacja funkcyjna
int is = static_cast<int>(d);       ← zalecane w C++
```

Wiele bibliotek oferuje dodatkowo funkcje do konwersji odpowiednich typów, zgodne z ich algorytmami zamiany. Na przykład po włączeniu 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`.

Funkcje te nie zgłaszają błędów: dla napisu niebędącego liczbą zwracają 0 (nie da się odróżnić od napisu `"0"`), a przekroczenie zakresu typu jest niezdefiniowanym zachowaniem.

Bezpieczniejsze są funkcje `strtol()`, `strtoul()` i `strtod()`, które wskazują miejsce, w którym zakończyła się konwersja, a przy przekroczeniu zakresu ustawiają zmienną `errno` na `ERANGE`.

Dziwne deklaracje

Język C pozwala tworzyć bardzo zawiłe formy danych. Znaczenie deklaracji może zostać zmienione przez dodanie do nazwy zmiennej *modyfikatora*. Możliwe jest też użycie wielu modyfikatorów w jednej deklaracji.

```
int foo[8][8];       ← tablica tablic typu int
int **bar;           ← wskaźnik do wskaźnika do int
int *baz[10];        ← 10-elementowa tablica wskaźników do int
int (*qux)[10];      ← wskaźnik do 10-elementowej tablicy typu int
int *buff[3][4];     ← tablica 3 × 4 wskaźników do int
int (*puff)[3][4];   ← wskaźnik do tablicy 3 × 4 wartości int
int (*huff[3])[4];   ← 3-elementowa tablica wskaźników do
                       4-elementowych tablic typu int
```

```
char *foo(int);        ← funkcja zwracająca wskaźnik do char
char (*bar)(int);      ← wskaźnik do funkcji, która zwraca
                         wartość typu char
char (*baz[3])(int);   ← tablica 3 wskaźników do funkcji
                         zwracających wartość typu char
```

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

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

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

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 jeden 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 **prawostronnie 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 i jakie mają priorytety, określa definicja języka. Od implementacji (kompilatora) może natomiast zależeć kolejność obliczania argumentów – w języku C jest ona nieokreślona.

Zróżnicowany jest również sposób zapisu operatorów arytmetycznych. Stosuje się zapis, bądź za pomocą symboli (znaku lub znaków niebędących 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 w językach programowania dowolne wyrażenie typu liczbowego. Może być ono złożone ze zmiennych, liczb, funkcji, symboli działań (tu: operatorów), itp.

```
float result;

result = 10 / 3;       ← 3 - dzielenie w int, wynik zamieniony na float
result = 10.0 / 3.0;   ← 3.333333
```

```
int result;

result = 10 * 3;       ← 30
result = 10.0 / 3.0;   ←  3 - dzielenie w double, wynik obcięty przy przypisaniu
result = 10 % 3;       ←  1
result = 10 + 3;       ← 13
result = 10 - 3;       ←  7
result =  5;           ←  5
result = -result;      ← -5
```

Typ działania zależy od typów argumentów. Typ zmiennej decyduje dopiero o konwersji gotowego wyniku przy przypisaniu.

Kolejność (priorytety) wykonywania działań jest taka jak w matematyce, można ją regulować za pomocą nawiasów okrągłych. Operatory o tym samym priorytecie grupowane są zgodnie z ich łącznością (wiązaniem), zwykle od lewej do prawej.

wybrane funkcje matematyczne z `math.h`	
`sin()`	sinus
`cos()`	cosinus
`tan()`	tangens
`asin()`	arcus sinus
`acos()`	arcus cosinus
`atan()`	arcus tangens
`exp()`	funkcja wykładnicza (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 niespełnienie 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 i ich priorytet określa definicja języka. Od implementacji (kompilatora) może natomiast zależeć kolejność obliczania argumentów.

Zróżnicowany jest również sposób zapisu operatorów logicznych, stosuje się zapis, bądź w postaci słów kluczowych, bądź symboli (znaku lub znaków niebędących 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 w 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 = (value % 2) == 0;
range = ((value >= 10) && (value <= 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 **prefiksowej** i **postfiksowej** – 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;
```

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 struktur)	
`++ --` (przyrostkowe)	L
`*` (wskaźnik) `&` (adres) `+ -` (znak) `!` (negacja) `~` (negacja bitowa)	
`++ --` (przedrostkowe) `sizeof` `(typ)` (rzutowanie)	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 łączność, czyli sposób grupowania operatorów o tym samym priorytecie.

```
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
```

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

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("k = %f", k);   ← k = 10.2
  }
  printf("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("Suma jest rowna 3.");
```

if - then - else

```
float take, bonus;

take = 29999.99;

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

if - then - else if - then - else

```
float 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("Brak pierwiastkow rzeczywistych.");
```

Uwaga na zagnieżdżone instrukcje warunkowe

Która wersja sugerowana przez wcięcia jest poprawna?

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

```
if(value >= 0)
  if(value > 0)
    printf("Liczba dodatnia.");
  else
    printf("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.

```c
int body;

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

printf("Podaj typ nadwozia: ");
scanf("%d", &body);

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

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

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

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

```c
int body;

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

printf("Podaj typ nadwozia: ");
scanf("%d", &body);

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

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

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

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

int body;

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

printf("Podaj typ nadwozia: ");
scanf("%d", &body);
...
```

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

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

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

  default:
    printf("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, którą nazywa się **zmienną sterującą**, **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 zmniejszana.

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, zasięg.

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 pozwala 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:

- ustalenie jako bieżący, pierwszego elementu kontenera danych, jeżeli kontener jest pusty, zakończenie działania pętli,
- przypisanie wartości 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(index > 10);

printf("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("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 index;

index = 0;                          ← inicjalizacja

while(index < 10)                   ← warunek kontynuacji
{
  printf("indeks = %d\n", index);   ← ciało iteracji
  index++;                          ← modyfikacja zmiennej sterującej
}
```

```
int index;

for(index = 0; index < 10; index++)  ← inicjalizacja
                                        warunek kontynuacji
                                        modyfikacja zmiennej
  printf("indeks = %d\n", index);    ← ciało iteracji
```

W części inicjalizacyjnej wolno deklarować zmienne (w C od standardu C99, wcześniej tylko w C++), np. `for(int i = 0; i < n; i++)`. Zasięg takiej zmiennej ogranicza się do pętli.

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

```c
int index;

for(index = 0; index < 10; printf("indeks = %d\n", index), index++);
```

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

```c
int key, c;

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

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

  if(key == EOF || key == '0')
    break;

  if(key == '\n')                              ← samo Enter
    continue;

  while((c = getchar()) != '\n' && c != EOF);  ← czyszczenie bufora wejścia

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

Instrukcje `break` i `continue`

Instrukcja `break` pozwala na **natychmiastowe zakończenie** 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.

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

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/I**). 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/I**,
  - `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/I**, **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 parametrom formalnym zdefiniowanym w podprogramie,
- rezultatów funkcji (wartości zwracanych do miejsca wywołania),
- pól obiektu (dla metod danego obiektu),
- innych, rzadko stosowanych lub niezalecanych (np.: obszarów wspólnych, zmiennych nakładanych).

Funkcje

```c
#include <stdio.h>

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

int main()
{
  float b, p;

  printf("Podaj dlugosc boku: ");
  scanf("%f", &b);

  p = squareArea(b);
  printf("Pole = %f.", p);

  return 0;
}
```

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.

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

```c
#include <stdio.h>

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

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

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

  printInfo();
  scanf("%f", &b);

  p = squareArea(b);
  printf("Pole = %f.", p);

  return 0;
}
```

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 (do standardu C17) puste nawiasy oznaczają **nieokreśloną liczbę parametrów** – kompilator nie sprawdza argumentów wywołania. Brak parametrów zapisuje się jawnie słowem `void`.

```
int foo()              ← C do C17: parametry nieokreślone; C23 i C++: brak parametrów
{
  ...
}
```

```
int foo(void)          ← brak parametrów
{
  ...
}
```

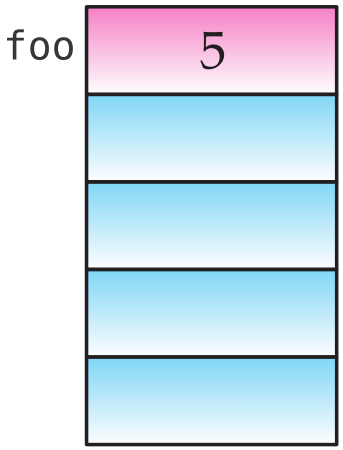
Funkcja o **zmiennej liczbie parametrów** (jak `printf()`) wymaga przed wielokropkiem co najmniej jednego nazwanego parametru.

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

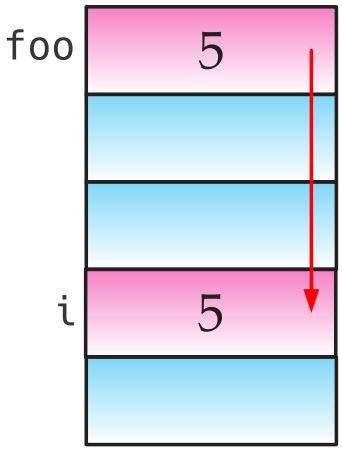
Przekazywanie parametrów

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

int foo = 5;

inc(foo);
printf("foo = %d", foo);   ← foo = 5
```

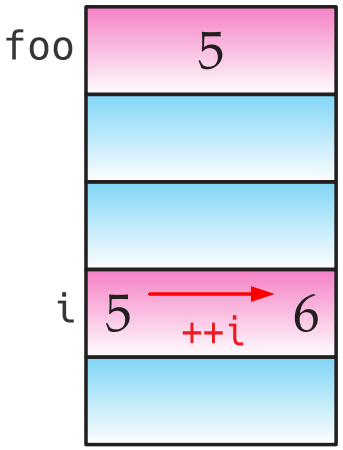
przed wywołaniem `inc(foo)`



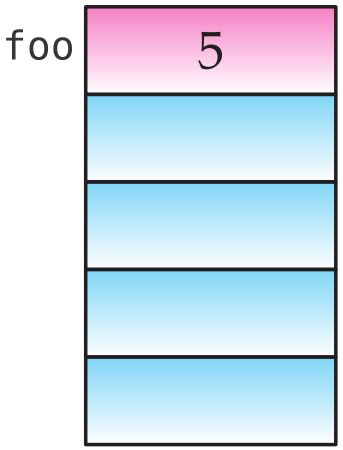
wywołanie `inc(foo)`



wykonanie `inc(foo)`



po wykonaniu `inc(foo)`



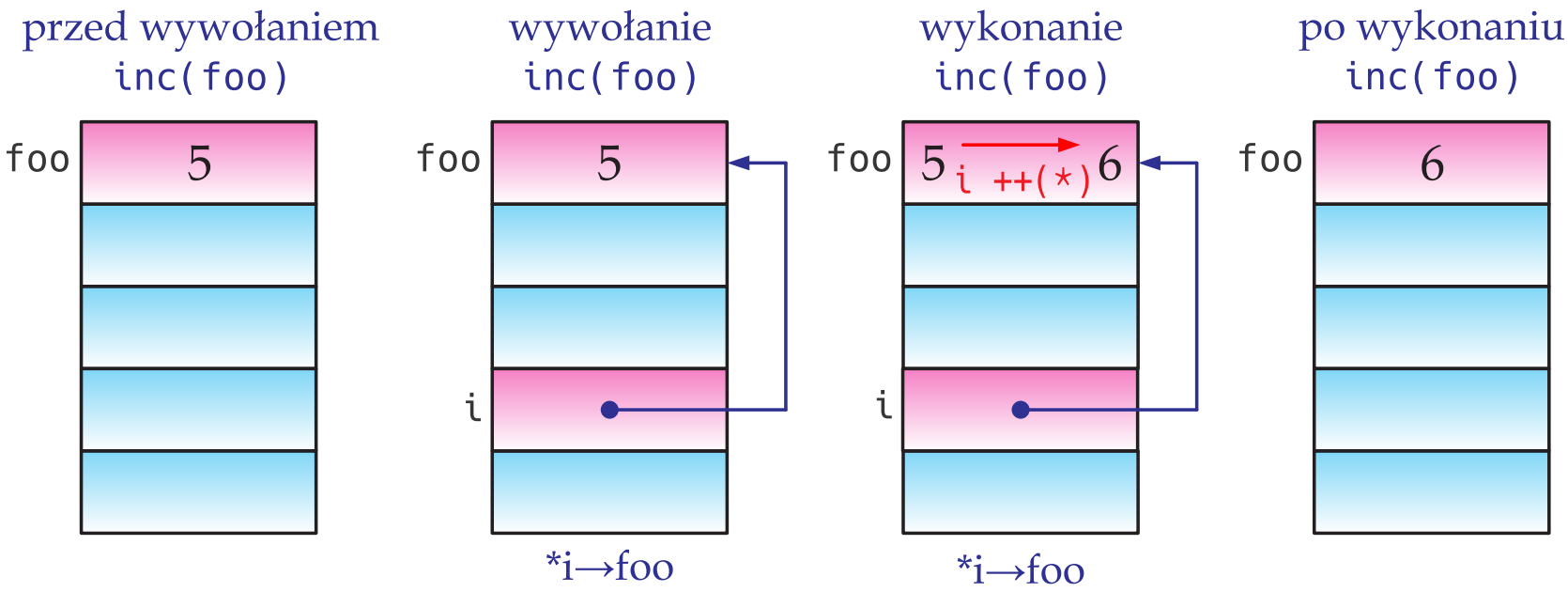
Przy przekazywaniu parametrów przez wartość, wartość **parametru aktualnego** wywołania funkcji **kopiowana** jest do **parametru formalnego** funkcji.

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 aktualnego wywołania.

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

int foo = 5;

inc(&foo);
printf("foo = %d", foo);  ← foo = 6
```



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

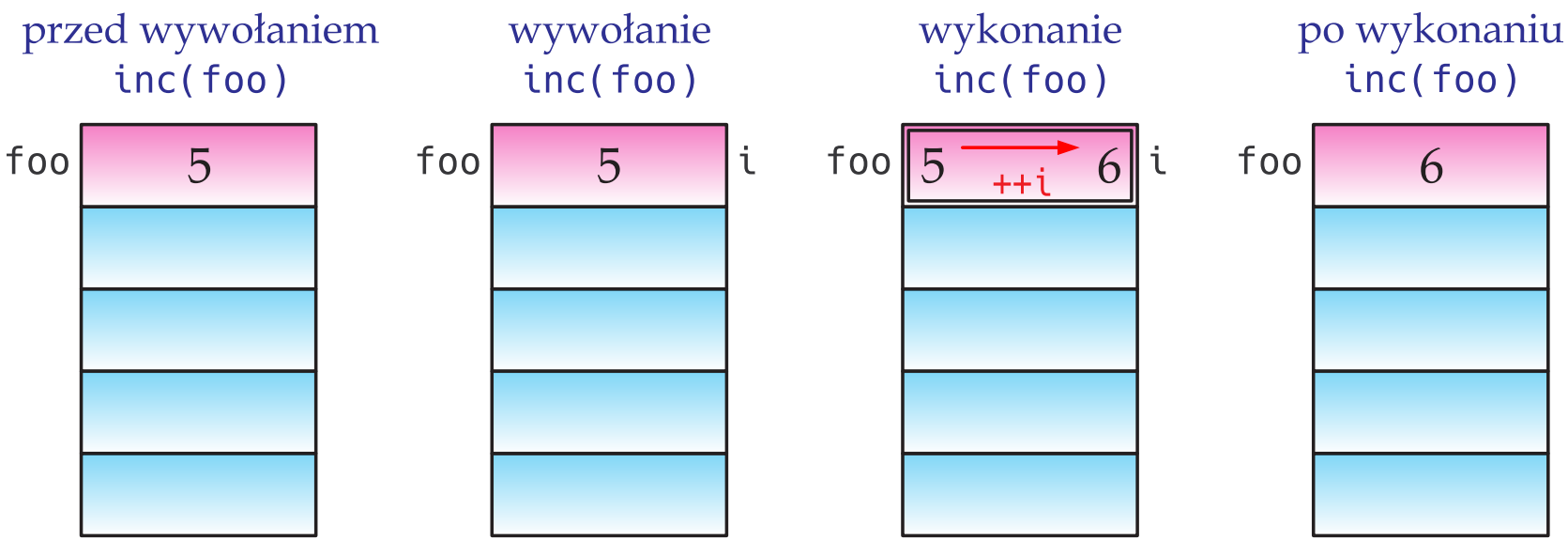
Wskaźniki także są przekazywane przez **wartość**, wynika to z faktu, że wskaźnik też jest typem zmiennej (zmienna wskaźnikowa).

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

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

W języku C nie ma referencji i nie można przekazywać parametru przez referencję (można m.in. w C++).

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

int foo = 5;

inc(foo);
printf("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 aktualny wywołania.

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.

```c
#include <stdio.h>

float squareArea(float);

int main()
{
  float b, p;

  printf("Podaj dlugosc boku: ");
  scanf("%f", &b);

  p = squareArea(b);
  printf("Pole = %f.", p);

  return 0;
}

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

Starsze implementacje C dopuszczały wywoływanie funkcji wcześniej kompilatorowi nieznanych. Standard C99 tę możliwość usunął, a gcc od wersji 14 domyślnie traktuje takie wywołanie jako błąd.

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

Nie pozwalało 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 (od standardu C99) i C++ można definiować zmienne w obrębie zasięgu instrukcji.

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

i = 0;  ← błąd
```

Zmienne statyczne

Słowo kluczowe **static** wpływa na zasięg i czas życia zmiennej.

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;  ← inicjalizacja wykonuje się raz

  if(counter < 10)
    counter++;
  else
    return;
}
```

Zasięg pozostaje niezmieniony, nazwa jest widoczna tylko wewnątrz funkcji. Zmienia się czas życia, który obejmuje teraz całe działanie programu.

Zmienna taka nie leży na stosie, lecz w tym samym obszarze co zmienne globalne.

Jednostką translacji w C jest pojedynczy plik źródłowy wraz ze wszystkim, co wciągnęły do niego dyrektywy `#include`.

Kompilator przetwarza każdą jednostkę osobno, a powstałe pliki obiektowe łączy dopiero konsolidator.

Nazwa opatrzona słowem `static` na poziomie pliku nie trafia do zewnętrznej tablicy symboli pliku obiektowego. Konsolidator jej nie widzi, więc inne jednostki translacji nie mogą się do niej odwołać.

```
static int counter = 0;   ← niewidoczny poza tym plikiem
static void foo(void) {}  ← niewidoczny poza tym plikiem
```

Daje to dwie korzyści. Po pierwsze, nazwy nie kolidują – dwa pliki mogą mieć własną funkcję `foo( )`. Po drugie, moduł można potraktować jako całość o określonym interfejsie – na zewnątrz widoczne jest tylko to, co udostępniono świadomie.

Zmienne extern

Słowo `extern` informuje kompilator, że nazwa istnieje, ale zdefiniowana jest w innej jednostce translacji.

```
// plik main.c
int foo = 100;    ← definicja, tworzy zmienną
```

```
// plik function.c
extern int foo;  ← deklaracja, tylko zapowiedź
```

Definicja rezerwuje pamięć i w całym programie może wystąpić dokładnie raz.
Deklaracja jedynie zapowiada istnienie nazwy i może pojawić się wielokrotnie.
Powiązaniem jednego z drugim zajmuje się konsolidator.

Dla funkcji `extern` jest domyślne, dlatego prototypy w plikach nagłówkowych zapisuje się bez tego słowa.

Częsty błąd polega na umieszczeniu w pliku nagłówkowym **definicji** zmiennej globalnej. Po dołączeniu nagłówka w różnych plikach powstają wielokrotne definicje tej samej nazwy.

W nagłówku powinno umieszczać się deklarację z `extern`, a definicję w jednym wybranym pliku źródłowym.

deklaracja, miejsce, czas życia, zasięg, symbol				
`int i;`	w funkcji	blok	blok	brak
`static int i;`	w funkcji	cały program	blok	brak
`int i;`	poza funkcjami	cały program	do końca pliku	eksportowany
`static int i;`	poza funkcjami	cały program	do końca pliku	brak
`extern int i;`	poza funkcjami	jak definicji	do końca pliku	wymagany

Kolumna **symbol** opisuje, co trafia do pliku obiektowego i czym dysponuje konsolidator.

Wiersz z `extern` ma w kolumnie **czasu życia** odwołanie do definicji, ponieważ sam żadnej zmiennej nie tworzy.

Praktycznie wszystko, co nie musi być widoczne na zewnątrz pliku, opatruje się słowem `static`. Zmniejsza to ryzyko kolizji nazw i ułatwia późniejsze zmiany.

Zmienne współdzielone przez wiele plików stosuje się oszczędnie, ponieważ utrudniają śledzenie, a w programach wielowątkowych wymagają dodatkowej ostrożności.

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.

W C nie można pobrać adresu zmiennej `register` (operator `&`). W C++17 słowo `register` usunięto – przy kompilacji kodu C jako `*.cpp` kompilator zgłosi ostrzeżenie lub błąd.

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

Zmienne o nieprzewidywalnie zmiennej wartości

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

Obiektami takimi mogą być zmienne zmieniane poza normalnym przebiegiem programu – rejestry urządzeń (np. w mikrokontrolerach) czy zmienne modyfikowane w funkcji obsługi sygnału lub przerwania.

Kompilator nie może pominąć, połączyć ani przenieść odczytów i zapisów takiej zmiennej – każde odwołanie w kodzie musi trafić do pamięci.

```c
#include <signal.h>

volatile sig_atomic_t stop = 0;

void onSignal(int sig)
{
  stop = 1;                        ← wywoływana np. po Ctrl+C
}

...

signal(SIGINT, onSignal);
while(!stop)
{
  ...
}
```

Bez `volatile` kompilator może uznać, że `stop` nie zmienia się wewnątrz pętli, i odczytać ją tylko raz – pętla nigdy się nie zakończy.

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 użytku 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 `foofun.h`:

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

Część implementacyjna modułu `foofun.c`:

```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 konieczne.

Moduły mogą też włączać inne pliki nagłówkowe (`math.h` i stała `M_PI`). Stała `M_PI` nie należy do standardu C – w MSVC wymaga zdefiniowania `_USE_MATH_DEFINES` przed włączeniem `math.h`, a w gcc nie jest dostępna w trybie ścisłym (`-std=c17`).

```c
#include "foofun.h"
#define _USE_MATH_DEFINES
#include <math.h>

double squareArea(double side)
{
  return side * side;
}
double squarePerimeter(double side)
{
  return 4 * side;
}
...
```

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 (w języku C z `static`, w przeciwnym razie każdy plik dołączający nagłówek utworzy własną definicję i konsolidator zgłosi błąd):

```
static const int maxSpeed = 50;
static 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)
{
  ...
}
```

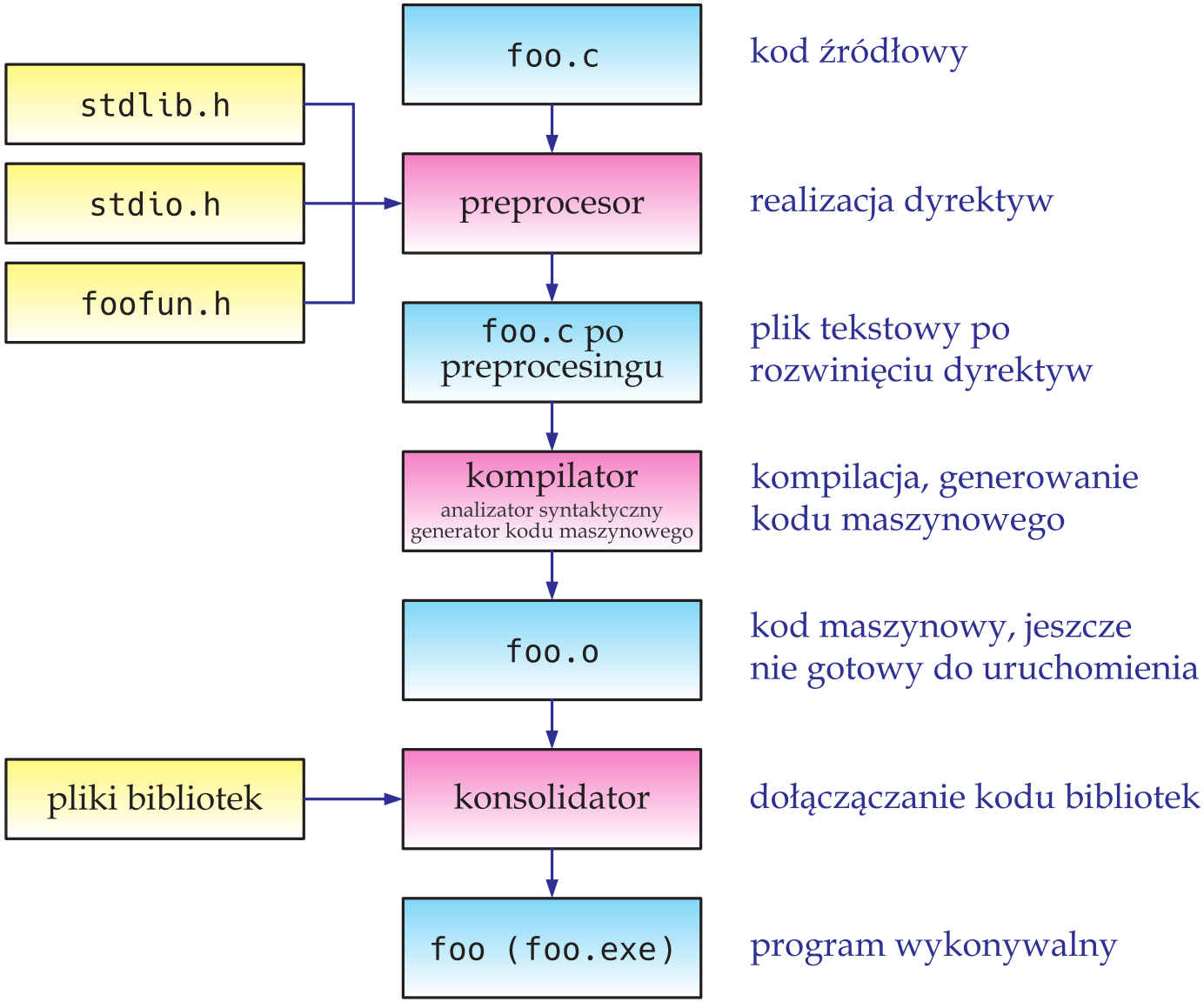
Kompilacja warunkowa

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

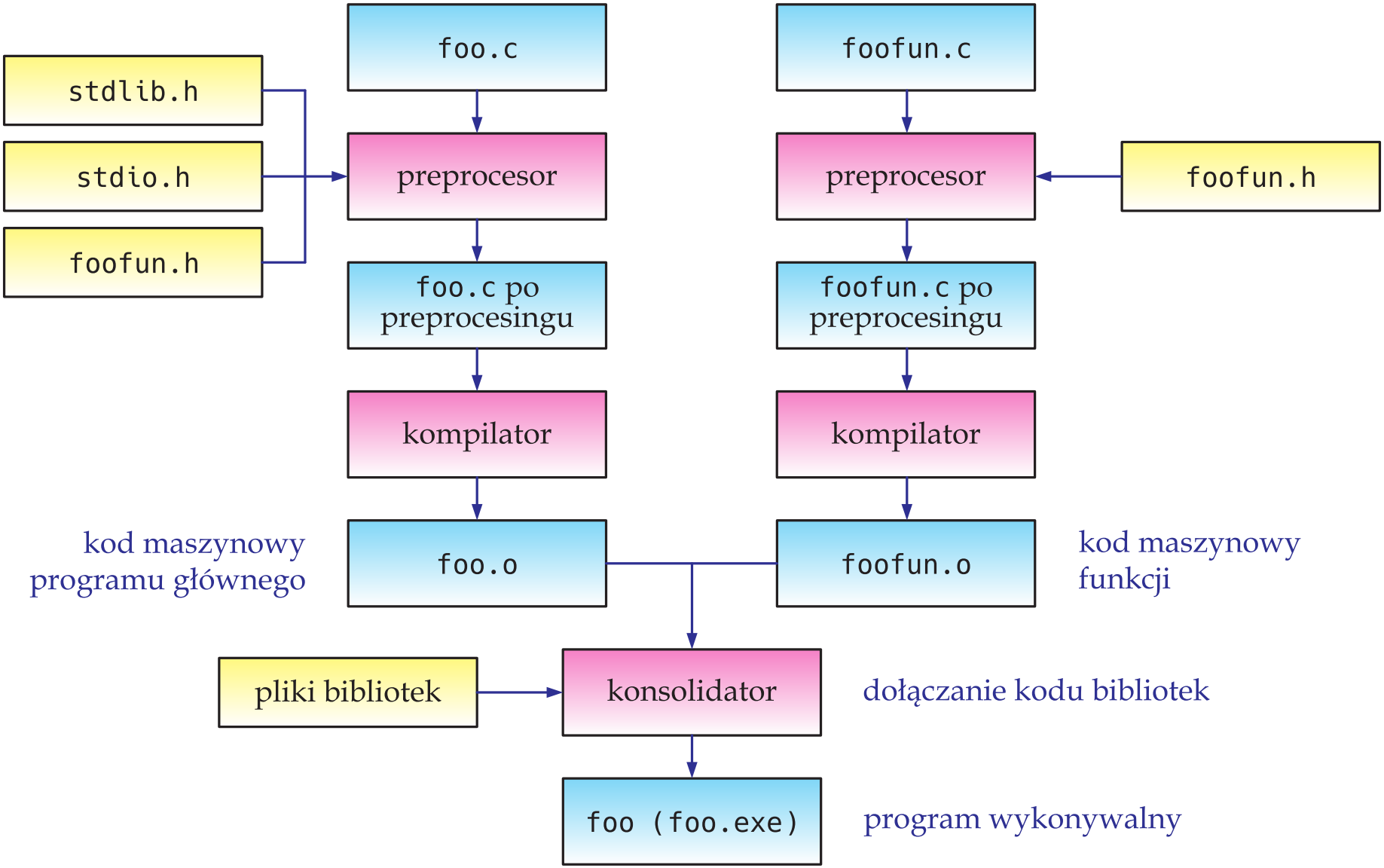
```
#ifndef FOOFUN_H
#define FOOFUN_H
static const double myPI = 5.0;.

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

#endif
```

Strażnik `#ifndef` chroni przed wielokrotnym włączeniem w obrębie **jednego** pliku źródłowego, nie chroni przed wielokrotną definicją w różnych plikach.

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` do tymczasowego pliku pośredniego (usuwanego po zakończeniu),
- połączenie go z plikami bibliotek i wygenerowanie pliku wynikowego `a.out` (Linux) lub `a.exe` (Windows).

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

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

W przypadku GNU/Linux:

```
gcc -o foo 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.

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

W przypadku GNU/Linux:

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

Automatyzacja procesu kompilacji

W przypadku większych aplikacji dużym ułatwieniem budowy plików wynikowych są systemy automatyzujące proces kompilacji.

Tego typu rozwiązania pozwalają na budowanie aplikacji z minimalną lub bez interakcji programisty i bez użycia komputera programisty.

Automatyzacja budowania obejmuje dodatkowo konfigurowanie systemu kompilacji, a także powstałej aplikacji.

Istnieje wiele systemów zarządzających kompilacją, kilka przykładów:

- **Apache Ant** – system budowy do aplikacji Java, używa plików XML.
- **Apache Maven** – narzędzie do zarządzania zależnościami w trakcie budowy aplikacji.
- **Make** – jeden z pierwszych systemów zarządzania kompilacją, ma wiele odmian.
- **Ninja** – system charakteryzujący się dużą szybkością, jego pliki wejściowe są generowane przez system kompilacji wyższego poziomu.
- **CMake** – narzędzie wieloplatformowe, główną cechą jest niezależność od używanego kompilatora oraz platformy sprzętowej, nie kompiluje programu samodzielnie, lecz tworzy pliki z regułami kompilacji.
- **GNU build system** (Autotools) – zestaw narzędzi od GNU, pozwala na budowę przenośnych pakietów oprogramowania dla wielu systemów operacyjnych z rodziny UNIX i Linux.
- **Meson** – narzędzie do automatyzacji tworzenia oprogramowania do budowania bazy kodu, minimalizuje dane wymagane do skonfigurowania najczęstszych operacji.
- **qmake** – narzędzie do automatyzacji kompilacji oprogramowania, które generuje pliki `Makefile`.

Bardzo popularnym systemem budowy aplikacji jest **Make**. Make nadaje się również do innych prac, które wymagają przetwarzania wielu plików zależnych od siebie.

Program przetwarza plik reguł `Makefile` i na tej podstawie kompiluje pliki aplikacji. Potrafi też stwierdzić, które pliki wymagają rekompilacji. Dzięki temu nie ma potrzeby kompilacji całego projektu.

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

Przykładowy `Makefile` (do wcięć należy używać **tabulatorów**):

```
.PHONY: all clean

all: foo.exe

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

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

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

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

`.PHONY` oznacza cele, które nie są nazwami plików. Bez tego istnienie w katalogu pliku o nazwie `clean` sprawiłoby, że `make clean` nic by nie zrobił.

W przypadku GNU/Linux:

```
.PHONY: all clean

all: foo

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

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

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

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

CMake

Ten sam projekt opisany dla **CMake** – plik `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.10)
project(foo C)

add_executable(foo foo.c foofun.c)
```

Zależności od plików nagłówkowych CMake wykrywa samodzielnie. Na podstawie tego opisu generuje pliki dla wybranego narzędzia (`Makefile`, `build.ninja`, rozwiązanie Visual Studio), a następnie je uruchamia:

```
cmake -B build                  # konfiguracja, katalog build
cmake -B build -G Ninja         # to samo z generatorem Ninja
cmake --build build             # kompilacja
```

Visual Studio i CLion potrafią otworzyć projekt CMake bezpośrednio, dla Code::Blocks CMake generuje plik projektu.

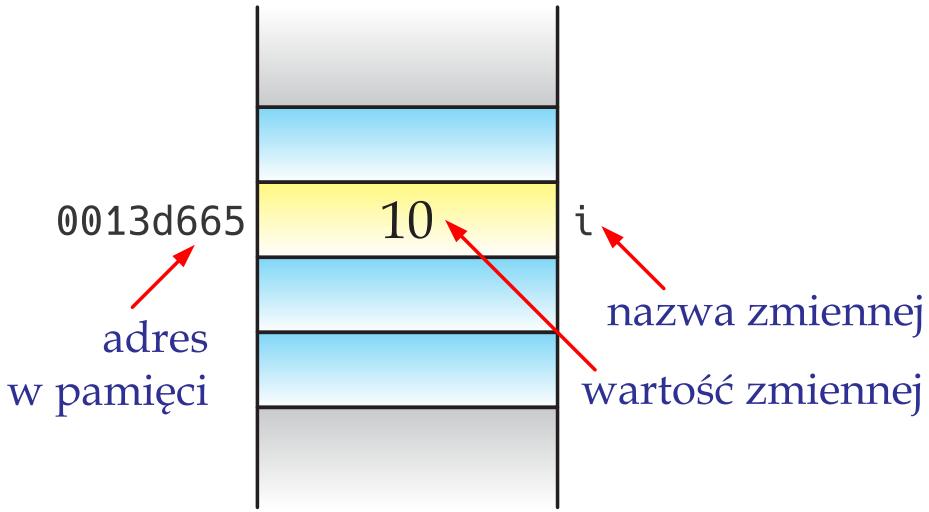
1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

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.

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

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



Zmienna wskaźnikowa

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.

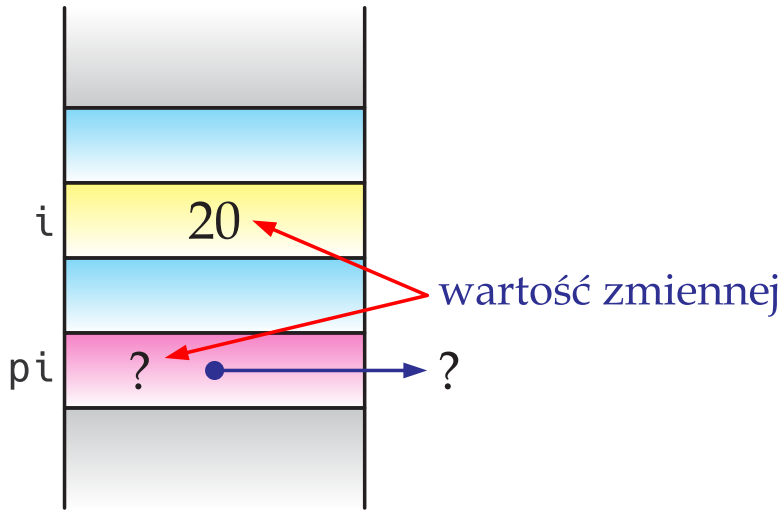
Deklaracja i definicja zmiennej wskaźnikowej

Deklarowana zmienna będzie wskaźnikiem, kompilator będzie znał jej rozmiar.

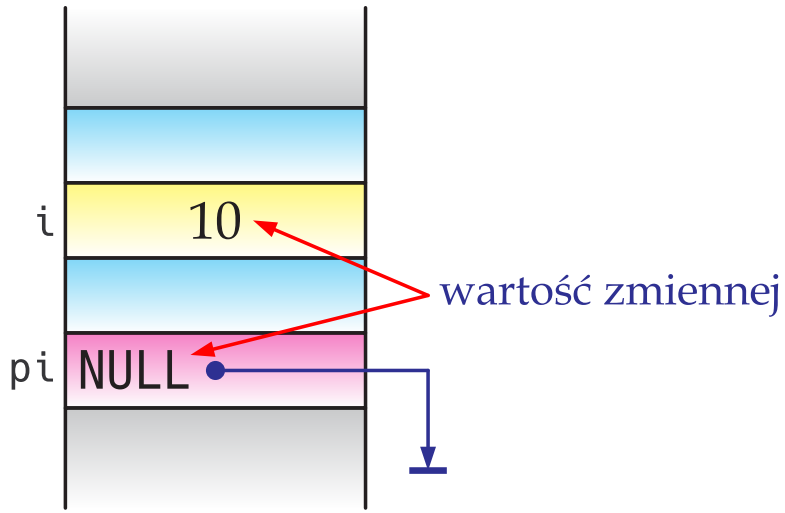
```
int *pi;
```

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 nieokreślona, a zmienna globalna lub statyczna jest inicjowana wartością pustą.

W pliku nagłówkowym `stddef.h` (a także m.in. w `stdio.h`, `stdlib.h`, `string.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`, `0L` lub `((void *)0)` (tak robi np. biblioteka glibc). Można zatem posługiwać się wartością `0` zamiast `NULL`.

W języku C praktykuje się stosowanie `NULL`, od standardu C23 dostępne jest też słowo kluczowe `nullptr`.

W języku C++ (od C++11) stosuje się `nullptr`, wcześniej praktykowano wartość `0`.

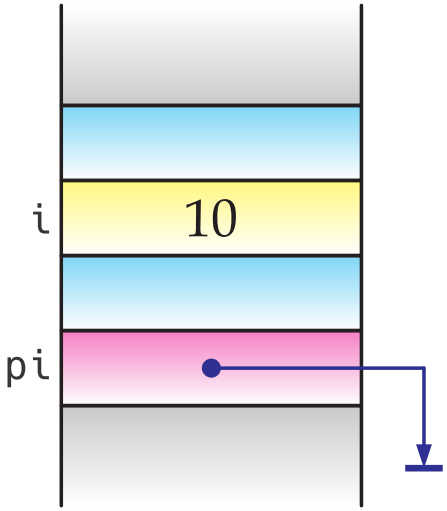
Dobrą praktyką jest jawne inicjowanie zmiennych wskaźnikowych oraz ustawianie na wartość pustą dla wskaźników niezakotwiczonych. Można wtedy sprawdzić, czy zmienna nie jest pusta.

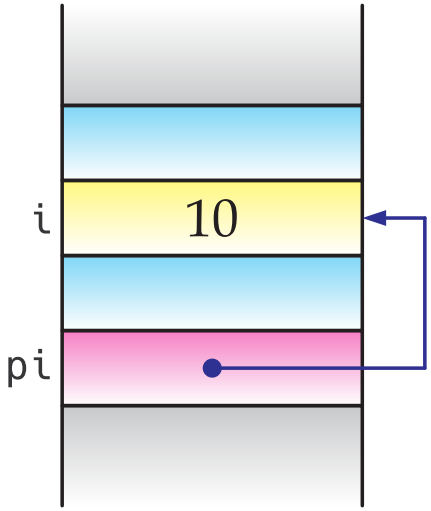
```
if(pi != NULL)
{
  ...
}
```

Przypisywanie wartości i odwołanie

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

pi = &i;
```

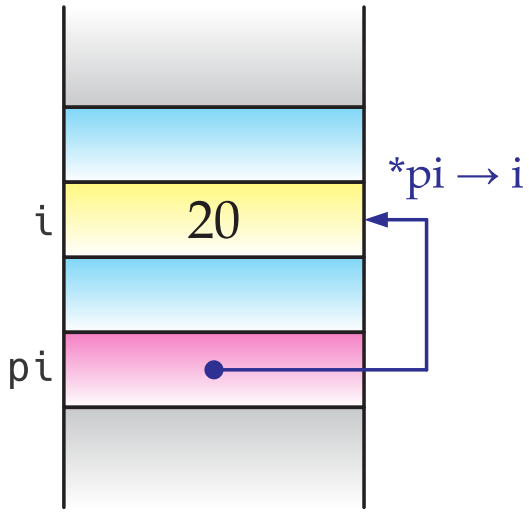
Wyrażenie `&i` **pobiera adres** (lokalizuje w pamięci) zmiennej `i`.





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

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

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



Wskaźnik typu `void *`

Typ `void *` oznacza wskaźnik niezwiązany z żadnym typem.

Tego typu wskaźnik może wskazywać daną dowolnego typu.

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

void *p;

p = &i;

p = &c;

p = &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 *p;

p = &i;
printf("i = %d\n", *(int *)p);

p = &c;
printf("c = %c\n", *(char *)p);

p = &f;
printf("f = %f\n", *(float *)p);
```

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

Zmienne referencyjne **nie występują w języku C**, są **dostępne w języku C++**.

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

printf("i = %d\n", i);
printf("*pi = %d\n", *pi);

++(*pi);

printf("i = %d\n", i);
printf("*pi = %d\n", *pi);
```

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

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

++ri;

cout << "i = " << i << endl;
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 funkcją `malloc()`, `calloc()` lub `realloc()`.

Blok może zostać **przeniesiony** w inne miejsce pamięci – zawartość jest kopiowana, a stary wskaźnik traci ważność. Należy korzystać wyłącznie ze zwróconego adresu.

Jeżeli nowy rozmiar jest większy od poprzednio przydzielonego, dodatkowe bajty mają nieokreśloną wartość. W razie niepowodzenia funkcja zwraca `NULL`, a stary blok pozostaje nienaruszony.

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

Wynik `realloc()` należy przypisywać do zmiennej pomocniczej:

```
int *tmp = realloc(ptr, n * sizeof(int));

if(tmp == NULL)
{
  ...  ← ptr nadal ważny - można go użyć lub zwolnić
}
else
  ptr = tmp;
```

Zapis `ptr = realloc(ptr, n)` w razie błędu nadpisuje jedyny wskaźnik do starego bloku wartością `NULL` – pamięć wycieka.

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("*ptr = %d", ++(*ptr));

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

```cpp
int *ptr = nullptr;

ptr = new (nothrow) int;

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

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

  delete ptr;
  ptr = nullptr;
}
```

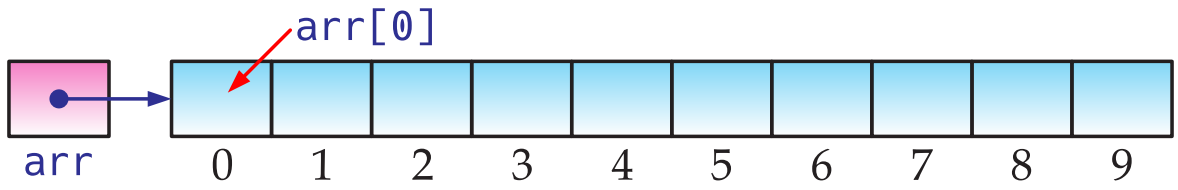
Obecnie częściej stosuje się wyjątki.

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

  *ptr = 10;

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

  delete ptr;
  ptr = nullptr;
}

catch(const bad_alloc &)
{
  ...  ← obsługa wyjątku (brak pamięci)
}
```

Tablice a wskaźniki

Nazwa tablicy w wyrażeniu jest zamieniana na wskaźnik na jej pierwszy element. Sama tablica wskaźnikiem nie jest – dlatego `sizeof(arr)` daje rozmiar całej tablicy, a nie wskaźnika.

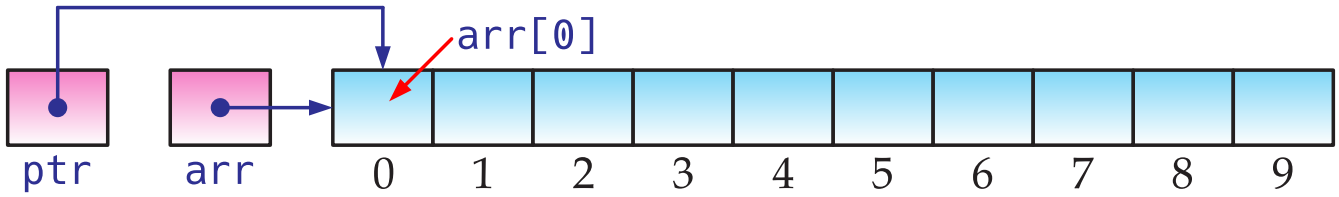
```
int arr[10];
```



Przypisania równoznaczne.

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

ptr = arr;
```

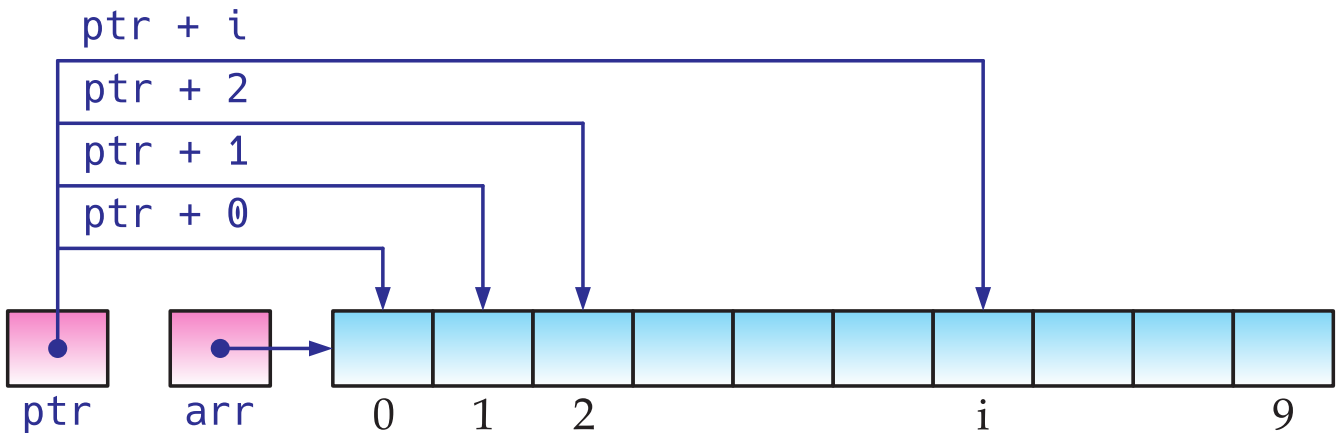
```c
int arr[10];
int *ptr;

ptr = &arr[0];
```



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

arr[i] = 13;
```

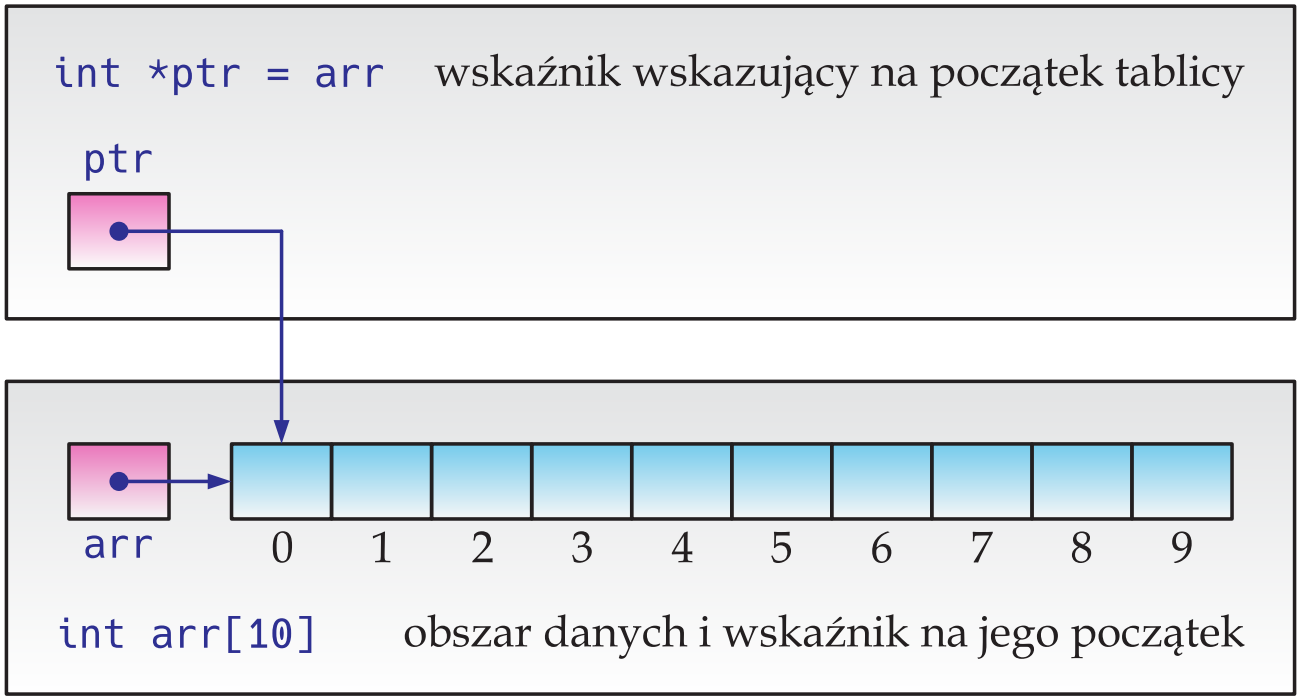
```c
*arr = 3;
*(arr + 1) = 7;
*(arr + 2) = 9;

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

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

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



Wskaźnik to nie tablica!

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



Nazwa tablicy nie jest zmienną wskaźnikową, dlatego **nie można jej modyfikować** (np. `arr++` jest błędem). Zwykłe wskaźniki można.

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

ptr = arr + 3;  ← jest poprawne
ptr++;          ← jest poprawne

arr = ptr;      ← błąd
arr++;          ← błąd
```

Odwołanie:

```
arr[i]
```

jest tożsame z:

```
*(arr + i)
```

a dodawanie jest przemienne, zatem:

```
*(i + arr)
```

Zagadka – czy odwołanie:

```
*(i + arr)
```

można zapisać w postaci:

```
i[arr]
```

Wskaźniki a tablice wielowymiarowe

Przykładowa deklaracja tablicy dwuwymiarowej:

```
int arr[4][2];
```

Zmienna `arr`, będąca nazwą tablicy, jest w wyrażeniu zamieniana na adres jej pierwszego elementu. W tym przypadku pierwszy element tablicy `arr` jest tablicą składającą się z dwóch wartości typu `int`, a zatem `arr` jest adresem tablicy zawierającej dwie liczby całkowite.

Ponieważ `arr` jest adresem pierwszego elementu tablicy, `arr` jest równe `&arr[0]`. Z kolei `arr[0]` jest tablicą składającą się z dwóch liczb całkowitych, tak więc `arr[0]` jest równoważne `&arr[0][0]`, adresowi jej pierwszego elementu (liczby typu `int`).

W skrócie, `arr[0]` jest adresem obiektu o rozmiarze `int`, a `arr` jest adresem obiektu o rozmiarze dwóch wartości typu `int`.

Ponieważ liczba całkowita, jak i tablica dwóch liczb całkowitych rozpoczynają się w tym samym miejscu, oba wskaźniki: `arr` i `arr[0]` mają tę samą wartość liczbową.

Dodanie `1` do wskaźnika lub adresu zwiększa jego wartość o rozmiar wskazywanego obiektu.

Tu `arr` i `arr[0]` różnią się, ponieważ `arr` wskazuje obiekt o rozmiarze dwóch wartości `int`, a `arr[0]` obiekt o rozmiarze jednej wartości `int`.

Wyrażenia `arr + 1` i `arr[0] + 1` nie mają tej samej wartości.

Dereferencja wskaźnika (operator `*` lub `[]`) daje wartość wskazywanego obiektu.

Ponieważ `arr[0]` jest adresem elementu `arr[0][0]`, wyrażenie `*(arr[0])` oznacza wartość przechowywaną w tym elemencie.

Podobnie `arr` przechowuje adres podtablicy `arr[0]`, która jest adresem elementu `arr[0][0]`, a zatem `*arr` to `&arr[0][0]`.

Zastosowanie operatora dereferencji do obu wyrażeń prowadzi do wniosku, że `**arr` jest równoważne `*&arr[0][0]`, czyli `arr[0][0]`, wartości typu `int`.

Innymi słowy, aby z `arr` otrzymać zwykłą wartość (a nie adres), trzeba dokonać dwukrotnej dereferencji. Nie oznacza to jednak, że `arr` jest typu `int **` – jest to wskaźnik na tablicę dwóch liczb, `int (*)[2]`.

Program wyświetlający adresy:

```
int arr[4][2] = {{2, 4}, {6, 8}, {1, 3}, {5, 7}};

printf("   arr = %p,    arr + 1 = %p\n",    (void *)arr,    (void *)(arr + 1));
printf("arr[0] = %p, arr[0] + 1 = %p\n", (void *)arr[0], (void *)(arr[0] + 1));
printf("  *arr = %p,   *arr + 1 = %p\n",   (void *)*arr,   (void *)(*arr + 1));

printf("arr[0][0] = %d\n", arr[0][0]);
printf("  *arr[0] = %d\n",   *arr[0]);
printf("    **arr = %d\n",     **arr);

printf("        arr[2][1] = %d\n",         arr[2][1]);
printf("*(*(arr + 2) + 1) = %d\n", *(*(arr + 2) + 1));
```

Przykładowy rezultat działania programu:

```
   arr = 0x7ffe543f8f60,    arr + 1 = 0x7ffe543f8f68
arr[0] = 0x7ffe543f8f60, arr[0] + 1 = 0x7ffe543f8f64
  *arr = 0x7ffe543f8f60,   *arr + 1 = 0x7ffe543f8f64

arr[0][0] = 2
  *arr[0] = 2
    **arr = 2

        arr[2][1] = 3
*(*(arr + 2) + 1) = 3
```

Adresy mogą być inne przy każdym uruchomieniu programu, ale ogólna zależność ilustrowana programem będzie taka sama.

Adres tablicy dwuwymiarowej `arr` oraz adres jednowymiarowej tablicy `arr[0]` są identyczne.

Każdy z nich jest adresem pierwszego elementu odpowiedniej tablicy, który jest liczbowo równoważny `&arr[0][0]`.

Typ `int` ma **4** bajty. `arr[0]` i `*arr` wskazują na **4**-bajtowy obiekt danych. Dodanie **1** do każdego z nich powinno zwiększyć wartość o **4** (szesnastkowo `0x60 + 0x4` wynosi `0x64`).

Nazwa `arr` jest adresem tablicy składającej się z dwóch wartości `int`, a więc wskazuje ona obiekt danych o długości **8** bajtów. Dodanie **1** do `arr` powinno zwiększyć adres o **8** (szesnastkowo `0x60 + 0x8` wynosi `0x68`).

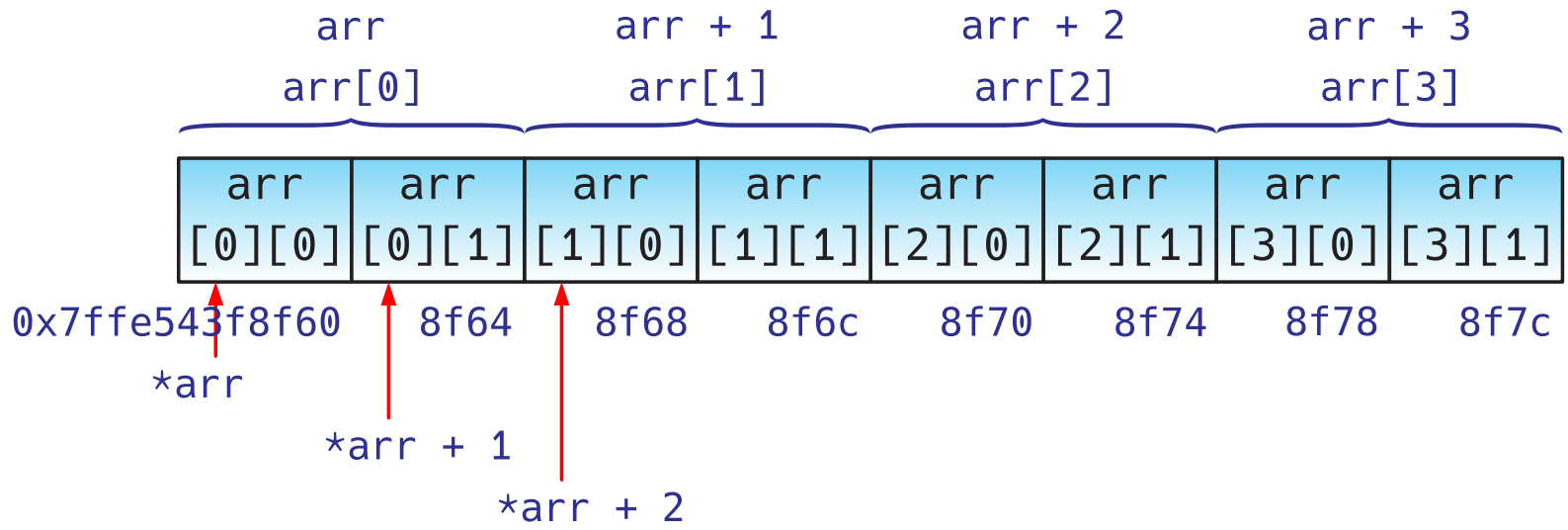
Wartości `arr[0]` i `*arr` są identyczne. Należy dokonać podwójnej dereferencji nazwy dwuwymiarowej tablicy, aby otrzymać wartość przechowywanego w tablicy elementu.

Można zastosować dwa razy operator dereferencji `*` lub skorzystać dwukrotnie z operatora `[]` (można je mieszać).

Odpowiednikiem notacji `arr[2][1]` jest `*(*(arr + 2) + 1)`.

Pojedyncze składniki dereferencji z użyciem operatora `*`:

- `arr` – adres pierwszego elementu złożonego z dwóch liczb całkowitych,
- `arr + 2` – adres trzeciego elementu złożonego z dwóch liczb całkowitych,
- `*(arr + 2)` – trzeci element tablicy, czyli dwuelementowa podtablica (typu `int[2]`), która w wyrażeniu jest zamieniana na adres jej pierwszego elementu (typu `int *`),
- `*(arr + 2) + 1` – adres drugiego elementu tej podtablicy (także typu `int *`),
- `*(*(arr + 2) + 1)` – w końcu druga wartość typu `int` w trzecim rzędzie (`arr[2][1]`).



Wskaźniki do tablic wielowymiarowych

Zmienną wskaźnikową `ptr` zgodną z tablicą `arr`, przydatną na przykład w funkcji przetwarzającej tablice dwuwymiarowe, można zadeklarować tak:

```
int (* ptr)[2];
```

Zmienna `ptr` jest wskaźnikiem do tablicy składającej się z dwóch wartości typu `int`. Nawiasy są niezbędne, ponieważ operator `[]` ma wyższy priorytet niż operator `*`.

Przykład użycia wskaźnika:

```
int arr[4][2] = {{2, 4}, {6, 8}, {1, 3}, {5, 7}};

int (*ptr)[2];
ptr = arr;

printf("   ptr = %p,    ptr + 1 = %p\n",    (void *)ptr,    (void *)(ptr + 1));
printf("ptr[0] = %p, ptr[0] + 1 = %p\n", (void *)ptr[0], (void *)(ptr[0] + 1));
printf("  *ptr = %p,   *ptr + 1 = %p\n",   (void *)*ptr,   (void *)(*ptr + 1));

printf("ptr[0][0] = %d\n", ptr[0][0]);
printf("  *ptr[0] = %d\n",   *ptr[0]);
printf("    **ptr = %d\n",     **ptr);

printf("        ptr[2][1] = %d\n",         ptr[2][1]);
printf("*(*(ptr + 2) + 1) = %d\n", *(*(ptr + 2) + 1));
```

Przykładowy rezultat działania programu:

```
   ptr = 0x7ffe543f8f60,    ptr + 1 = 0x7ffe543f8f68
ptr[0] = 0x7ffe543f8f60, ptr[0] + 1 = 0x7ffe543f8f64
  *ptr = 0x7ffe543f8f60,   *ptr + 1 = 0x7ffe543f8f64

ptr[0][0] = 2
  *ptr[0] = 2
    **ptr = 2

        ptr[2][1] = 3
*(*(ptr + 2) + 1) = 3
```

Adresy mogą być inne przy każdym uruchomieniu programu, ale ogólna zależność ilustrowana programem będzie taka sama.

Można stosować notację `ptr[2][1]`, gdy `ptr` jest tylko wskaźnikiem, a nie tablicą.

Za pomocą notacji tablicowej lub wskaźnikowej można przedstawiać pojedyncze elementy, niezależnie od tego, czy korzystamy ze wskaźnika, czy z nazwy tablicy.

Zapis `arr[m][n]` jest równoważny z `*(*(arr + m) + n)` natomiast `ptr[m][n]` jest równoważny z `*(*(ptr + m) + n)`.

Zgodność wskaźników

Zasady obowiązujące podczas przypisywania wskaźników są ściślejsze niż te, które odnoszą się do wartości numerycznych.

Można przypisać wartość `int` do zmiennej `double` bez stosowania konwersji typu, co jest niedozwolone w przypadku wskaźników.

```
int n = 5;
double d;

int *pn = &n;
double *pd = &d;

d = n;     ← niejawna konwersja typów
pd = pn;   ← błąd podczas kompilacji
```

Takie ograniczenia dotyczą też typów związanych z tablicami.

```
int *pt;
int **pp;
int (*pa)[3];
int arr1[2][3];
int arr2[3][2];
```

Wówczas:

```
 pt = &arr1[0][0];  ← oba wskaźniki do int
 pt = arr1[0];      ← oba wskaźniki do int
 pt = arr1;         ← instrukcja nieprawidłowa
 pa = arr1;         ← oba wskaźniki do int[3]
 pa = arr2;         ← instrukcja nieprawidłowa
 pp = &pt;          ← oba wskaźniki do int *
*pp = arr2[0];      ← oba wskaźniki do int
 pp = arr2;         ← instrukcja nieprawidłowa
```

Dynamiczna alokacja tablic jednowymiarowych

Tablice **automatyczne** w języku C są **statyczne**. Oznacza to, że ich rozmiar powinien być **znany** w trakcie kompilacji i **nie może być zmieniany** w trakcie działania programu.

Z tego powodu często zdarzają się sytuacje, w których wielkość tablicy jest niewystarczająca lub zużywa się więcej pamięci niż jest to konieczne.

Aby rozwiązać ten problem, można stosować tablice dynamiczne.

Tablica dynamiczna jest przydzielana w trakcie działania programu i może być zmieniana.

Do tego celu można użyć funkcji alokowania, realokowania i zwalniania pamięci. Funkcje powinny być dostępne po dołączeniu pliku nagłówkowego `stdlib.h`.

Przykład z zastosowaniem funkcji `malloc()` z notacją tablicową (sterta):

```
int i, size = 10;
int *ptr;

ptr = (int *) malloc(size * sizeof(int));

if(ptr == NULL)
{
  printf("Pamiec nie zaalokowana.\n");
  return 1;
}

for(i = 0; i < size; ++i)
  ptr[i] = i + 1;

for(i = 0; i < size; ++i)
  printf("%d ", ptr[i]);

free(ptr);
```

Przykład z zastosowaniem funkcji `calloc()` z notacją tablicową (sterta):

```
int i, size = 10;
int *ptr;

ptr = (int *) calloc(size, sizeof(int));

if(ptr == NULL)
{
  printf("Pamiec nie zaalokowana.\n");
  return 1;
}

for(i = 0; i < size; ++i)
  ptr[i] = i + 1;

for(i = 0; i < size; ++i)
  printf("%d ", ptr[i]);

free(ptr);
```

Tablica VLA (stos):

```
int i, size = 10;
int arr[size];

for(i = 0; i < size; ++i)
  arr[i] = i + 1;

for(i = 0; i < size; ++i)
  printf("%d ", arr[i]);
```

Dynamiczna alokacja tablic dwuwymiarowych

W przypadku tablic wielowymiarowych jest więcej możliwości.

Przykład z użyciem wskaźnika i arytmetyki wskaźnikowej:

```
int i, j, row = 3, col = 4;
int *ptr;

ptr = malloc((row * col) * sizeof(int));

for(i = 0; i < row * col; i++)
  ptr[i] = i + 1;

for(i = 0; i < row; i++)
  for (j = 0; j < col; j++)
    printf("%d ", ptr[i * col + j]);

free(ptr);
```

Przykład z użyciem wskaźnika do wskaźnika:

```
int i, j, row = 3, col = 4, count = 0;
int **ptr;

ptr = (int **) malloc(row * sizeof(int *));

for(i = 0; i < row; i++)
  ptr[i] = (int *) malloc(col * sizeof(int));

for(i = 0; i < row; i++)
  for(j = 0; j < col; j++)
    ptr[i][j] = ++count; // *(*(ptr + i) + j) = ...

for(i = 0; i < row; i++)
  for(j = 0; j < col; j++)
    printf("%d ", ptr[i][j]);

for(i = 0; i < row; i++)
  free(ptr[i]);

free(ptr);
```

Przykład z użyciem tablicy wskaźników:

```
int i, j, row = 3, col = 4, count = 0;
int *arr[row];

for(i = 0; i < row; i++)
  arr[i] = (int *) malloc(col * sizeof(int));

for(i = 0; i < row; i++)
  for(j = 0; j < col; j++)
    arr[i][j] = ++count; // *(*(arr + i) + j) = ...

for(i = 0; i < row; i++)
  for(j = 0; j < col; j++)
    printf("%d ", arr[i][j]);

for(i = 0; i < row; i++)
  free(arr[i]);
```

Przykład z użyciem wskaźnika do tablic:

```
int i, j, row = 3, col = 4, count = 0;
int (*arr)[col];

arr = malloc(row * sizeof(*arr));

for(i = 0; i < row; i++)
  for(j = 0; j < col; j++)
    arr[i][j] = ++count; // *(*(arr + i) + j) = ...

for(i = 0; i < row; i++)
  for(j = 0; j < col; j++)
    printf("%d ", arr[i][j]);

free(arr);
```

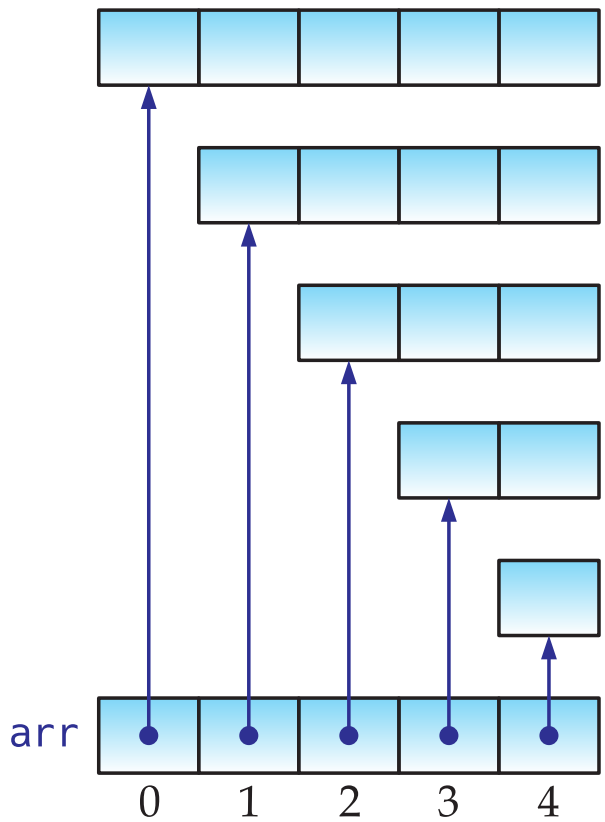
Tablica VLA:

```c
int i, j, row = 3, col = 4, count = 0;
int arr[row][col];

for(i = 0; i < row; i++)
  for(j = 0; j < col; j++)
    arr[i][j] = ++count;

for(i = 0; i < row; i++)
  for(j = 0; j < col; j++)
    printf("%d ", arr[i][j]);
```

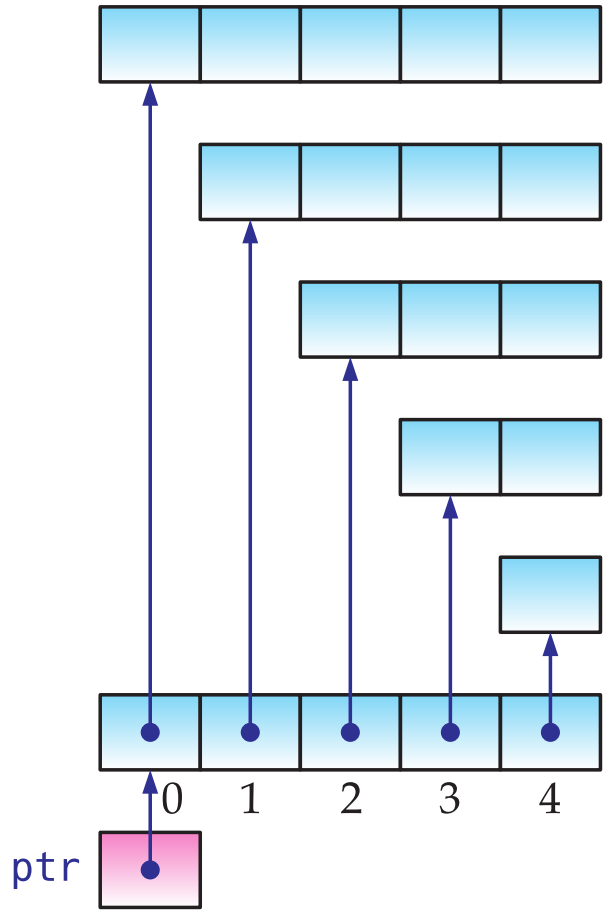
Tablice o zmiennej liczbie elementów

Używanie wskaźników pozwala na deklarację tablic o różnej liczbie elementów w wierszach. Takie tablice są przydatne do reprezentowania np. macierzy symetrycznych.

```
int i, j, size = 5, count = 0;
int *arr[size];

for(i = 0; i < size; i++)
  arr[i] = (int *) malloc((size - i) * sizeof(int));

for(i = 0; i < size; i++)
  for(j = 0; j < (size - i); j++)
    arr[i][j] = ++count;

for(i = 0; i < size; i++)
  for(j = 0; j < (size - i); j++)
    printf("%d ", arr[i][j]);

for (int i = 0; i < size; ++i)
  free(arr[i]);
```



```c
int i, j, size = 5, count = 0;
int **ptr;

ptr = (int **) malloc(size * sizeof(int *));

for(i = 0; i < size; i++)
  ptr[i] = (int *) malloc((size - i) * sizeof(int));

for(i = 0; i < size; i++)
  for(j = 0; j < (size - i); j++)
    ptr[i][j] = ++count;

for(i = 0; i < size; i++)
  for(j = 0; j < (size - i); j++)
    printf("%d ", ptr[i][j]);

for (int i = 0; i < size; ++i)
  free(ptr[i]);

free(ptr);
```



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++);
}
```

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

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`).

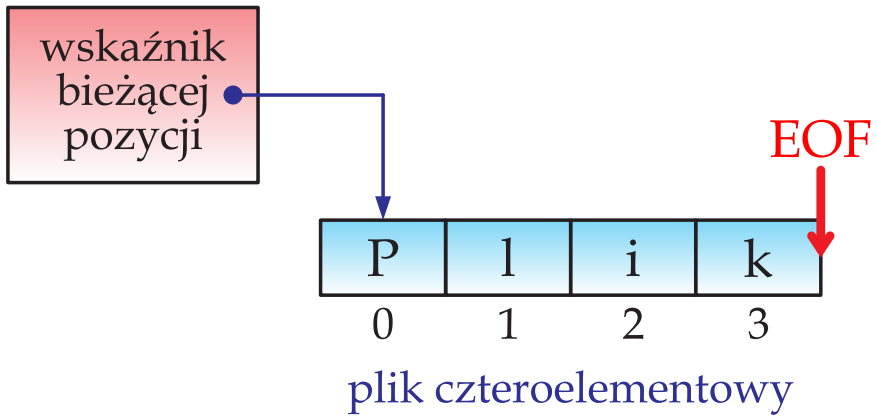
Można też korzystać z funkcji niższego poziomu (np. `io.h`) lub napisać własne funkcje.

Plik jest reprezentowany przez strumień znaków (bajtów) o zmiennej długości. W pliku nie ma żadnego znacznika końca – system zna jego długość. Wartość `EOF` zwracają funkcje odczytu, gdy danych już nie ma (lub wystąpił błąd).

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.



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( )`.

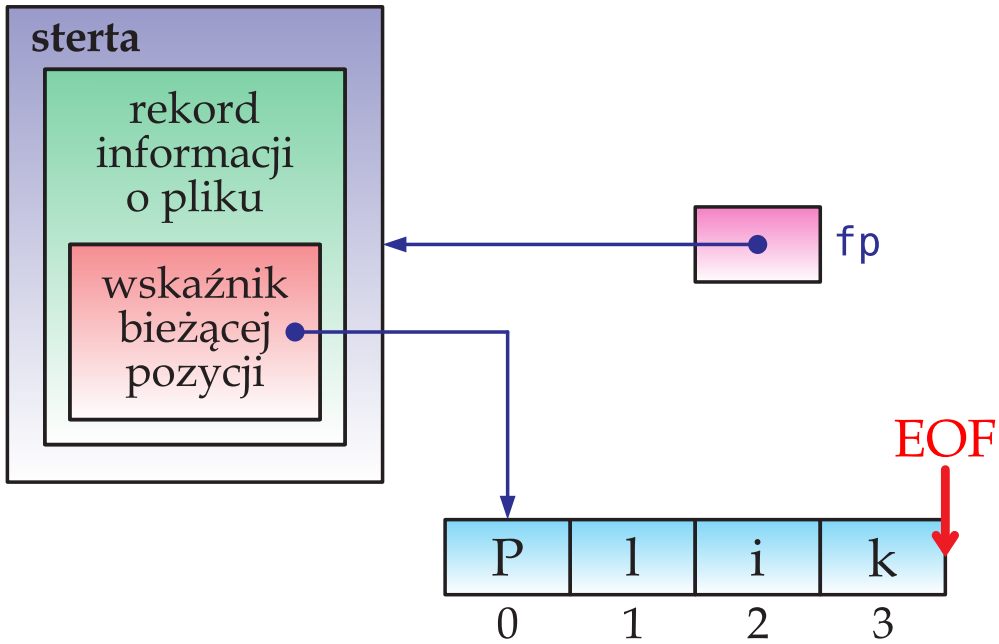
```
FILE * fopen(const char *filename, const char *mode);
```

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 reprezentację fizycznego pliku.

Jeżeli pliku nie udało się otworzyć, rezultatem funkcji jest `NULL`.

```c
#include <stdio.h>

FILE *fp = NULL;
fp = fopen("file.txt", "r");

if(fp != NULL)
  → poprawne otwarcie
else
  → otwarcie nieudane
```



Specyfikacja sposobu otwarcia pliku:

- `b` – otwarcie w trybie binarnym,
- brak `b` – otwarcie w trybie tekstowym (litera `t` to rozszerzenie Microsoft, inne biblioteki ją ignorują).

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,
- `w+` – 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()`.

W Windows (MSVC, MinGW), jeżeli w trybie otwarcia nie podano `t` ani `b`, przyjmowany jest tryb zgodnie z wartością globalnej zmiennej `_fmode`. W Linuksie tej zmiennej nie ma – tryb tekstowy i binarny niczym się tam nie różnią.

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", "r");
```

Otwarcie pliku `image.bmp` jako pliku **binarnego** do **odczytu**:

```
fp = fopen("image.bmp", "rb");
```

Otwarcie pliku `report.txt` jako pliku tekstowego do **zapisu**, wskaźnik pliku ustawiany jest w pozycji końcowej, jeżeli nie istnieje, tworzony jest nowy plik:

```
fp = fopen("report.txt", "a");
```

Otwarcie pliku `image.jpg` jako pliku binarnego, do **zapisu** i **odczytu**, jeżeli plik istnieje zostanie nadpisany:

```
fp = fopen("image.jpg", "w+b");
```

```c
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.

```c
#include <stdio.h>

FILE *fp = NULL;
fp = fopen("file.txt", "r");

if(fp != NULL)
{
  → otwarcie poprawne, 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`:

```c
#include <stdio.h>

FILE *fp = NULL;

if((fp = fopen("data.txt", "r")) != NULL)
{
  int c;
  c = fgetc(fp);
  printf("Przeczytano znak %c.", c);
  fclose(fp);
}
```

```c
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`:

```c
#include <stdio.h>

FILE *fp = NULL;

if((fp = fopen("data.txt", "w")) != NULL)
{
  fputc('C', fp);
  fclose(fp);
}
```

Przetwarzanie sekwencyjne

```c
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** dopiero wtedy, gdy **próba odczytu nie powiodła się** z powodu braku danych. Samo dojście do końca pliku nie wystarcza – `feof()` zwraca prawdę dopiero po nieudanym odczycie.

„Why is `while(!feof(file))` always wrong?”

Pytanie o tym tytule jest jednym z najczęściej odwiedzanych w serwisie **Stack Overflow** w kategorii języka C. Częsty błąd – pętla sterowana funkcją `feof( )`:

```
FILE *fp;
long int counter = 0;

if((fp = fopen("data.txt", "r")) != NULL)
{
  while(!feof(fp))
  {
    putchar(fgetc(fp));
    counter++;
  }

  fclose(fp);
  printf("\nLiczba znakow w pliku: %ld.", counter - 1);
}
```

Dla pliku zawierającego trzy znaki `abc` program wypisze:

```
abc
Liczba znakow w pliku: 3.
```

Wynik wygląda poprawnie, ale tylko dzięki korekcie `counter - 1`, a za `c` wypisany został niewidoczny bajt `0xFF`. Pętla wykonała się cztery razy:

- `feof()` nie sprawdza, czy w pliku zostały jeszcze dane, tylko czy **poprzedni** odczyt trafił na koniec pliku,
- po odczytaniu `c` flaga końca nie jest jeszcze ustawiona, więc pętla wykonuje się kolejny raz – `fgetc()` zwraca `EOF`, a `putchar()` wypisuje go jako znak,
- przy **błędzie odczytu** (np. odłączony nośnik) `fgetc()` stale zwraca `EOF`, ale `feof()` zwraca fałsz – pętla nigdy się nie kończy.

Poprawnie – odczyt sterowany wartością zwracaną przez `fgetc()`, a `feof()` lub `ferror()` dopiero po pętli, by ustalić przyczynę jej zakończenia:

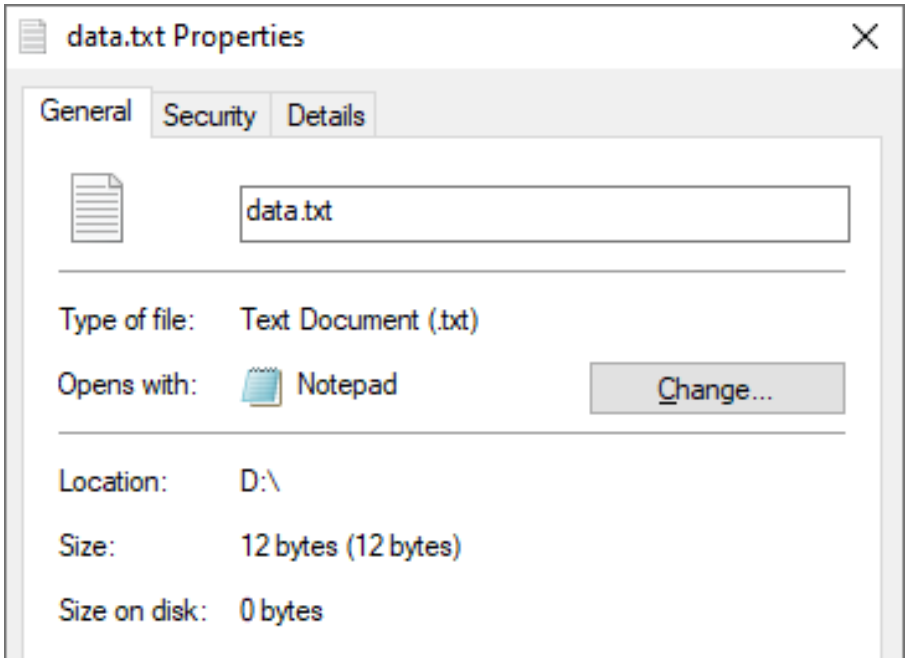
```
FILE *fp;
long int counter = 0;
int c;

if((fp = fopen("data.txt", "r")) != NULL)
{
  while((c = fgetc(fp)) != EOF)
  {
    putchar(c);
    counter++;
  }

  if(!feof(fp))
    printf("\nBlad odczytu pliku.");

  fclose(fp);
  printf("\nLiczba znakow w pliku: %ld.", counter);
}
```

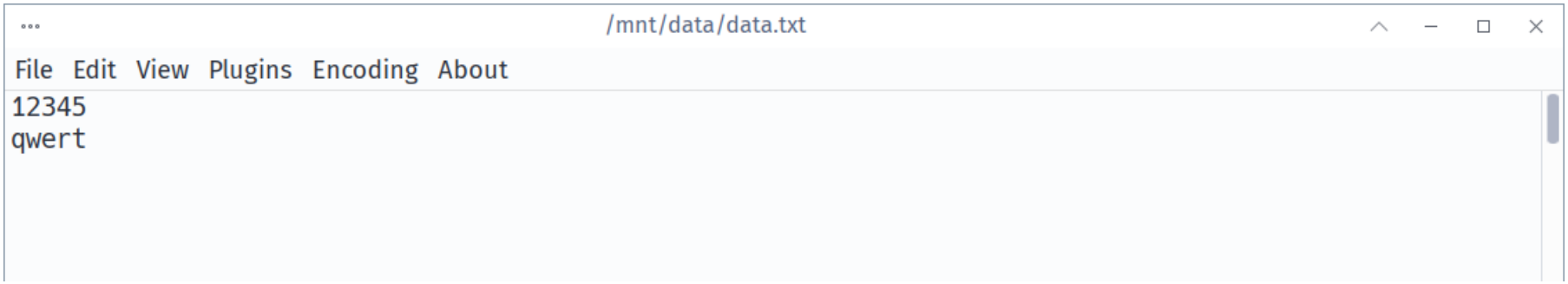
Rozmiar pliku, znak końca linii i problemy

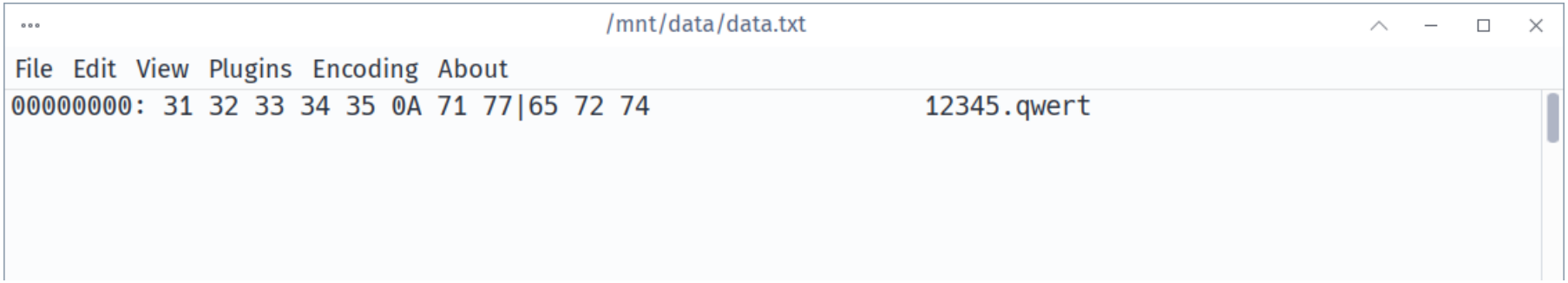
Wyznaczanie rozmiaru pliku (trochę inaczej):

```
if((fp = fopen("data.txt", "r")) != NULL)
{
  for(counter = 0; fgetc(fp) != EOF; counter++);
  fclose(fp);
  printf("Rozmiar pliku: %ld bajtow.", counter);
}
```

```
Rozmiar pliku: 11 bajtow.
```

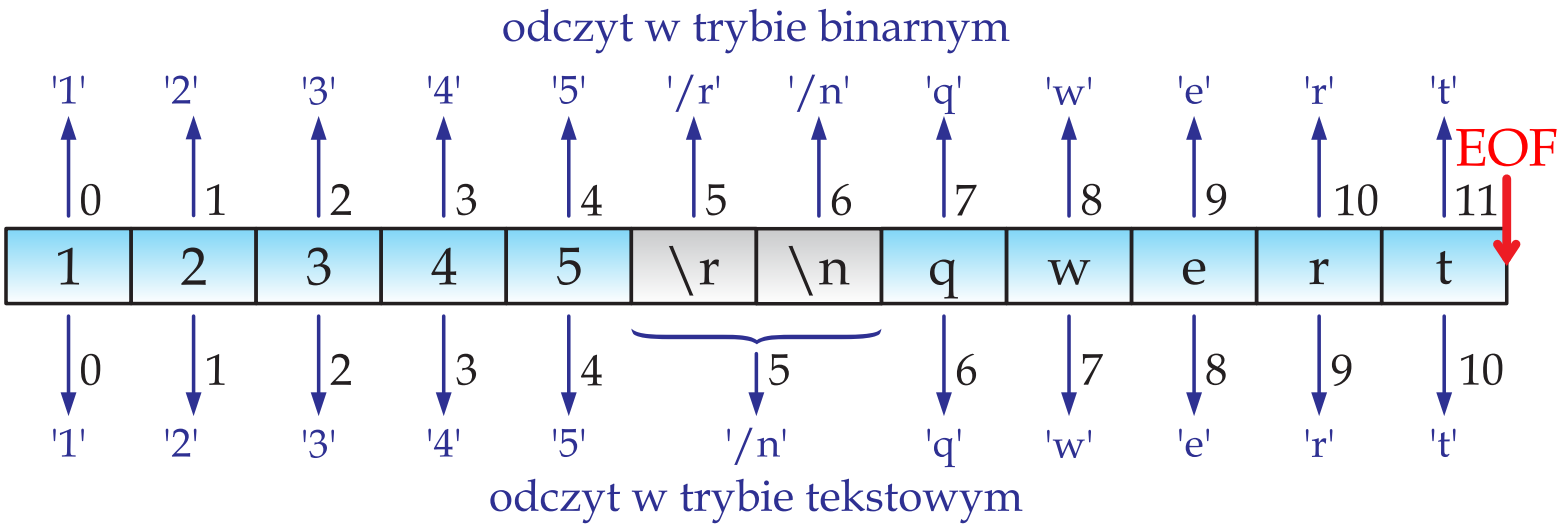






Przyczyną wadliwego działania programu są konwersje znaczników końca linii w trybie tekstowym.

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);
}
```

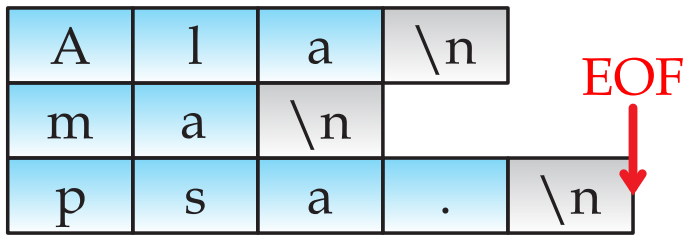
Przetwarzanie plików linia po linii

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:

Przy przetwarzaniu linia po linii:



Odczyt linii tekstu z pliku realizuje 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 **wartość nieujemna**, w przypadku gdy zapis zakończył się sukcesem, lub `EOF`, gdy wystąpił błąd.

```c
#include <stdio.h>

int main()
{
  FILE *fp = NULL;

 if((fp = fopen( "data.txt", "w")) != NULL)
  {
    fputs("Jestem C", fp);
    fputs(", jezyk C.", fp);
    fclose(fp);
  }

  return 0;
}
```

```
Jestem C, jezyk C.
```

```c
#include <stdio.h>

int main()
{
  FILE *fp = NULL;

  if((fp = fopen("data.txt", "w")) != NULL)
  {
    fputs("Jestem C", fp);
    fputs(",\njezyk C.\nKoniec.", fp);
    fclose(fp);
  }

  return 0;
}
```

```
Jestem C,
jezyk C.
Koniec.
```

```c
int fprintf(FILE *stream, const char *format, ...);
```

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 znaków**, w przypadku gdy zapis zakończył się sukcesem, lub **wartość ujemna**, gdy wystąpił błąd.

Wszystkie zasady formatowania znane z wykorzystania funkcji `printf()` obowiązują dla funkcji `fprintf()`.

```c
#include <stdio.h>

int main()
{
  FILE *fp = NULL;

  if((fp = fopen("data.txt", "w")) != NULL)
  {
    char brand[80] = "Fiat";
    char model[80] = "126p";
    int year = 1977;
    float mileage = 128.23;

    fprintf(fp, "Dane samochodu:\n%s\n%s\n%d\n%g", brand, model,
      year, mileage);

    fclose(fp);
  }

  return 0;
}
```

```
Dane samochodu:
Fiat
126p
1977
128.23
```

```c
#include <stdio.h>

int main()
{
  FILE *fp = NULL;

  if((fp = fopen("data.txt", "w")) != NULL)
  {
    char brand[80] = "Fiat";
    char model[80] = "126p";
    int year = 1977;
    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 0;
}
```

```
Dane samochodu:
    Marka: Fiat
    Model: 126p
    Rocznik: 1977
    Przebieg: 128.23
```

```c
char * fgets(char *s, int n, FILE *stream );
```

Funkcja `fgets()` wczytuje dane do bufora `s` ze strumienia (pliku) `stream`.

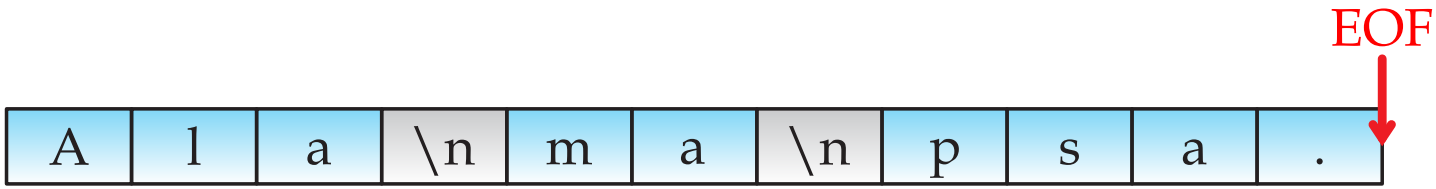
Parametr `n` określa maksymalną pojemność bufora, uwzględniającą miejsce na znacznik końca napisu.

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.

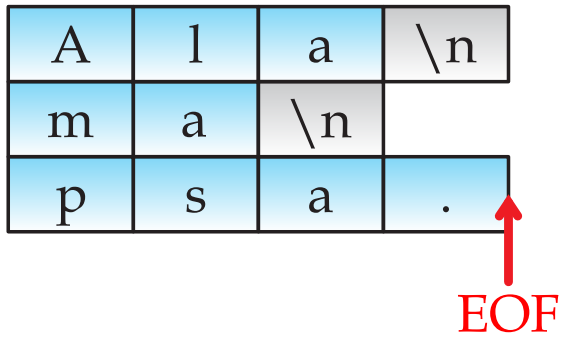
```c
#include <stdio.h>

#define MAX_LINE 256

int main()
{
  FILE *fp = NULL;

  if((fp = fopen( "data.txt", "r")) != NULL)
  {
    char line[MAX_LINE];

    while(fgets(line, MAX_LINE, fp) != NULL)
      printf("%s", line);

    fclose(fp);
  }

  return 0;
}
```

Ostatni wiersz pliku może nie być zakończony znakiem nowego wiersza `\n`. Funkcja `fgets()` odczyta ten wiersz normalnie, jednak w tablicy, w odróżnieniu od pozostałych wierszy, nie będzie na jego końcu znaku `\n`.

Zawartość pliku:



Przy przetwarzaniu linia po linii:

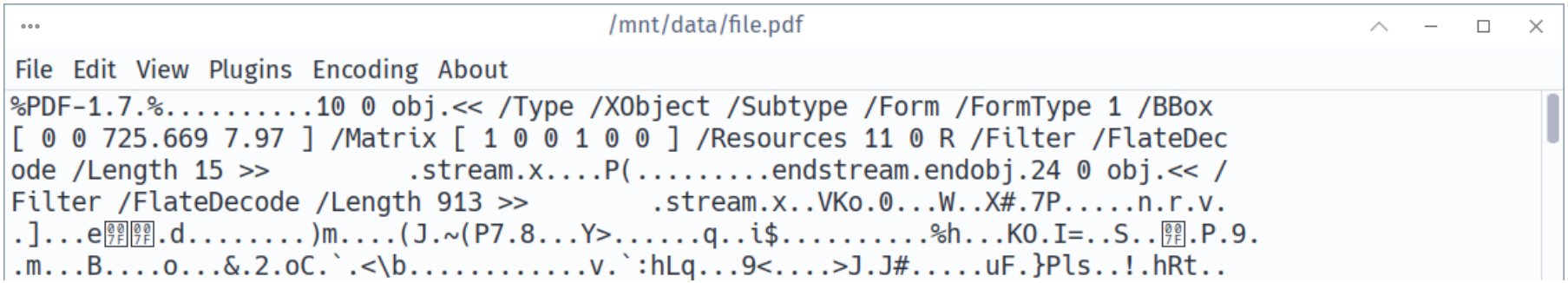


1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

Pliki binarne

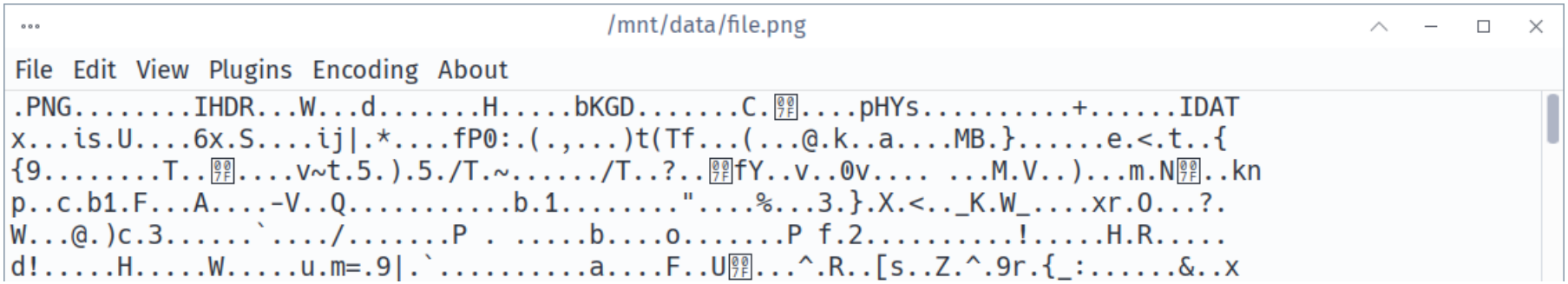
Często pliki **nie zawierają danych tekstowych**. Przykładem są pliki graficzne, dźwiękowe czy multimedialne.

Ich zawartość to najczęściej **binarny obraz zawartości pamięci operacyjnej** (np. ciąg bajtów opisujących kolory piksela) uzupełniony o dodatkowe informacje (np. nagłówek pliku graficznego czy dane EXIF).



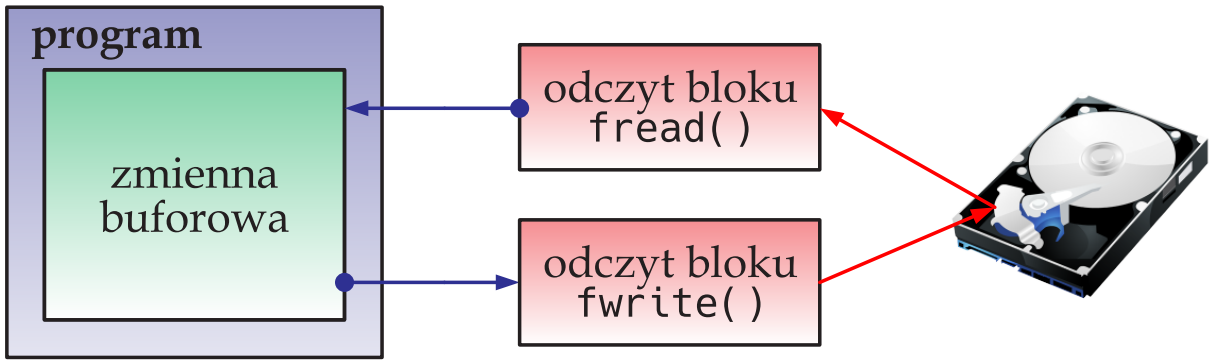
Do przetwarzania plików, których zawartość ma charakter binarny, wykorzystuje się najczęściej **odczyt i zapis bloków**.

Pod pojęciem bloku rozumieć będziemy **ciąg bajtów o określonej długości**, nie zakładamy, że ciąg taki ma jakąkolwiek strukturę. Rozmiar takiego bloku jest określony liczbą jego bajtów.



Przetwarzanie plików binarnych wymaga **otwartego pliku**. Plik powinien być otwarty w trybie binarnym.

Dodatkowo w programie należy posiadać **zmienną**, która będzie realizowała funkcję **bufora**. Ze zmiennej będą pobierane dane do zapisu i do niej będą zapisywane dane w przypadku odczytu.



Zapis i odczyt liczby

Przykład programu, który tworzy nowy plik, zapisuje do niego liczbę `float` i zamyka plik.

Potem otwiera plik w trybie do odczytu, odczytuje zapisaną liczbę i wyświetla jej wartości.

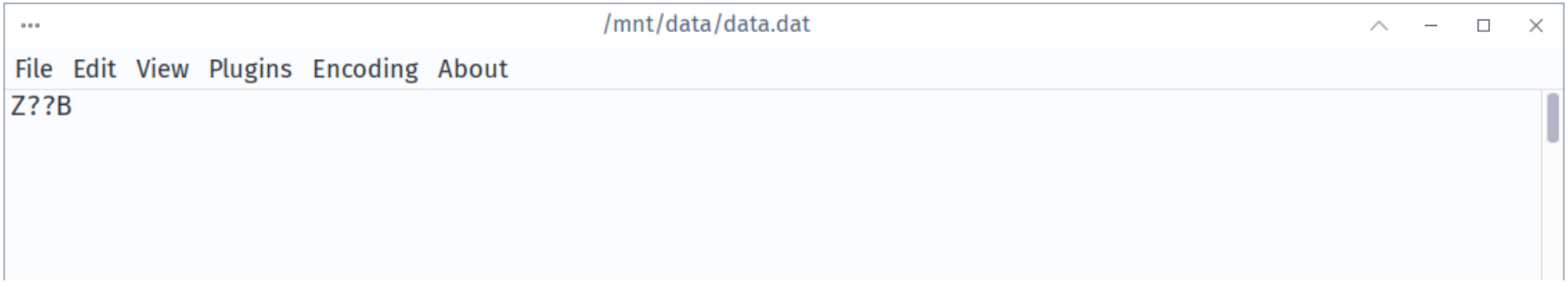
```c
#include <stdio.h>

int main()
{
  FILE *fp;
  float f = 123.321;

  if((fp = fopen("data.dat", "wb")) != NULL)
  {
    printf("\nZapis liczby: %g", f);
    fwrite(&f, sizeof(f), 1, fp);
    fclose(fp);
  }

  return 0;
}
```

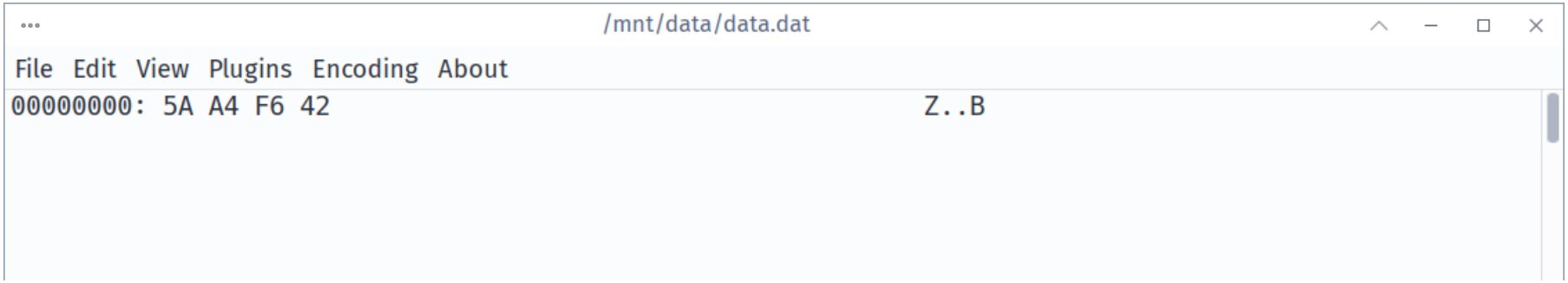
```
Zapis liczby: 123.321
```

```
fwrite(&f, sizeof(f), 1, fp)
```

Kolejne argumenty funkcji `fwrite()`:

- `&f` – **wskaźnik** na zmienną `f`, która ma być zapisana do pliku `fp`,
- `sizeof(f)` – **rozmiar** zapisywanego bloku,
- `1` – liczba zapisywanych **bloków**,
- `fp` – **wskaźnik pliku** otwartego do zapisu.


/mnt/data/data.dat
File Edit View Plugins Encoding About
Z??B


/mnt/data/data.dat
File Edit View Plugins Encoding About
00000000: 5A A4 F6 42 Z..B

```c
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);          ← do C99
```

```c
size_t fwrite(const void *restrict ptr, size_t size, size_t n, FILE *restrict stream);
                                                                              → od C99
```

Funkcja `fwrite()` zapisuje dane z obszaru pamięci wskazywanego przez `ptr` do strumienia `stream`. Funkcja zapisuje `n` bloków o rozmiarze `size`.

Łączna liczba zapisanych bajtów to `n * size`. Rezultatem funkcji jest liczba zapisanych **bloków** (nie bajtów).

W przypadku wystąpienia **końca pliku** lub **błędu**, rezultatem funkcji jest liczba, **bezbłędnie zapisanych bloków** (może być zerowa).

```c
#include <stdio.h>

int main()
{
  FILE *fp;
  float f = 0;

  if((fp = fopen("data.dat", "rb")) != NULL)
  {
    fread(&f, sizeof(f), 1, fp);
    printf("\nOdczyt liczby: %g", f);
    fclose(fp);
  }

  return 0;
}
```

```
Odczyt liczby: 123.321
```

```
fread(&f, sizeof(f), 1, fp)
```

Kolejne argumenty funkcji `fread()`:

- `&f` – wskaźnik na zmienną `f`, gdzie ma być zapisany blok odczytany z pliku `fp`,
- `sizeof(f)` – **rozmiar** odczytywanego bloku,
- `1` – liczba odczytywanych **bloków**,
- `fp` – **wskaźnik pliku** otwartego do odczytu.

```c
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Funkcja `fread()` czyta dane ze strumienia `stream` do obszaru pamięci wskazywanego przez `ptr`. Funkcja odczytuje `n` bloków o rozmiarze `size`.

Łączna liczba odczytanych bajtów to `n * size`. Rezultatem funkcji jest liczba odczytanych **bloków** (nie bajtów).

W przypadku wystąpienia **końca pliku** lub **błędu** rezultatem funkcji jest liczba **bezbłędnie odczytanych bloków** (może być zerowa).

Kontrola poprawności

Funkcje `fread()` i `fwrite()` pozwalają na kontrolę poprawności wykonywanych operacji odczytu i zapisu.

Wystarczy kontrolować rezultat wywołania tych funkcji i porównywać z liczbą określonych bloków.

```
if((fp = fopen("data.dat", "wb")) != NULL)
{
  if(fwrite(&f, sizeof(f), 1, fp) != 1)
    printf("\nBlad zapisu!");
  else
    printf("\nZapis wykonany.");
  fclose(fp);
}
```

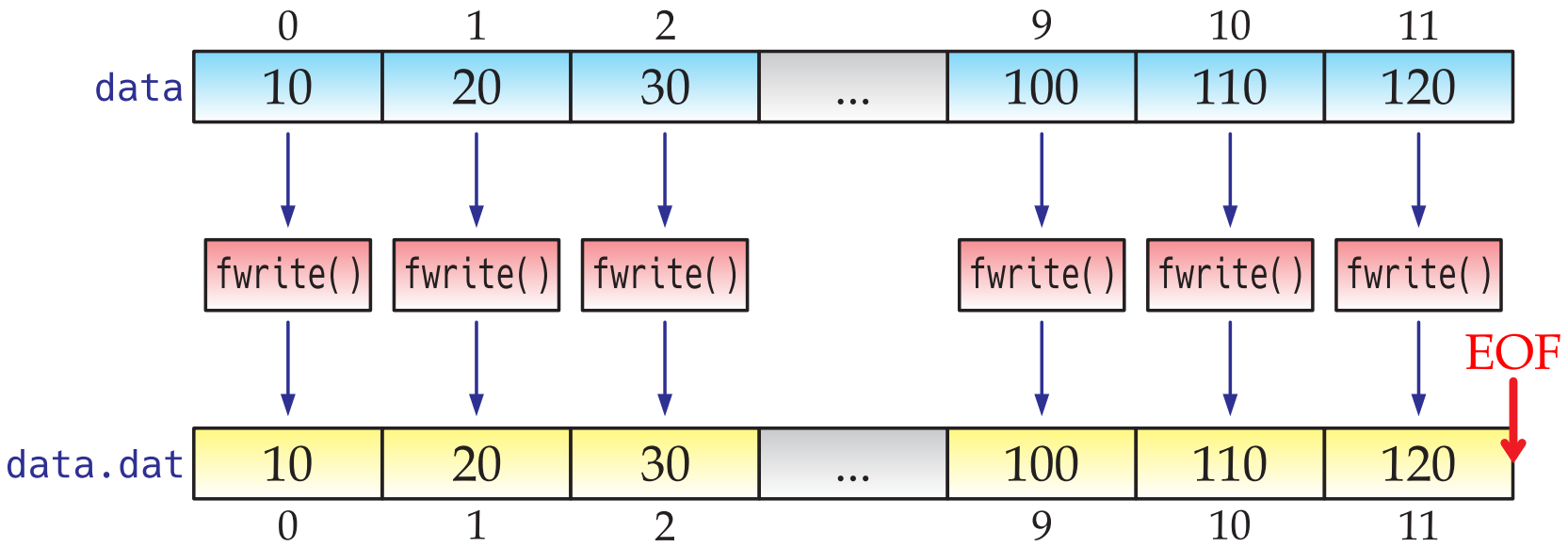
```
if((fp = fopen("data.dat", "rb")) != NULL)
{
  if(fread(&f, sizeof(f), 1, fp) != 1)
    printf("\nBlad odczytu!");
  else
    printf("\nOdczyt liczby: %g.", f);
  fclose(fp);
}
```

Zapis i odczyt ciągów danych

Zapisujemy do pliku dwanaście liczb typu `float` reprezentujących jakąś comiesięczną wartość zmiennej. Dane są zapisane w dwunastoelementowej tablicy `data`.

```
#define MONTHS 12

float data[MONTHS];
```

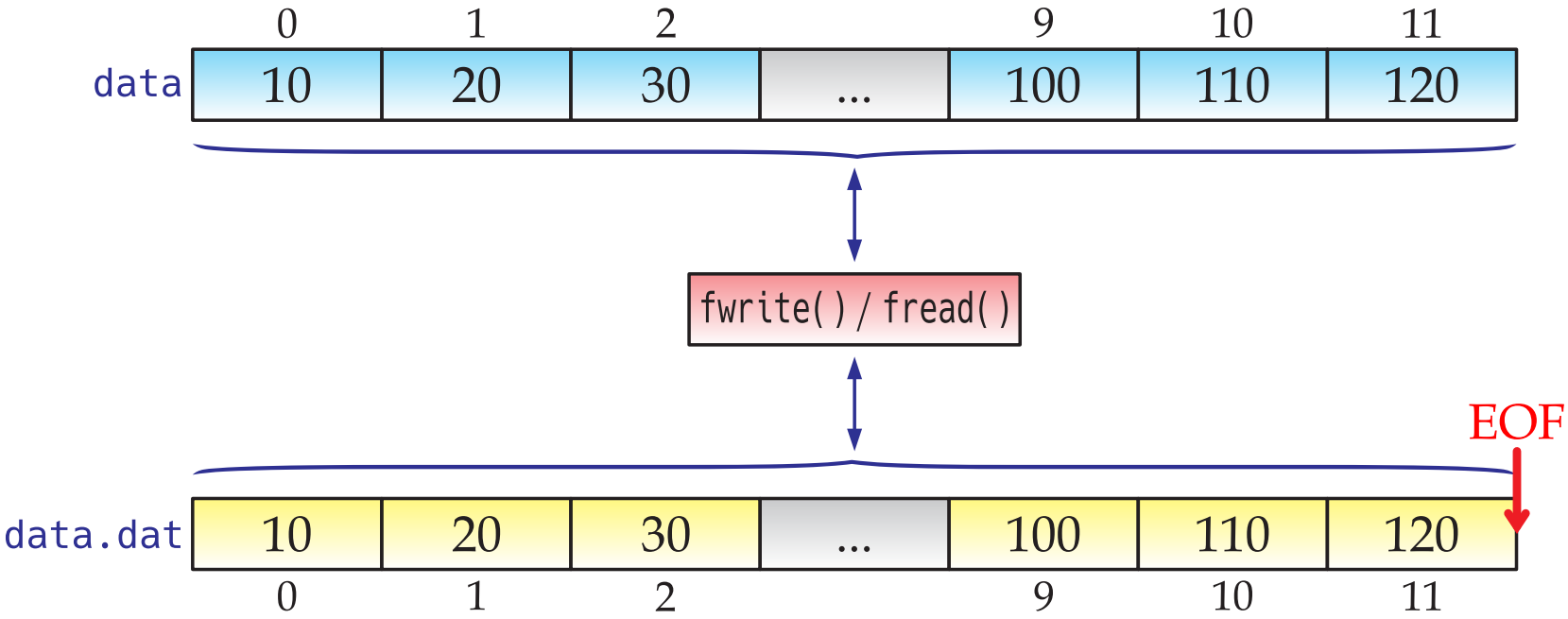
Pierwszym rozwiązaniem jest zapisanie kolejno każdego elementu tablicy jako bloku, wykorzystując funkcję `fwrite()`.



```c
#include <stdio.h>

#define MONTHS 12

int main()
{
  FILE *fp;
  float data[MONTHS];
  int i;

  for(i = 0; i < MONTHS; i++)                          ← przykładowe dane
    data[i] = 10 * (i + 1);

  if((fp = fopen("data.dat", "wb")) != NULL)
  {
    for(i = 0; i < MONTHS; i++)
      if(fwrite(&data[i], sizeof(data[i]), 1, fp) != 1)  ← zapis po elemencie
        printf("\nBlad zapisu!");
      else
        printf("\nZapisano: %g.", data[i]);

    fclose(fp);
  }
...
```

```
...
  for(i = 0; i < MONTHS; i++)                              ← zerowanie tablicy
    data[i] = 0;

  if((fp = fopen("data.dat", "rb")) != NULL)
  {
    for(i = 0; i < MONTHS; i++)
      if(fread(&data[i], sizeof(data[i]), 1, fp) != 1)     ← odczyt po elemencie
        printf("\nBlad odczytu!");
      else
        printf("\nOdczytano: %g.", data[i]);

    fclose(fp);
  }

  return 0;
}
```

Cała tablica może być blokiem zapisywanym lub odczytywanym w jednym wywołaniu funkcji `fwrite()` lub `fread()`.



```c
#include <stdio.h>

#define MONTHS 12

int main()
{
  FILE *fp;
  float data[MONTHS];
  int i;

  for(i = 0; i < MONTHS; i++)
    printf("\nZapis: %g.", data[i] = 10 * (i + 1));

  if((fp = fopen("data.dat", "wb")) != NULL)
  {
    if(fwrite(&data[0], sizeof(data[0]), MONTHS, fp) != MONTHS)
      printf("\nBlad zapisu!");

    fclose(fp);
  }
...
```

```
fread(&data[0], sizeof(data[0]), MONTHS, fp)
```

Kolejne argumenty funkcji `fread()`:

- `&data[0]` – **wskaźnik** na pierwszy element tablicy `data`, czyli początek bloku, który ma być odczytany z pliku `fp`,
- `sizeof(data[0])` – **rozmiar** odczytywanego elementu,
- `MONTHS` – **liczba** odczytywanych elementów,
- `fp` – **wskaźnik pliku** otwartego do odczytu,
- **rozmiar** odczytywanego bloku: `sizeof(data[0]) * MONTHS`.

```
...
  for(i = 0; i < MONTHS; i++)
    data[i] = 0;

  if((fp = fopen("data.dat", "rb")) != NULL)
  {
    if(fread(&data[0], sizeof(data[0]), MONTHS, fp) != MONTHS)
      printf("\nBlad odczytu!");

    fclose(fp);
  }

  for(i = 0; i < MONTHS; i++)
    printf("\nOdczyt: %g.", data[i]);

  return 0;
}
```

```
fwrite(&data[0], sizeof(data[0]), MONTHS, fp)
```

Kolejne argumenty funkcji `fwrite()`:

- `&data[0]` – **wskaźnik** na pierwszy element tablicy `data`, czyli początek bloku, który ma być zapisany do pliku `fp`,
- `sizeof(data[0])` – **rozmiar** zapisywanego elementu,
- `MONTHS` – **liczba** zapisywanych elementów,
- `fp` – **wskaźnik pliku** otwartego do zapisu,
- **rozmiar** zapisywanego bloku: `sizeof(data[0]) * MONTHS`.

Nazwa tablicy jest ustalonym wskaźnikiem na jej pierwszy element.

Zamiast zastosowanego:

```
fread(&data[0], sizeof(data[0]), MONTHS, fp);
```

równie dobrze można użyć:

```
fread(data, sizeof(data[0]), MONTHS, fp);
```

Pomocne funkcje dla typów int i float

```
int i = 10;
FILE *fp;

if(fwrite(&i, sizeof(i), 1, fp) != 1)
  printf("\nBlad zapisu!");
else
  printf("\nZapisano liczbe %d.", i);
```

Można w łatwy sposób napisać funkcję realizującą zapis pojedynczej zmiennej:

```
int writeInt(int i, FILE *file)
{
  return fwrite(&i, sizeof(i), 1, file) == 1;
}
```

```
int i = 10;
FILE *fp;
...
if(!writeInt(i, fp))
  printf("\nBlad zapisu!");
else
  printf("\nZapisano liczbe %d.", i);
```

```c
float f = 10.0;
FILE *fp;

if(fwrite(&f, sizeof(f), 1, fp) != 1)
  printf("\nBlad zapisu!");
else
  printf("\nZapisano liczbe %g.", f);
```

Analogiczna funkcja:

```c
int writeFloat(float f, FILE *file)
{
  return fwrite(&f, sizeof(f), 1, file) == 1;
}
```

```c
float f = 10.0;
FILE *fp;
...
if(!writeFloat(f, fp))
  printf("\nBlad zapisu!");
else
  printf("\nZapisano liczbe %g.", f);
```

Warto mieć pod ręką więcej podobnych funkcji.

Kopiowanie pliku

Przykładowa funkcja wykonuje kopiowanie zawartości pliku źródłowego do pliku docelowego. Wykorzystuje blokowe operacje zapisu i odczytu plików.

Parametrami są poprawnie otwarte binarnie pliki. Funkcja zwraca wartość 1, jeśli kopiowanie powiodło się lub 0, gdy wystąpił błąd podczas kopiowania.

```c
int copyFile(FILE *src, FILE *dst)
{
  char *copyBuff = NULL;          ← wskaźnik na bufor
  size_t buffSize = 10 * 1024;    ← rozmiar bufora
  size_t in = 0;                  ← liczba odczytanych bloków (tu bajtów)
  int result = 1;

  if((copyBuff = malloc(buffSize)) == NULL)
    return 0;

  while((in = fread(copyBuff, 1, buffSize, src)) != 0)
    if(fwrite(copyBuff, 1, in, dst) != in)
    {
      result = 0;                 ← błąd zapisu
      break;
    }

  if(ferror(src))
    result = 0;                   ← błąd odczytu

  free(copyBuff);

  return result;
}
```

Funkcja `fread()` zwraca liczbę odczytanych **jednobajtowych** bloków, czyli liczbę odczytanych bajtów, która potem wykorzystywana jest do określenia liczby bajtów do zapisu w funkcji `fwrite()`.

```
while((in = fread(copyBuff, 1, buffSize, src)) != 0)
  if(fwrite(copyBuff, 1, in, dst) != in)
    return 0;
```

Kolejne elementy programu:

- `in` – liczba odczytanych jednobajtowych bloków, czyli **liczba odczytanych bajtów**,
- `1` – rozmiar odczytywanego elementu (uwaga: **1**),
- `buffSize` – liczba odczytywanych elementów,
- `in` – liczba zapisywanych bajtów, tyle ile udało się odczytać,
- rozmiar odczytywanego bloku: `1 * buffSize`.

Wyświetlanie zawartości plików

Przykładowa funkcja wyświetla zawartość pliku na standardowym wyjściu. Plik wyświetlany jest szesnastkowo i jeśli to możliwe również z wykorzystaniem znaków ASCII. Parametrem jest poprawnie otwarty plik.

```
#define BUFFER_LEN 19                  ← liczba znaków w linii

void hexFileDump(FILE *file)
{
  unsigned char buffer[BUFFER_LEN];    ← bufor
  int in, i = 0;

  while((in = fread(buffer, 1, BUFFER_LEN, file)) > 0)
  {
    for(i = 0; i < in; i++)
      printf("%02X ", buffer[i]);      ← hex, dwa znaki, wiodące zera, duże litery

    printf("| ");                      ← separator

    for(i = 0; i < in; i++)            ← wypisz ASCII jeśli to możliwe
      printf("%c", isprint(buffer[i]) ? buffer[i] : '.');
                                       isprint() wymaga ctype.h
    putchar('\n');
  }
}
```

```
25 50 44 46 2D 31 2E 37 0A 25 CC D5 C1 D4 C5 D8 D0 C4 C6 | %PDF-1.7.%.....
0A 31 30 20 30 20 6F 62 6A 0A 3C 3C 20 2F 54 79 70 65 20 | .10 0 obj.<< /Ty
2F 58 4F 62 6A 65 63 74 20 2F 53 75 62 74 79 70 65 20 2F | /XObject /Subtyp
46 6F 72 6D 20 2F 46 6F 72 6D 54 79 70 65 20 31 20 2F 42 | Form /FormType 1
42 6F 78 20 5B 20 30 20 30 20 37 32 35 2E 36 36 39 20 37 | Box [ 0 0 725.66
2E 39 37 20 5D 20 2F 4D 61 74 72 69 78 20 5B 20 31 20 30 | .97 ] /Matrix [
20 30 20 31 20 30 20 30 20 5D 20 2F 52 65 73 6F 75 72 63 |  0 1 0 0 ] /Resc
65 73 20 31 31 20 30 20 52 20 2F 46 69 6C 74 65 72 20 2F | es 11 0 R /Filte
46 6C 61 74 65 44 65 63 6F 64 65 20 2F 4C 65 6E 67 74 68 | FlateDecode /Len
20 31 35 20 3E 3E 20 20 20 20 20 20 20 20 20 0A 73 74 72 |  15 >>          .
...
```

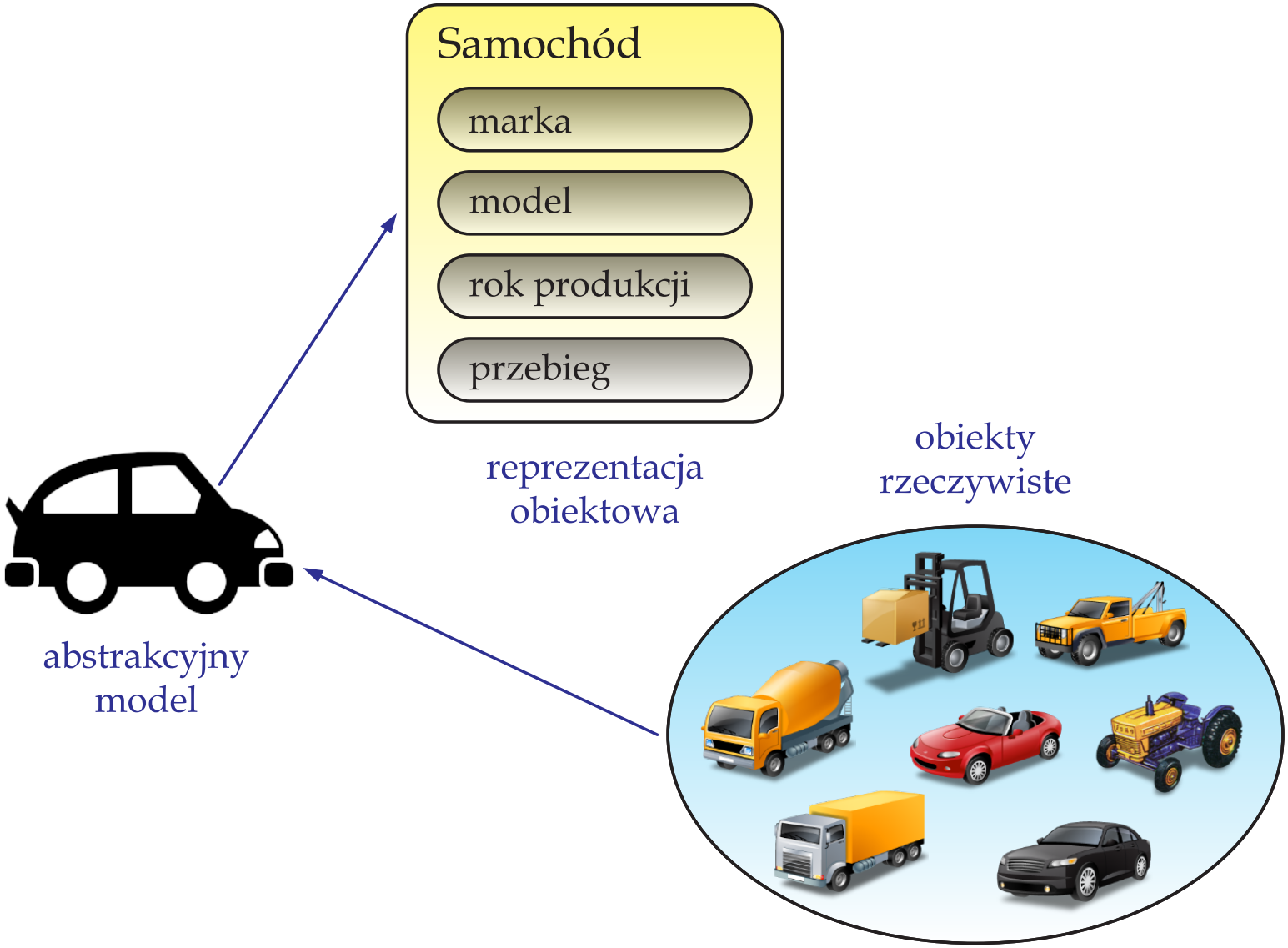
1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

Model rzeczywistości

Obiekty, elementy, pojęcia i opisy zjawisk znane z rzeczywistości muszą zostać odwzorowane danymi w programie.

Dane występujące w programie stanowią uproszczony, komputerowy wyabstrahowany **model** rzeczywistości.

Potrzeba wiele porcji danych, każda porcja jest złożona i zawiera dane opisujące obiekt.


Samochód
marka
model
rok produkcji
przebieg
reprezentacja obiektowa
obiekty rzeczywiste
abstrakcyjny model

Definicja i parametryzacja struktur

Struktura to złożony typ danych mogący przechowywać wiele wartości w jednej zmiennej. Od tablic jednakże różni się tym, iż te wartości mogą być różnych typów.

Definicja **typu strukturalnego** zawiera nazwy i typy pól struktury.

```
struct car
{
  char brand[20];
  char model[20];
  int year;
  float mileage;
};
```

Przykład deklaracji zmiennej strukturalnej:

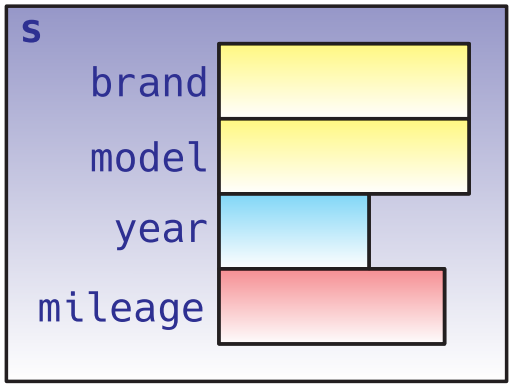
```
struct car s;
```

Wielkości tablic w strukturach można parametryzować – w języku C dyrektywą `#define`, w C++ także stałą `const`:

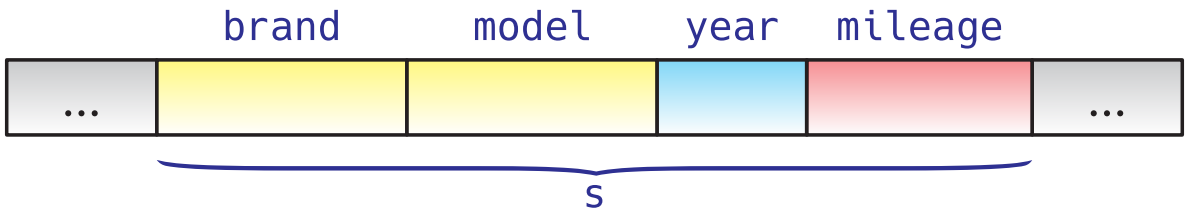
```c
#define MAX_NAME 20

struct car
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  int year;
  float mileage;
};
```

```c
const int MAX_NAME = 20;

struct car
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  int year;
  float mileage;
};
```

Struktura przypomina rekord znany z baz danych.



Kolejne pola struktury umieszczane są w pamięci jedno za drugim, w kolejności deklaracji. W odróżnieniu od tablicy, między polami mogą jednak wystąpić puste bajty (**dopełnienie**, *padding*), wynikające z wyrównania danych w pamięci – dlatego `sizeof` struktury może być większy od sumy rozmiarów pól.

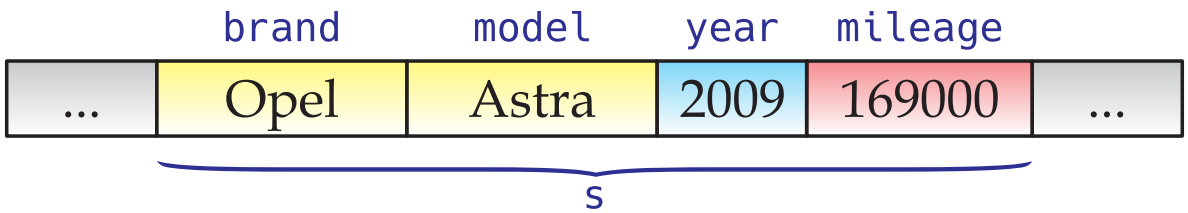


Odwołanie do pól struktury

Aby odwołać się do pola struktury należy po jej nazwie użyć operatora selekcji pola (.) i nazwy pola struktury.

```
struct car
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  int year;
  float mileage;
};
```

```
struct car s;

strcpy(s.brand,"Opel");
strcpy(s.model, "Astra");

s.year = 2009;
s.mileage = 169000;
```



```c
struct car
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  int year;
  float mileage;
};
```

```c
struct car s;

printf("Marka: %s", s.brand);
printf("Model: %s", s.model);
printf("Rok produkcji: %d", s.year);
printf("Przebieg: %g", s.mileage);
```

Typy strukturalne

W języku C nazwa struktury **nie jest samodzielną nazwą** typu strukturalnego. W deklaracjach zmiennych należy używać słowa kluczowego `struct`.

Aby nie używać słowa kluczowego `struct` można zadeklarować nazwę synonimiczną.

```
struct car
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  int year;
  float mileage;
};

typedef struct car car;
```

```
car s;
```

```
typedef struct
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  int year;
  float mileage;
} car;
```

```
car s;
```

Nazwa struktury i nazwa synonimu mogą być takie same – należą do różnych przestrzeni nazw.

W języku C++ nazwa oznacznikowa struktury występująca po słowie `struct` jest pełnoprawną **nazwą typu strukturalnego**. Nie trzeba używać słowa `struct` przy deklaracji.

```
struct car
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  int year;
  float mileage;
};
```

```
car s;
```

Struktury i wskaźniki

```
typedef struct
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  int year;
  float mileage;
} car;
```

```
car s;    ← deklaracja zmiennej strukturalnej typu car
car *sp;  ← deklaracja zmiennej wskaźnikowej do typu car
```

Tak zadeklarowana zmienna (automatyczna) `sp` ma wartość nieokreśloną.

```c
typedef struct
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  int year;
  float mileage;
} car;
```

```c
car s;
car *sp;

sp = &s;              ← zmienna sp wskazuje zmienną s
(*sp).year = 2009;    ← wyrażenie *sp reprezentuje strukturę s
```

Zmienna `*sp` lokalizuje w pamięci zmienną `s` typu `car`.

Odwołanie z użyciem wskaźnika można realizować na dwa sposoby:

```
(*sp).year = 2009;
```

- `sp` – zmienna wskaźnikowa,
- `(*sp)` – wskazywana struktura,
- `(*sp).year` – pole struktury wskazywanej przez `sp`,

oraz

```
sp->year = 2009;
```

- `sp` – zmienna wskaźnikowa,
- `->` – operator dostępu do składowej,
- `sp->year` – pole struktury wskazywanej przez `sp`.

Struktury jako parametry funkcji

Przykład przekazywania struktury przez wartość:

```
void printInfo(car s)
{
  printf("\nMarka: %s", s.brand);
  printf("\nModel: %s", s.model);
  printf("\nRok produkcji: %d", s.year);
  printf("\nPrzebieg: %g", s.mileage);
}
```

```
strcpy(s.brand,"Opel");
strcpy(s.model, "Astra");
s.year = 2009;
s.mileage = 169000;

printInfo(s);
```

Przykład przekazywania wskaźnika do struktury:

```
void printInfo(car *s)
{
  printf("\nMarka: %s", s->brand);
  printf("\nModel: %s", s->model);
  printf("\nRok produkcji: %d", s->year);
  printf("\nPrzebieg: %g", s->mileage);
}
```

```
strcpy(s.brand,"Opel");
strcpy(s.model, "Astra");
s.year = 2009;
s.mileage = 169000;

printInfo(&s);
```

Zmienne strukturalne mają często duży rozmiar. Przekazywanie ich przez wartość:

- **zabiera pamięć** – parametr formalny alokowany jest na stosie i jest kopią parametru aktualnego,
- **zabiera czas** – robienie kopii parametru aktualnego wywołania wymaga przesłania potencjalnie dużej liczby bajtów,
- **uniemożliwia modyfikacje** – modyfikacje wykonane na parametrze formalnym funkcji nie przenoszą się na parametr aktualny wywołania.

Przekazywanie wskaźników do struktury:

- **oszczędza pamięć** – parametr formalny alokowany na stosie zawiera jedynie adres parametru aktualnego,
- **jest szybkie** – przekazanie wskaźnika to przesłanie małej liczby bajtów,
- **pozwala na modyfikacje** – modyfikacje wykonane przez parametr formalny funkcji przenoszą się na parametr aktualny.

Jeżeli modyfikacje obiektu wskazywanego mają być zabronione, można użyć słowa kluczowego `const` w deklaracji funkcji.

```
void printInfo(const car *s)
{
  printf("\nMarka: %s", s->brand);
  printf("\nModel: %s", s->model);
  printf("\nRok produkcji: %d", s->year);
  printf("\nPrzebieg: %g", s->mileage);
}
```

W języku C++ można przekazywać parametry przez **referencję**. Nie trzeba wtedy używać wskaźników, a działanie funkcji jest analogiczne. Referencja ustalona (z `const`) nie pozwala na modyfikację parametru aktualnego wywołania.

```
void printInfo(car &s)
{
  printf("\nMarka: %s", s.brand);
  printf("\nModel: %s", s.model);
  printf("\nRok produkcji: %d", s.year);
  printf("\nPrzebieg: %g", s.mileage);
}
```

```
void printInfo(const car &s)
{
  printf("\nMarka: %s", s.brand);
  printf("\nModel: %s", s.model);
  printf("\nRok produkcji: %d", s.year);
  printf("\nPrzebieg: %g", s.mileage);
}
```

Przykładowa funkcja zmieniająca zawartość struktury.

```
void scanInfo(car *s)
{
  char buff[MAX_BUFF];

  printf("Marka: ");
  fgets(buff, sizeof(buff), stdin);
  buff[strcspn(buff, "\r\n")] = '\0';

  if(strlen(buff) >= MAX_NAME)
    buff[MAX_NAME - 1] = '\0';
  strcpy(s->brand, buff);

  printf("Model: ");
  fgets(buff, sizeof(buff), stdin);
  buff[strcspn(buff, "\r\n")] = '\0';

  if(strlen(buff) >= MAX_NAME)
    buff[MAX_NAME - 1] = '\0';
  strcpy(s->model, buff);
  ...
```

```
  ...
  printf("Rok produkcji: ");
  fgets(buff, sizeof(buff), stdin);

  s->year = atoi(buff);

  printf("Przebieg: ");
  fgets(buff, sizeof(buff), stdin);

  s->mileage = atof(buff);
}
```

1 Informacje ogólne
2 Wprowadzenie do programowania
3 Język programowania, system operacyjny, środowisko programowania
4 Paradygmat programowania
5 Środowisko programowania
6 Łagodny start
7 Jednostki leksykalne i typy danych
8 Operatory i wyrażenia
9 Instrukcje sterujące
10 Podprogramy i struktura programu
11 Zmienne wskaźnikowe i tablice
12 Pliki i przetwarzanie plików tekstowych
13 Przetwarzanie plików binarnych
14 Struktury i rekordy
15 Tablice struktur i pliki struktur

Definicja typu strukturalnego jeszcze raz

Zamieniony został typ `int` na 16-bitowy typ `short int`.

```
#define MAX_NAME 20

typedef struct
{
  char brand[MAX_NAME];
  char model[MAX_NAME];
  short int year;
  float mileage;
} car;
```

Rozmiar struktury nie zmniejszy się jednak – pole `mileage` musi zaczynać się od adresu podzielnego przez 4, więc po polu `year` kompilator wstawi 2 bajty dopełnienia. W obu przypadkach `sizeof(car)` wynosi 48.

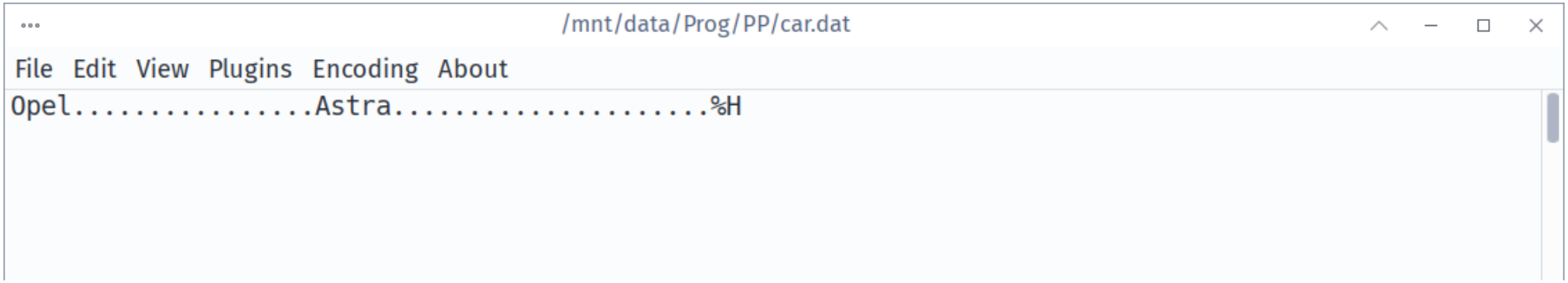
Mamy już zdefiniowane funkcje do wyświetlania i wczytywania pól struktury.

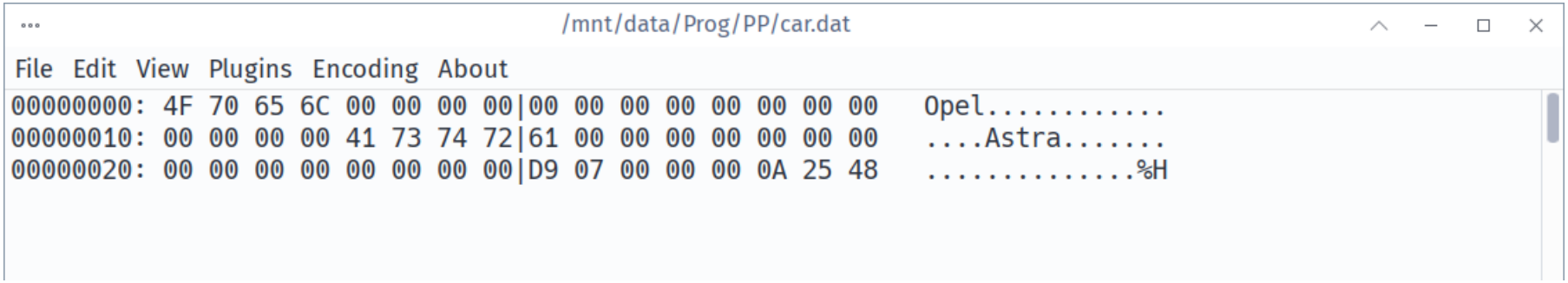
```
void printInfo(const car *s)
{
  printf("\nMarka: %s", s->brand);
  printf("\nModel: %s", s->model);
  printf("\nRok produkcji: %d", s->year);
  printf("\nPrzebieg: %g", s->mileage);
}
```

```c
void scanInfo(car *s)
{
  char buff[MAX_BUFF];

  printf("Marka: ");
  fgets(buff, sizeof(buff), stdin);
  buff[strcspn(buff, "\r\n")] = '\0';

  if(strlen(buff) >= MAX_NAME)
    buff[MAX_NAME - 1] = '\0';
  strcpy(s->brand, buff);

  printf("Model: ");
  fgets(buff, sizeof(buff), stdin);
  buff[strcspn(buff, "\r\n")] = '\0';

  if(strlen(buff) >= MAX_NAME)
    buff[MAX_NAME - 1] = '\0';
  strcpy(s->model, buff);
  ...
  ...
  printf("Rok produkcji: ");
  fgets(buff, sizeof(buff), stdin);

  s->year = atoi(buff);

  printf("Przebieg: ");
  fgets(buff, sizeof(buff), stdin);

  s->mileage = atof(buff);
}
```

Zapis i odczyt struktury – plik binarny

```
FILE * fp;
car a, b;

scanInfo(&a);

if((fp = fopen("car.dat", "wb")) != NULL)
{
  fwrite(&a, sizeof(car), 1, fp);
  fclose(fp);
}

if((fp = fopen("car.dat", "rb")) != NULL)
{
  fread(&b, sizeof(car), 1, fp);
  fclose(fp);
}

printInfo(&b);
```


/mnt/data/Prog/PP/car.dat
File Edit View Plugins Encoding About
Opel.....Astra.....%H


/mnt/data/Prog/PP/car.dat
File Edit View Plugins Encoding About
00000000: 4F 70 65 6C 00 00 00 00|00 00 00 00 00 00 00 00 Opel............
00000010: 00 00 00 00 41 73 74 72|61 00 00 00 00 00 00 00 ....Astra.......
00000020: 00 00 00 00 00 00 00 00|D9 07 00 00 00 0A 25 48 ....%H

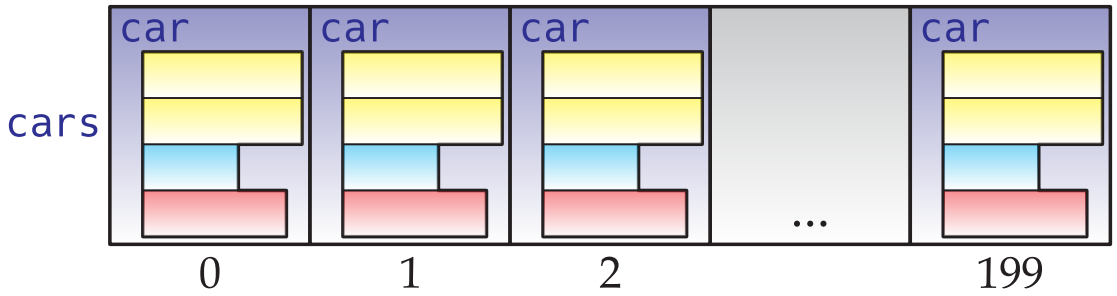
Wygodnie jest zdefiniować funkcje do zapisu i wczytywania struktur.

```
void infoToFile(const char *fileName, const car *s)
{
  FILE *fp;

  if((fp = fopen(fileName, "wb")) == NULL)
    printf("\nBlad otwarcia pliku.");

  else
  {
    int n = fwrite(s, sizeof(car), 1, fp);

    if(n != 1)
      printf("\nBlad zapisu pliku.");

    fclose(fp);
  }
}
```

```c
void infoFromFile(const char *fileName, car *s)
{
  FILE *fp;

  if((fp = fopen(fileName, "rb")) == NULL)

    printf("\nBlad otwarcia pliku.");

  else
  {
    int n = fread(s, sizeof(car), 1, fp);

    if(n != 1)
      printf("\nBlad odczytu pliku.");

    fclose(fp);
  }
}
```

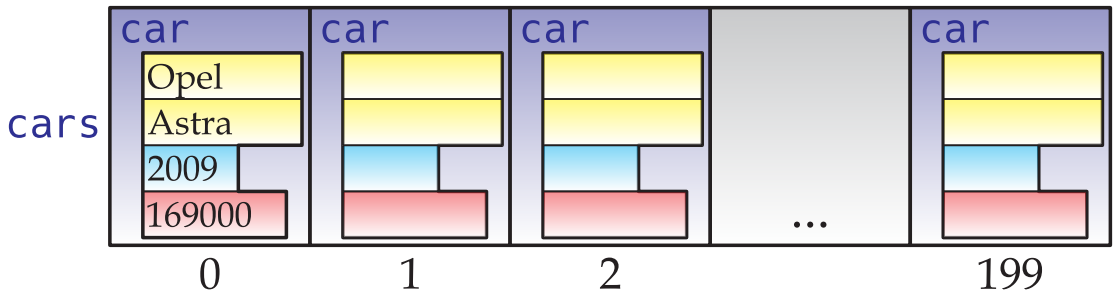
Tablica struktur

```
#define MAX_CARS 200    ← maksymalna liczba samochodów

car cars[MAX_CARS];     ← tablica struktur opisujących samochody
short int noCars = 0;   ← aktualna liczba pojazdów
```



Odwołanie do pól struktury w tablicy

```
strcpy(cars[0].brand, "Opel");
strcpy(cars[0].model, "Astra");

cars[0].year = 2009;
cars[0].mileage = 169000;

noCars++;
```



Przykładowa funkcja służąca do dodawania kolejnych wpisów do tablicy:

```
void scanCars(short int *noCars, car *cars)
{
  char buff[MAX_BUFF];
  int again = 0;

  if(*noCars < MAX_CARS)
    do
    {
      scanInfo(&cars[*noCars]);
      (*noCars)++;

      printf("\nDodac kolejny samochod (t/n)? ");

      fgets(buff, sizeof(buff), stdin);
      again = (tolower(buff[0]) == 't');
    }
    while(again && *noCars < MAX_CARS);

  if(*noCars == MAX_CARS)
    printf("\nNie mozna dodac wiecej samochodow.");
}
```

Przykładowa funkcja służąca do przeglądania kolejnych wpisów z tablicy:

```
void printCars(const short int *noCars, const car *cars)
{
  char buff[MAX_BUFF];
  int i = 0, again;

  if(*noCars == 0)
    printf("\nTablica pusta.");

  if(*noCars > 0)
    do
    {
      printf("\nInfo samochodu nr: %d", i + 1);
      printInfo(&cars[i]);
      i++;

      printf("\nKolejny samochod (t/n)? ");

      fgets(buff, sizeof(buff), stdin);
      again = (tolower(buff[0]) == 't');
    }
    while(again && i < *noCars);
}
```

Struktura w pliku

Zapis tablicy struktur można zrealizować w podobny sposób jak w przypadku tablic z elementami o typach prostych.

Dodajmy jeszcze do pliku binarnego wielkość zapisywanej tablicy.

W takim przypadku zapis będzie się odbywał w dwóch etapach:

- zapis i kontrola poprawności zapisu liczby elementów tablicy,
- zapis i kontrola poprawności zapisu tablicy rekordów.

Odczyt pliku będzie operacją odwrotną:

- odczyt i kontrola poprawności odczytu liczby elementów tablicy,
- odczyt i kontrola poprawności odczytu tablicy rekordów.

Zapis z kontrolą poprawności:

```
int n;

n = fwrite(&noCars, sizeof(noCars), 1, fp);

if(n != 1)
  printf("\nBlad zapisu pliku.");
```

```
int n;

if(noCars > 0)
{
  n = fwrite(cars, sizeof(car), noCars, fp);

  if(n != noCars)
    printf("\nBlad zapisu pliku.");
}
```

Przykładowa funkcja zapisująca tablicę struktur z liczbą elementów do pliku:

```
void structToFile(const char *fileName, const short int *noCars, const car *s)
{
  FILE *fp;

  if((fp = fopen(fileName, "wb")) == NULL)
    printf("\nBlad otwarcia pliku.");

  else
  {
    int n;
    n = fwrite(noCars, sizeof(*noCars), 1, fp);

    if(n != 1)
      printf("\nBlad zapisu pliku (liczba).");

    if(*noCars > 0)
    {
      n = fwrite(s, sizeof(car), *noCars, fp);

      if(n != *noCars)
        printf("\nBlad zapisu pliku (struktura).");
    }
    fclose(fp);
  }
}
```

Odczyt z kontrolą poprawności. Liczbie rekordów odczytanej z pliku nie można ufać – uszkodzony lub spreparowany plik może zawierać dowolną wartość, a `fread()` zapisze wtedy dane poza tablicą.

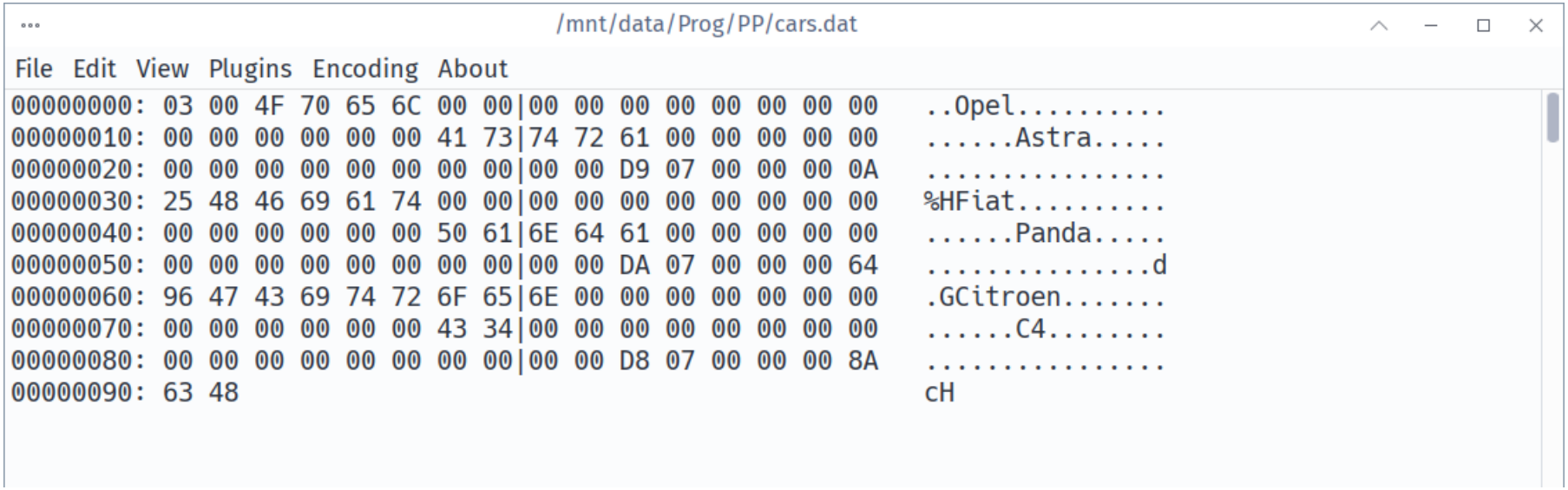
```c
int n;

n = fread(&noCars, sizeof(noCars), 1, fp);

if(n != 1 || noCars < 0 || noCars > MAX_CARS)
  printf("\nBlad odczytu pliku (liczba).");
```

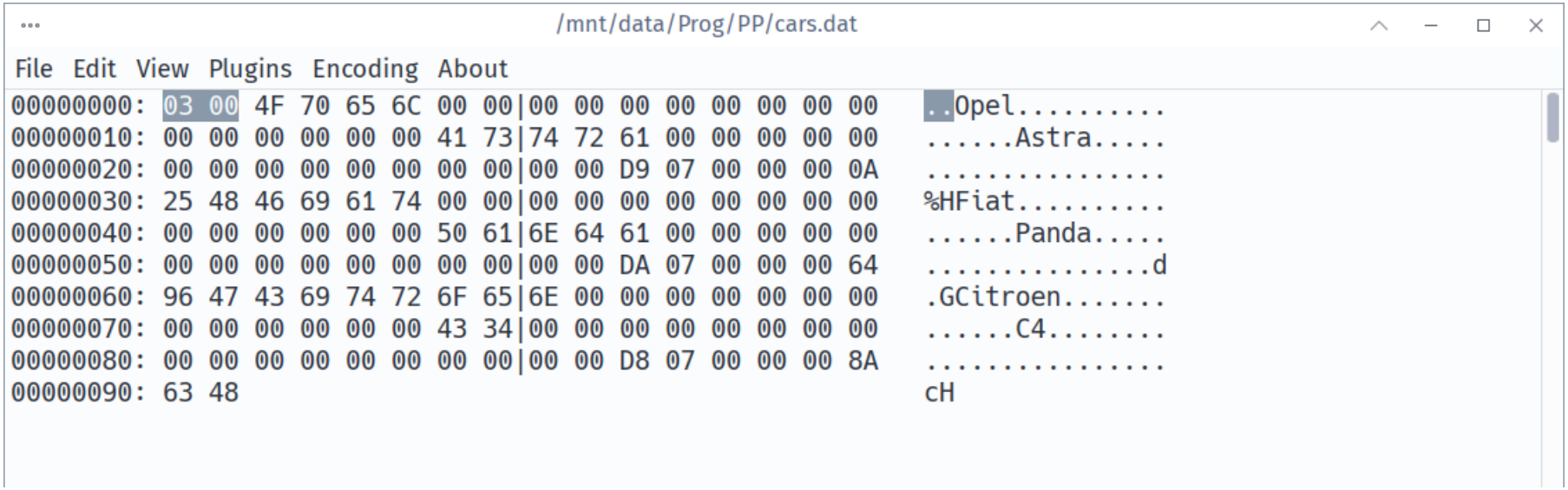
```c
int n;

n = fread(cars, sizeof(car), noCars, fp);

if(n != noCars)
  printf("\nBlad odczytu pliku (struktura).");
```

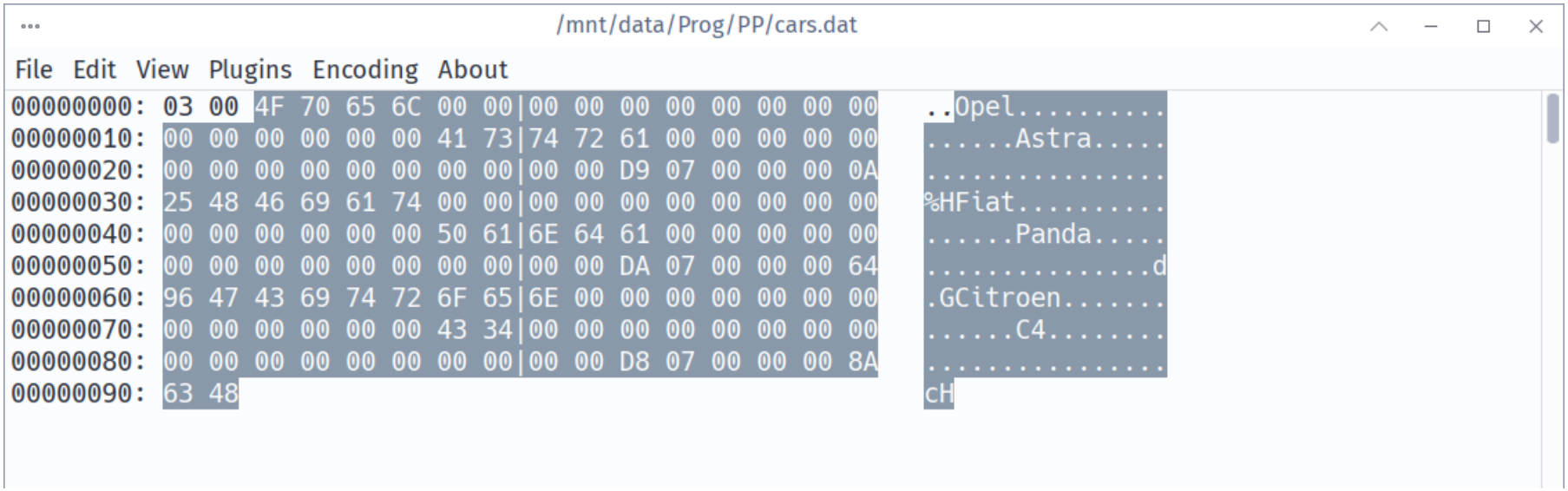
Przykładowa funkcja odczytująca tablicę struktur z liczbą elementów z pliku:

```
void structFromFile(const char *fileName, short int *noCars, car *s)
{
  FILE *fp;

  if((fp = fopen(fileName, "rb")) == NULL)
    printf("\nBlad otwarcia pliku.");

  else
  {
    int n;
    n = fread(noCars, sizeof(*noCars), 1, fp);
    ...
```

```
    ...
    if(n != 1 || *noCars < 0 || *noCars > MAX_CARS)
    {
      printf("\nBlad odczytu pliku (liczba).");
      *noCars = 0;
    }
    else if(*noCars > 0)
    {
      n = fread(s, sizeof(car), *noCars, fp);

      if(n != *noCars)
      {
        printf("\nBlad odczytu pliku (struktura).");
        *noCars = n;
      }
    }
    fclose(fp);
  }
}
```


/mnt/data/Prog/PP/cars.dat
File Edit View Plugins Encoding About
00000000: 03 00 4F 70 65 6C 00 00|00 00 00 00 00 00 00 00 ..Opel.....
00000010: 00 00 00 00 00 00 41 73|74 72 61 00 00 00 00 00 ......Astra.....
00000020: 00 00 00 00 00 00 00 00|00 00 D9 07 00 00 00 0A ....
00000030: 25 48 46 69 61 74 00 00|00 00 00 00 00 00 00 00 %HFiat.....
00000040: 00 00 00 00 00 00 50 61|6E 64 61 00 00 00 00 00 ......Panda.....
00000050: 00 00 00 00 00 00 00 00|00 00 DA 07 00 00 00 64 ....d
00000060: 96 47 43 69 74 72 6F 65|6E 00 00 00 00 00 00 00 .GCitroen.....
00000070: 00 00 00 00 00 00 43 34|00 00 00 00 00 00 00 00 ......C4.....
00000080: 00 00 00 00 00 00 00 00|00 00 D8 07 00 00 00 8A ....
00000090: 63 48 cH


/mnt/data/Prog/PP/cars.dat
File Edit View Plugins Encoding About
00000000: 03 00 4F 70 65 6C 00 00|00 00 00 00 00 00 00 00 ..Opel.....
00000010: 00 00 00 00 00 00 41 73|74 72 61 00 00 00 00 00 ......Astra.....
00000020: 00 00 00 00 00 00 00 00|00 00 D9 07 00 00 00 0A ....
00000030: 25 48 46 69 61 74 00 00|00 00 00 00 00 00 00 00 %HFiat.....
00000040: 00 00 00 00 00 00 50 61|6E 64 61 00 00 00 00 00 ......Panda.....
00000050: 00 00 00 00 00 00 00 00|00 00 DA 07 00 00 00 64 ....d
00000060: 96 47 43 69 74 72 6F 65|6E 00 00 00 00 00 00 00 .GCitroen.....
00000070: 00 00 00 00 00 00 43 34|00 00 00 00 00 00 00 00 ......C4.....
00000080: 00 00 00 00 00 00 00 00|00 00 D8 07 00 00 00 8A ....
00000090: 63 48 cH


/mnt/data/Prog/PP/cars.dat
File Edit View Plugins Encoding About
00000000: 03 00 4F 70 65 6C 00 00|00 00 00 00 00 00 00 00 ..Opel.....
00000010: 00 00 00 00 00 00 41 73|74 72 61 00 00 00 00 00 ......Astra.....
00000020: 00 00 00 00 00 00 00 00|00 00 D9 07 00 00 00 0A ....
00000030: 25 48 46 69 61 74 00 00|00 00 00 00 00 00 00 00 %HFiat.....
00000040: 00 00 00 00 00 00 50 61|6E 64 61 00 00 00 00 00 ......Panda.....
00000050: 00 00 00 00 00 00 00 00|00 00 DA 07 00 00 00 64 ....d
00000060: 96 47 43 69 74 72 6F 65|6E 00 00 00 00 00 00 00 .GCitroen.....
00000070: 00 00 00 00 00 00 43 34|00 00 00 00 00 00 00 00 ......C4.....
00000080: 00 00 00 00 00 00 00 00|00 00 D8 07 00 00 00 8A ....
00000090: 63 48 cH

Plik binarny ze strukturami zapisanymi w całości nie jest przenośny – rozmiary typów (np. `long`), dopełnienie i kolejność bajtów zależą od kompilatora i systemu.

Przenośny format binarny (jak PNG, ZIP czy WAV) określa dokładnie położenie i rozmiar każdego pola. W programie oznacza to:

- typy o ustalonym rozmiarze (`int16_t`, `int32_t` z `stdint.h`) zamiast `int` czy `long`,
- zapis i odczyt pól pojedynczo, a nie całej struktury – dopełnienie nie trafia wtedy do pliku,
- ustaloną kolejność bajtów (np. zawsze *little-endian*, jak w procesorach x86 i ARM),
- liczby zmiennoprzecinkowe w formacie IEEE-754, stosowanym na wszystkich popularnych platformach.

Najprostszą alternatywą jest plik tekstowy – większy i wolniejszy, ale czytelny wszędzie.