# C++ 20
# Biblioteka techniczna

Problemy i rozwiązania

—

J. Burton Browning
Bruce Sutherland

Apress®

Apress®

J. Burton Browning
Bruce Sutherland

# C++20
# Biblioteka techniczna

## Problemy i rozwiązania

Przekład: Marek Włodarz

APN Promise, Warszawa 2020

**C++20 Biblioteka techniczna. Problemy i rozwiązania**

First published in English under the title
C++20 Recipes: A Problem-Solution Approach by J. Burton Browning, Bruce Sutherland



APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: wydawnictwo@promise.pl



ISBN: 978-83-7541-434-9 (druk), 978-83-7541-437-0 (ebook)

Przekład: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

*Książkę dedykuję Zadzie Browning, miłości mojego życia.*

# Spis treści

# O autorach

**Dr J. Burton Browning** uzyskał doktorat na North Carolina State University pod kierunkiem dr. Richarda Petersona w roku 1999. Prowadził badania w takich obszarach, jak nauczanie zdalne, programowanie i techniki szkoleniowe. Jako nauczyciel i osoba o tak rozlicznych zainteresowaniach, jak programowanie, fotografia, robotyka, naprawa samochodów, obróbka drewna, łowiectwo, czytanie, łowienie ryb czy łucznictwo, nigdy nie znalazł się w sytuacji, gdy nie ma nic do zrobienia. Jego wcześniejsze publikacje obejmują prace na temat wielofunkcyjnych zespołów nauczania, The Utopian School (model szkoły kierowanej przez nauczyciela), programowania komputerów (wiele języków), oprogramowania otwartego, statystyk służby zdrowia i drążenia danych, działania przecinarki plazmowej, technik edukacyjnych, nauczania zdalnego i wielu innych. Po przejściu na emeryturę jako profesor kolegium w roku 2018 poświęcił się podróżom oraz pracy nad projektami motoryzacyjnymi i innymi.

**Bruce Sutherland** jest programistą gier wideo urodzonym w Dundee, w Szkocji. Uzyskał magisterium w dziedzinie Computer Games Technology na uniwersytecie Abertay w Dundee w roku 2005. Po ukończeniu studiów rozpoczął pracę w branży gier w firmie 4J Studios, gdzie pracował nad takimi tyułami, jak *Star Trek: Encounters* (PS2), *The Elder Scrolls IV: Oblivion* (PS3), *Star Trek: Conquest* (PS2, Wii), *Ducati Moto* (NDS) oraz *AMF Bowling Pinbusters!* (NDS). W lipcu 2008 roku przeprowadził się do Melbourne w Australii, gdzie dołączył do zespołu Visceral Studios i pracował jako inżynier oprogramowania nad tytułami *Dead Space* (Xbox 360, PS3, PC), *The Godfather II* (Xbox 360, PS3, PC) oraz *Dead Space 3* (Xbox 360, PS3, PC). W wolnych chwilach zajmuje się projektowaniem gier dla Androida oraz pisaniem tutoriali na swoim blogu.

# O recenzencie technicznym

**Michael Thomas** pracuje w branży programistycznej od ponad 20 lat jako indywidualny programista, szef zespołu, menedżer programu i wiceprezes do spraw inżynieryjnych. Ma ponad 10 lat doświadczenia w pracy z urządzeniami mobilnymi. Obecnie skupia się na zagadnieniach sektora medycznego i wykorzystaniu urządzeń mobilnych w celu przyśpieszenia przekazywania informacji pomiędzy pacjentami a przedsiębiorstwami opieki zdrowotnej.

## Podziękowania

Autorzy chcieliby przekazać swoje podziękowania Steve'owi Anglinowi, Matthewowi Moodie i Markowi Powersowi z wydawnictwa Apress oraz zespołowi produkcyjnemu za ich pomoc i wsparcie. Świetnie pracowało się z wami!

# Wprowadzenie

Język programowania C++ podlega ciągłemu rozwojowi i ulepszeniom. To dążenie do utrzymania języka w gronie najbardziej zaawansowanych i nowoczesnych wynika z prostego faktu, że C++ nadal odgrywa ważną rolę w tworzeniu wysokowydajnych, przenośnych aplikacji. Niewiele innych języków może być używanych na tak wielu platformach, jak C++, bez uzależnienia od środowiska wykonawczego. Częściowo wynika to z samej natury C++ jako kompilowanego języka programowania. Programy tworzone w C++ są budowane do postaci binarnych plików poprzez kombinację procesów obejmujących kompilację i konsolidację.

Wybór kompilatora jest szczególnie ważny w dzisiejszych czasach, co wynika z tempa zmian następujących w języku. Opracowywanie języka C++ rozpoczął Bjarne Stroustrup w roku 1979 – wówczas nosił on nazwę *C with Classes* (C z klasami). Język ten nie ujrzał formalnego ustandaryzowania aż do roku 1998, zaś uaktualniony standard został opublikowany w roku 2003. Nastąpiła wówczas kolejna ośmioletnia przerwa do ponownej aktualizacji standardu, co nastąpiło wraz z wprowadzeniem C++11 w roku 2011. Wersja ta wprowadziła do języka znaczącą liczbę zmian i rozszerzeń i jest wyraźnie odróżniana od „starszych" wersji C++ poprzez określenie *nowoczesny C++*. W wersjach C++17 i C++20 spowodowały wycofanie starszych funkcjonalności i dodały do języka wiele znaczących zmian.

W tej książce przedstawiamy kod napisany zgodnie ze standardami ISO od C++14 aż po C++20 z wykorzystaniem kompilatorów Clang, Microsoft Visual Studio (VS) 2019 oraz Xcode. Clang jest kompilatorem o otwartych źródłach, który rozpoczął istnienie jako zamknięty projekt firmy Apple. W roku 2007 Apple udostępniło kod społeczności open source i kompilator zyskuje nowe możliwości i funkcjonalności przez cały czas. Książka wyjaśnia, jak zainstalować i używać Clang na komputerach systemów OS X, Windows lub Linux (Ubuntu). Przykłady towarzyszące każdemu rozdziałowi zostały skompilowane i przetestowane przy użyciu Clang 3.5 i/lub Visual Studio 2019. Wszystkie wymienione aplikacje są bezpłatne, zatem należy wybrać tę, która najlepiej spełnia określone potrzeby albo używać wszystkich, aby nauczyć się więcej!

# Kod źródłowy

Dostęp do towarzyszącego książce kodu źródłowego można uzyskać poprzez łącze **Download Source Code** zlokalizowane na stronie *www.apress.com/9781484257128*. Znajdziemy tam kod źródłowy dla wszystkich listingów zawartych w książce, łącznie z plikami makefile, które można wykorzystać do budowania działających programów.

Rozdział 1

# Początki C++

Język programowania C++ to efektywny język niskiego poziomu, pozwalający na pisanie programów, które są kompilowane do instrukcji maszynowych w celu wykonania przez procesor komputera. Odróżnia to C++ od nowszych języków, takich jak C# i Java. Są to języki interpretowane. Oznacza to, że nie są one bezpośrednio wykonywane przez procesor, ale najpierw przekazywane do innego programu odpowiedzialnego za obsługę tego kodu. Programy napisane w Javie są wykonywane przy użyciu *Java Virtual Machine* (JVM), zaś za wykonywanie programów utworzonych w C# odpowiedzialny jest *Common Language Runtime* (CLR).

Dzięki temu, że C++ jest językiem kompilowanym zawczasu, nadal znajduje szerokie zastosowaniu w tych obszarach, w których bezwzględnie najważniejsza jest wydajność. Najbardziej oczywistym obszarem, w którym C++ nadal jest najczęściej używanym językiem programowania, jest branża gier wideo. C++ pozwala programistom pisać aplikacje, które w pełni wykorzystują dostępną architekturę systemu. Czytelnik próbujący swoich sił jako programista C++ mógł się już spotkać z takimi zwrotami, jak *spójność pamięci podręcznej* (*cache coherency*). Nie ma zbyt wielu innych języków, które pozwalają zoptymalizować nasze aplikacje, aby najlepiej pasowały do indywidualnych procesorów, na których ma działać program. Książka ta przedstawia również niektóre pułapki, które mogą wpływać na wydajność naszych aplikacji w różnych sytuacjach i pokazuje techniki radzenia sobie z takimi problemami.

Nowoczesny C++ znajduje się dziś w takim okresie rozwoju, w którym obserwujemy ciągłe uaktualnienia jego cech. Nie zawsze tak było. Pomimo tego, że język programowania C++ jest w użyciu od początku lat osiemdziesiątych XX wieku, ustandaryzowania doczekał się dopiero w roku 1998. Niewielkie uzupełnienie i ujednoznacznienie tego standardu zostało opublikowane w roku 2003 i jest znane jako C++03. Aktualizacja z roku 2003 nie dodała do języka żadnych nowych funkcji; objaśniła jednak część

istniejących funkcjonalności, które zostały wcześniej pominięte. Jedną z nich było uzupełnienie specyfikacji szablonu wektorowego w Standard Template Library (STL), określające, że elementy wektora muszą być przechowywane w ciągłym obszarze pamięci. Standard C++11 został opublikowany w roku 2011 i ujrzeliśmy wówczas wielką aktualizację języka C++. Język uzyskał funkcjonalności potrzebne dla uogólnionego systemu dedukcji typów poza szablonami, obsługę dla wyrażeń lambda i domknięć, wbudowaną bibliotekę współbieżności i wiele więcej. Standard C++14 przyniósł mniejszą aktualizację języka i w ogólności oparty jest na funkcjach już dostarczanych przez C++11. Zostały wyjaśnione takie funkcjonalności, jak automatyczna dedukcja typu zwracanego przez funkcje, wyrażenia lambda zostały uzupełnione o nowe możliwości, a także pojawiły się nowe sposoby definiowania poprawnie typowanych wartości literałów. Standard C++ 17 wprowadził takie funkcjonalności, jak *folds* i wyrażenia *static if*. Teraz wersja C++20 oferuje pewną liczbę nowych wydajnych funkcji, takich jak *moduły* i *koncepty*, aby wymienić choćby niektóre z ulepszeń, które sprawiają, że język staje się jeszcze potężniejszy.

W książce tej staram się pokazać, jak pisać przenośny, zgodny ze standardami kod C++20. W chwili pisania tych słów możliwe jest stosowanie wielu nowych funkcji kodu C++20 na komputerach systemów Windows, Linux i OS X, o ile używamy kompilatora udostępniającego wszystkie te funkcje języka. Jako że w chwili powstawania książki nie wszystkie standardy wersji 20 zostały formalnie zaakceptowane, a tym bardziej nie zaimplementowano ich w wielu kompilatorach, konieczne może być użycie więcej niż jednego narzędzia deweloperskiego w celu zaimplementowania różnych funkcji w naszych projektach. Biorąc to pod uwagę, w książce używam trzech różnych narzędzi: Clang jako kompilatora w systemach Windows i Ubuntu oraz Xcode w OS X, wraz z Microsoft Visual Studio 19 lub późniejszym na platformach Windows i Mac. W reszcie tego rozdziału skupię się na oprogramowaniu wymaganym dla pisania programów w C++, zanim przejdę do uzyskania niektórych bardziej popularnych opcji dostępnych dla systemów operacyjnych Windows, OS X i Linux.

# Przepis 1-1. Znajdowanie edytora tekstu

## Problem

Programy C++ są konstruowane z wielu różnych plików źródłowych, które programista lub programiści muszą utworzyć i edytować. Pliki źródłowe są po prostu plikami tekstowymi, które występują głównie w dwóch różnych typach: jako pliki nagłówkowe oraz właściwego kodu źródłowego. Pliki nagłówkowe służą do udostępniania informacji o naszych typach i klasach pomiędzy różnymi plikami, zaś pliki źródłowe w ogólności zawierają metody oraz faktyczny kod budujący program. Przy używaniu C++ w wersji 20 programiści mają teraz możliwość używania modułów zamiast tradycyjnych plików nagłówkowych, co pozwala na krótsze czasy kompilacji i prostszy projekt implementacji.

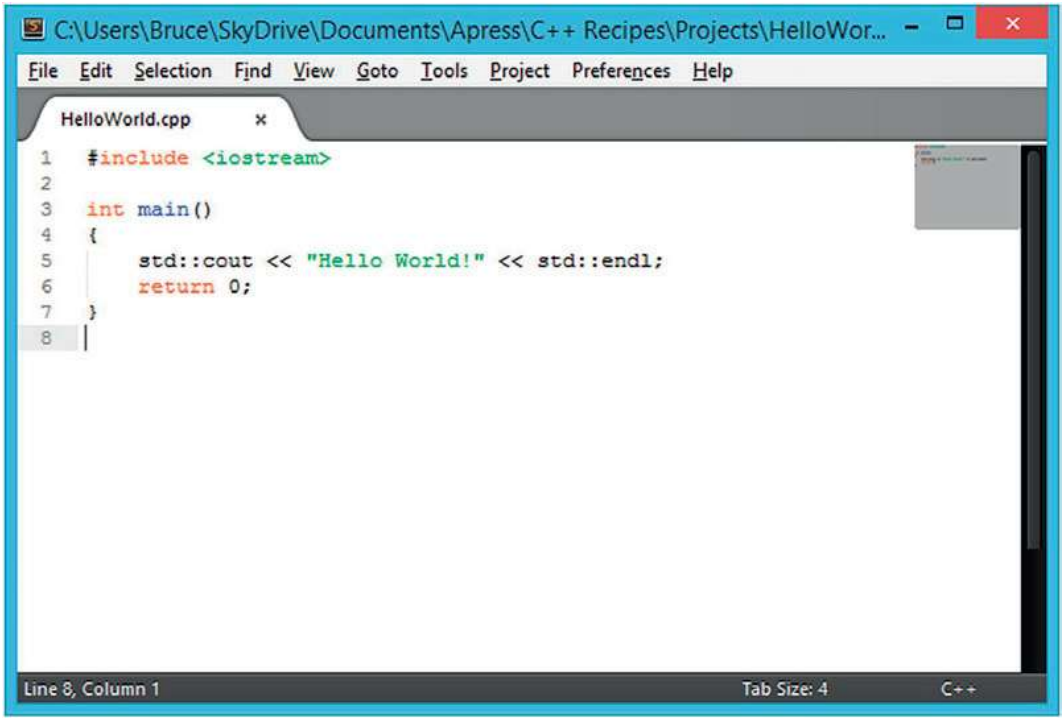
## Rozwiązanie

Zgodnie z powyższym, edytor tekstowy jest pierwszym ważnym składnikiem oprogramowania niezbędnego przy pisaniu programów w języku C++. Istnieje bogaty wybór doskonałych edytorów dla różnych platform. W mojej opinii najlepsze możliwości w chwili obecnej oferują bezpłatny Notepad++ dla Windows oraz Sublime Text 2, który, choć nie jest darmowy, jest dostępny dla wszystkich głównych systemów operacyjnych. Rysunek 1-1 pokazuje zrzut ekranu wykonany w Sublime Text 2. Vim oraz gvim to również doskonałe opcje dostępne dla wszystkich trzech systemów operacyjnych. Edytory te udostępniają wiele zaawansowanych funkcji i są świetnym wyborem dla kogoś, kto zechce się ich nauczyć.

---

**Uwaga** Nie należy śpieszyć się z wyborem „jedynie słusznego” edytora tekstowego. Niektóre z przepisów pokazanych w dalszej części tego rozdziału prezentują zintegrowane środowiska programistyczne (*integrated development environment* – IDE), które zawierają całość oprogramowania wymaganego do pisania, kompilowania i debugowania aplikacji C++. W istocie ustalenie, które z nich będzie najlepsze, jest kwestią preferencji użytkownika i używanej platformy. Warto wypróbować używanie zarówno różnych edytorów tekstowych, jak i rozbudowanego IDE (albo kilku!), aby się przekonać, które najlepiej będą pasować do przepływu pracy. Zapewne okaże się, że ostatecznie będziemy używać więcej niż jednego z nich.

---




**Rysunek 1-1.** *Zrzut ekranu z edytora tekstu Sublime Text 2*

Rysunek 1-1 pokazuje jedną z najważniejszych cech *dobrego* edytora tekstowego: powinien być w stanie wyróżniać w kodzie źródłowym różne rodzaje słów kluczowych. W prostym programie Hello World widocznym na rysunku 1-1 możemy zauważyć, że Sublime Text jest w stanie rozpoznać i wyróżnić słowa kluczowe języka C++, a mianowicie `include`, `int` oraz `return`[1]. Co więcej, dodał on również wyróżnienie innym kolorem nazwy funkcji `main` oraz ciągów `<iostream>` i `"Hello World!"`. W miarę nabierania doświadczenia w pisaniu kodu w wybranym edytorze często będziemy musieli szybko przeglądać zawartość plików źródłowych, skupiając się na interesujących obszarach, zaś wyróżnianie składni jest ważnym czynnikiem ułatwiającym ten proces.

Inną funkcjonalnością oferowaną przez część edytorów jest *autouzupełnianie*, albo zgodnie z nomenklaturą firmy Microsoft, *IntelliSense*. Ta funkcja jest specyficzna dla języka i pozwala przyśpieszyć tworzenie oprogramowania dzięki automatycznemu dopełnianiu lub rozwijaniu opcji związanych z funkcjami i metodami danego języka. W przypadku edytora Sublime i języka C++ można zainstalować wtyczkę *ClangAutoComplete*

1 Kolory te są mało odróżnialne w książce drukowanej tylko jednym kolorem. Zdecydowanie polecam sięganie do kodu źródłowego dostępnego na GitHubie i otwieranie przykładów w jednym z zalecanych edytorów, aby móc rzeczywiście ocenić opisane różnice.

dla Sublime. Choć nie jest darmowy, Sublime oferuje długi okres próbny, zanim zdecydujemy się na zakup licencji.

# Przepis 1-2. Instalowanie Clang w Ubuntu

## Problem

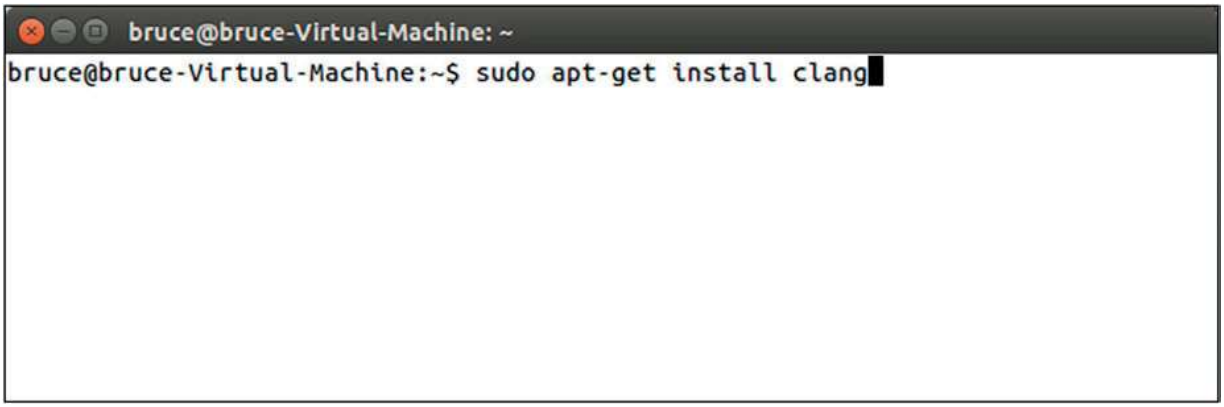
Zamierzamy kompilować programy C++ wykorzystujące najnowsze funkcje języka na komputerze pracującym pod kontrolą systemu operacyjnego Ubuntu Linux.

## Rozwiązanie

Kompilator Clang obsługuje wszystkie najnowsze funkcje języka C++20, zaś biblioteka libstdc++ wspiera wszystkie funkcje standardowej biblioteki szablonów (STL) C++20.

## Jak to działa

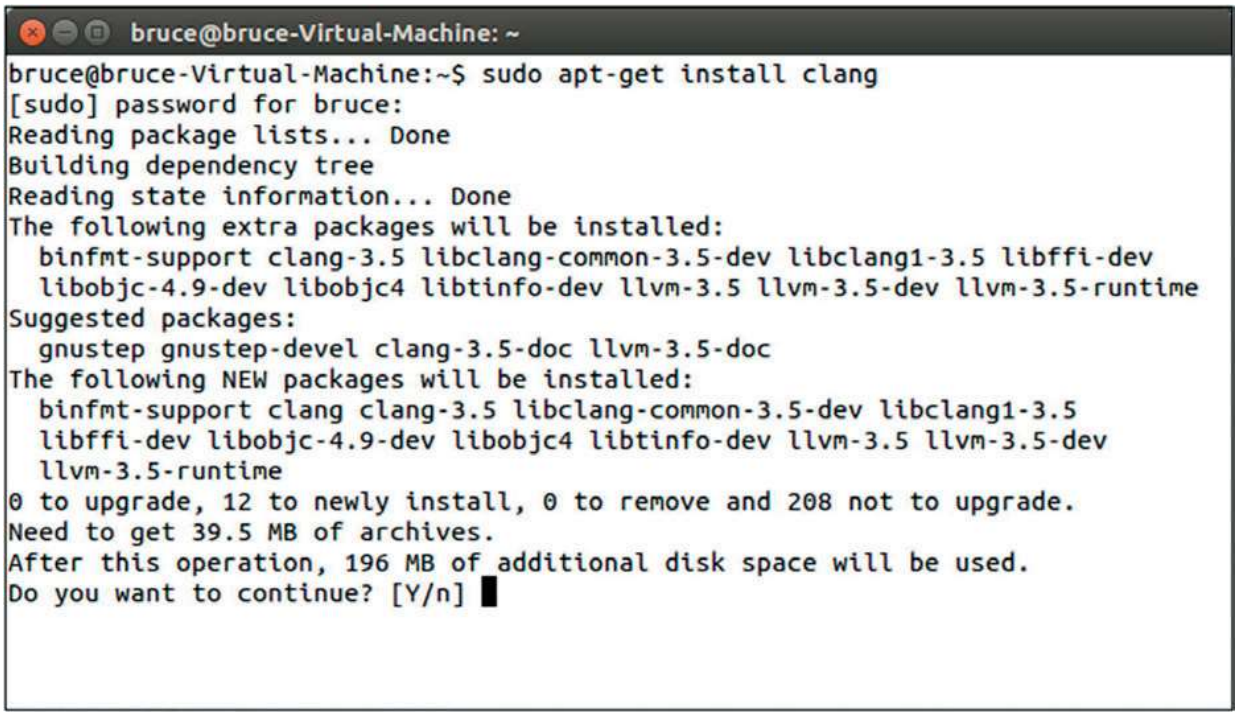
System operacyjny Ubuntu ma domyślnie skonfigurowane repozytoria pakietów, które pozwalają bez większego trudu zainstalować Clang. Można to osiągnąć, używając polecenia `apt-get` w oknie terminala. Rysunek 1-2 pokazuje polecenie, które należy wprowadzić w celu zainstalowania kompilatora Clang.




**Rysunek 1-2.** *Okno terminala systemu Ubuntu pokazujące polecenie wymagane do zainstalowania Clang*

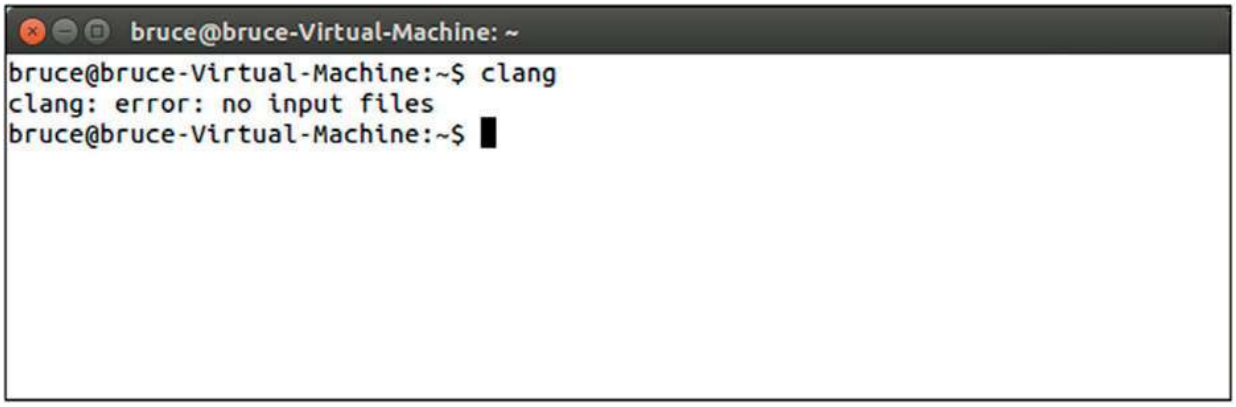
Aby zainstalować Clang, w oknie terminala wpisujemy następujące polecenie: `sudo apt-get install clang`. Wykonanie tego polecenia sprawi, że Ubuntu przeszuka

swoje repozytoria i wypracuje wszystkie zależności wymagane dla zainstalowania Clang. Po ukończeniu tego procesu pojawi się monit o potwierdzenie, że chcemy zainstalować Clang i wymagane przez niego pakiety. Monit ten pokazany jest na rysunku 1-3.




**Rysunek 1-3.** *Monit potwierdzenia instalowania zależności przez apt-get*

W tym momencie wystarczy nacisnąć Enter, aby kontynuować, gdyż domyślną opcją jest Yes (tak). Ubuntu pobierze i zainstaluje całe oprogramowanie wymagane w celu zainstalowania Clang na naszym komputerze. Możemy sprawdzić, że to zadanie ukończyło się sukcesem, wywołując polecenie `clang`. Rysunek 1-4 pokazuje, jak powinno wyglądać wyjście polecenia, jeśli wszystko odbyło się z powodzeniem.




**Rysunek 1-4.** *Udana instalacja Clang w Ubuntu*

# Przepis 1-3. Instalowanie Clang w Windows

## Problem

Będziemy chcieli kompilować programy oparte na języku C++20 w systemie operacyjnym Windows.

## Rozwiązanie

Do zainstalowania Clang i kompilowania aplikacji możemy wykorzystać środowisko Cygwin.

## Jak to działa

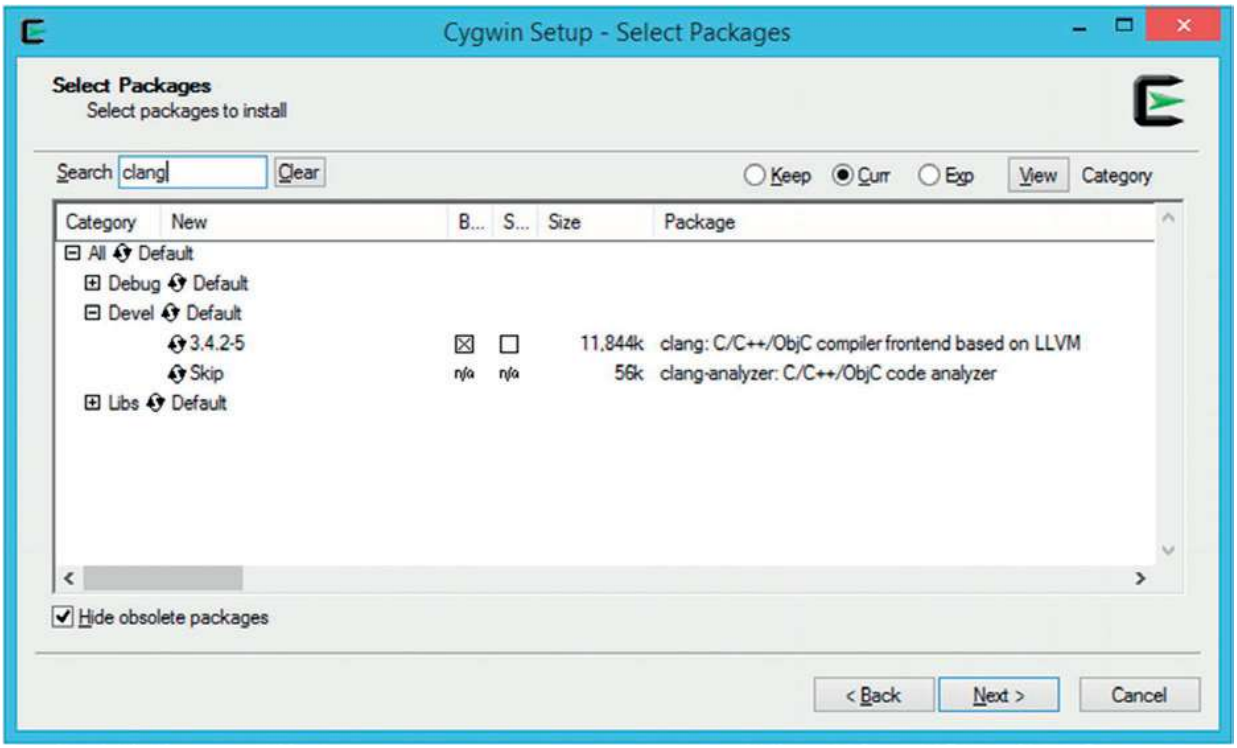
Cygwin udostępnia podobne do uniksowego środowisko wiersza polecenia (trybu tekstowego) dla komputerów systemu Windows. Jest to idealne rozwiązanie dla kompilowania programów przy użyciu Clang, gdyż środowisko Cygwin jest instalowane ze wstępnie skonfigurowanymi repozytoriami pakietów zawierającymi wszystko, co jest niezbędne dla instalacji i używania kompilatora Clang na komputerach Windows.

Instalator Cygwin można znaleźć w witrynie projektu pod adresem *www.cygwin.com*. Trzeba pamiętać, aby pobrać 32-bitową wersję instalatora, gdyż domyślne pakiety dostarczane przez Cygwin obecnie działają tylko w środowisku 32-bitowym.

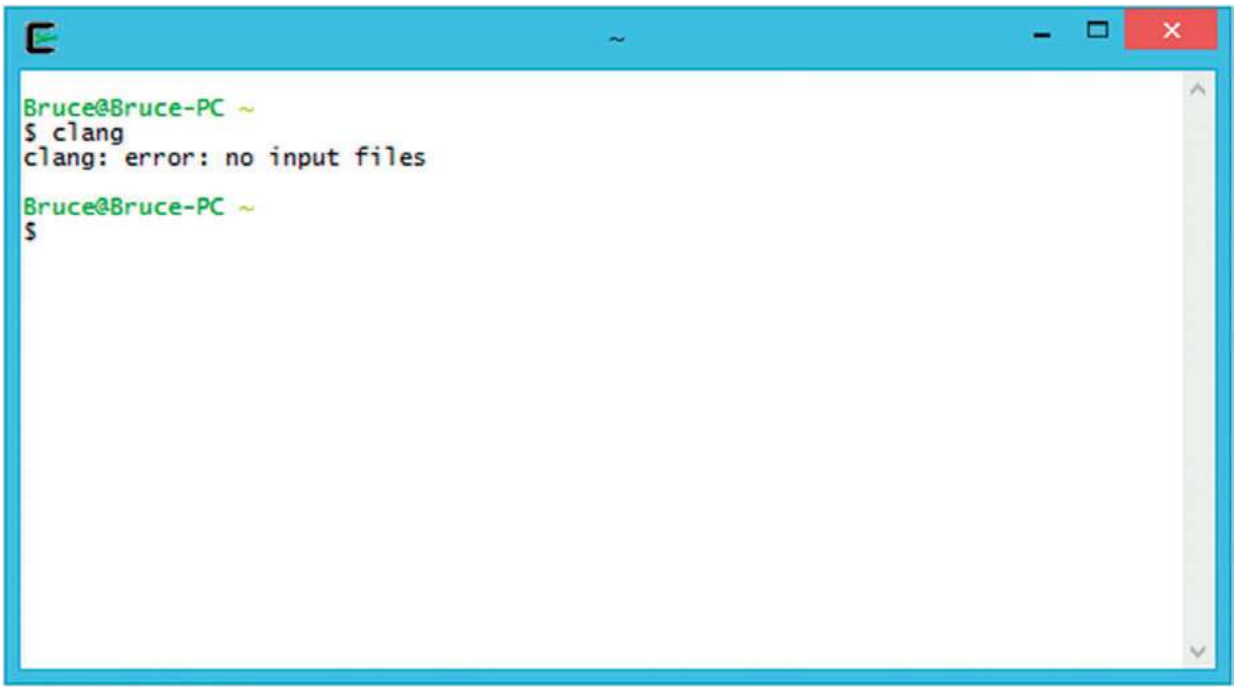
Po pobraniu instalatora należy go uruchomić, klikając w kolejnych ekranach, aż pojawi się lista pakietów do zainstalowania. W tym miejscu należy wybrać pakiety `Clang`, `make` oraz `libstdc++`. Rysunek 1-5 pokazuje instalator Cygwin z zaznaczonym pakietem Clang.

Pakiety do zainstalowania można zaznaczyć, klikając obszar **Skip** (pomiń) w wierszu danego pakietu, aby usunąć domyślne zaznaczenie pola wyboru. Jednokrotne kliknięcie Skip domyślnie powoduje przejście do najnowszej wersji pakietu. Należy wybrać najnowsze pakiety Clang, make i libstdc++. Po zaznaczeniu wszystkich trzech możemy kliknąć **Next**, co powoduje przejście do okna potwierdzającego instalowanie komponentów wymaganych przez te trzy pakiety.

Po pobraniu i zainstalowaniu wszystkich pakietów wymaganych do uruchamiania Clang możemy zweryfikować poprawność instalacji, otwierając terminal Cygwin i wpisując polecenie `clang`. Wynik tego polecenia pokazany jest na rysunku 1-6.




**Rysunek 1-5.** *Wyszukiwanie pakietu Clang w instalatorze środowiska Cygwin*




**Rysunek 1-6.** *Udane uruchomienie Clang w środowisku Cygwin w systemie Windows*

# Przepis 1-4. Instalowanie Clang w OS X

## Problem

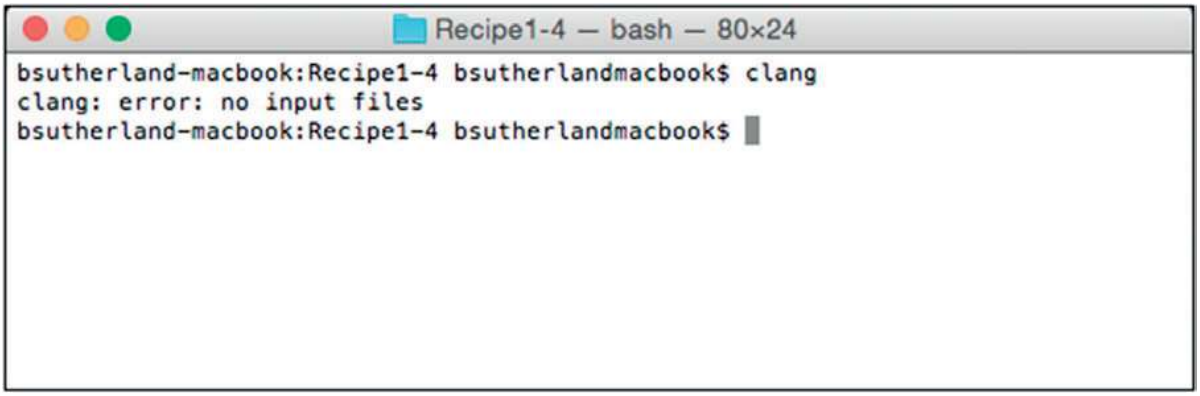
Będziemy chcieli kompilować programy oparte na C++20 na komputerze systemu OS X.

## Rozwiązanie

Środowisko IDE Xcode firmy Apple zawiera Clang jako domyślny kompilator. Zainstalowanie Xcode z witryny App Store dla OS X powoduje również zainstalowanie Clang. Trzeba jednak zwrócić uwagę na to, że system OS X musi być zaktualizowany, aby możliwe było zainstalowanie Xcode. Trzeba się upewnić, że komputer ma dostateczną ilość pamięci RAM, zanim zainstalujemy najnowszą aktualizację systemu!

## Jak to działa

Instalujemy na swoim komputerze najnowszą wersję Xcode z App Store. Po zainstalowaniu Xcode można otworzyć okno terminala używając Spotlight, po czym wpisać clang, aby upewnić się, że kompilator został zainstalowany. Rysunek 1-7 pokazuje, jak to powinno wyglądać.




**Rysunek 1-7.** *Uruchamianie Clang w OS X po zainstalowaniu Xcode*

# Przepis 1-5. Kompilowanie pierwszego programu C++

## Problem

Chcemy użyć swojego komputera do wygenerowania wykonywalnej aplikacji z kodu źródłowego C++.

## Rozwiązanie

Generowanie plików wykonywalnych z plików źródłowych w języku C++ obejmuje dwa kroki: kompilowanie i konsolidowanie. Kroki wykonane w przepisie 1-2, 1-3 albo 1-4, zależnie od używanego systemu operacyjnego, zapewniły zainstalowanie całego oprogramowania niezbędnego do budowania aplikacji z plików źródłowych C++20. Jesteśmy zatem gotowi do zbudowania pierwszego programu w języku C++20. Tworzymy folder, który będzie zawierał projekt, po czym dodajemy plik tekstowy o nazwie HelloWorld.cpp. W pliku tym wprowadzamy kod pokazany w listingu 1-1 i zapisujemy zmiany.

**Listing 1-1.** *Pierwszy program C++20*

```
#include <iostream>
#include <string>

int main(void)
{
    using namespace std::string_literals;

    auto output = "Hello World!"s;
    std::cout << output << std::endl;

    return 0;
}
```

Kod pokazany w tym listingu jest programem C++, który skompiluje się jedynie przy użyciu kompilatora zgodnego ze standardem C++14 lub późniejszym. Przepisy 1-2 do 1-4 w tym rozdziale zawierają instrukcje, jak uzyskać kompilator, którego będzie można użyć do skompilowania wielu spośród proponowanych funkcjonalności C++20

(według stanu z końca roku 2019) dla systemów Windows, Ubuntu i OS X. Będziemy mogli zbudować działającą aplikację, o ile utworzymy folder i plik źródłowy zawierający kod z listingu 1-1. Zadanie budowania aplikacji zrealizujemy przy użyciu narzędzia `make`. Tworzymy plik o nazwie makefile w tym samym folderze, co plik HelloWorld.cpp. Zwróćmy uwagę, że makefile nie może mieć rozszerzenia nazwy, co może się wydawać nieco dziwne deweloperom przyzwyczajonym do systemu operacyjnego Windows. Niemniej jednak jest to zupełnie normalne w systemach operacyjnych opartych na Unix, takich jak Linux czy OS X. Do pliku makefile wpisujemy kod pokazany w listingu 1-2.

**Listing 1-2.** *Makefile potrzebny do skompilowania kodu z listingu 1-1*

```
HelloWorld: HelloWorld.cpp
	clang++ -g -std=c++1y HelloWorld.cpp -o HelloWorld
```

---

**Uwaga** Odstęp widoczny przed poleceniem `clang++` w tym listingu to tabulator. Nie można zastąpić tego tabulatora spacjami, gdyż wówczas `make` nie zdoła wykonać kompilacji. Trzeba zawsze się upewnić, że formuły kompilatora w plikach makefile zawsze zaczynają się tabulatorami.

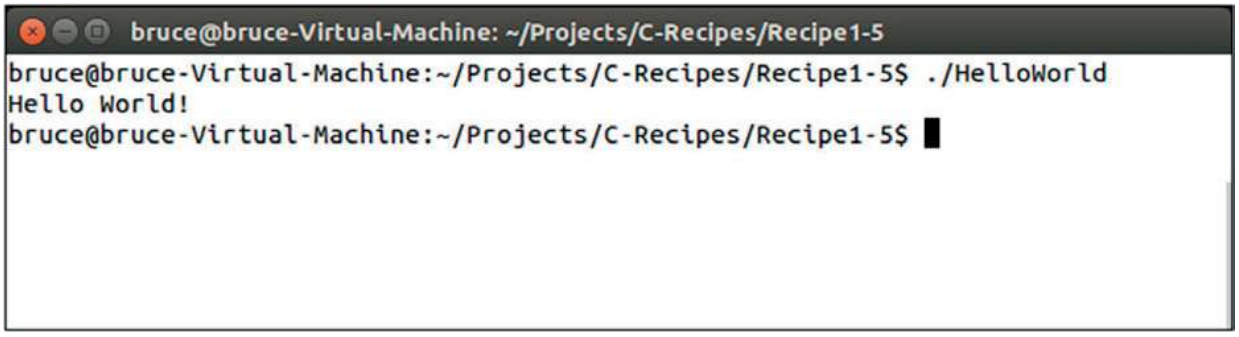
---

Tekst pokazany w listingu 1-2 zawiera instrukcje potrzebne do zbudowania aplikacji z pliku źródłowego HelloWorld.cpp. Pierwsze słowo w pierwszym wierszu to nazwa pliku docelowego makefile. Jest to nazwa, którą otrzyma plik wykonywalny aplikacji po ukończeniu procesu budowania. W tym przypadku zbudujemy plik wykonywalny o nazwie HelloWorld. Po tej nazwie wymienione są komponenty wymagane do zbudowania programu. W tym przypadku mamy tu plik HelloWorld.cpp jako jedyne wymaganie wstępne, gdyż jest to jedyny plik źródłowy używany do utworzenia pliku wykonywalnego.

Po celu i wymaganiach następuje lista formuł (poleceń), które muszą zostać wykonane w celu zbudowania naszej aplikacji. W tym małym przykładzie mamy wiersz wywołujący kompilator clang++, który wygeneruje kod wykonywalny na podstawie pliku HelloWorld.cpp. Przekazany do clang++ parametr `-std=c++1y` żąda od Clang kompilacji z wykorzystaniem standardu C++14 języka, zaś przełącznik `-o` określa nazwę wynikowego pliku kodu maszynowego generowanego przez proces kompilacji.

Przechodzimy do folderu utworzonego w celu przechowania pliku źródłowego oraz makefile, używając powłoki poleceń, takiej jak cmd w systemie Windows albo Terminal w systemach Linux i OS X, po czym wpisujemy `make`. Spowoduje to wywołanie programu GNU make, który automatycznie odczyta i wykona nasz makefile. Wyjściowy plik

wykonywalny zostanie utworzony w tym samym folderze i będzie można go uruchomić z wiersza polecenia. W rezultacie powinniśmy zobaczyć tekst „Hello World!” jako odpowiedź w oknie powłoki. Rysunek 1-8 pokazuje, jak będzie to wyglądać w oknie terminala Ubuntu.




**Rysunek 1-8.** *Wyjście programu HelloWorld w terminalu Ubuntu*

# Przepis 1-6. Debugowanie programów C++ przy użyciu GDB w środowisku Cygwin lub Linux

## Problem

Piszemy program w języku C++20 i chcemy mieć możliwość debugowania tej aplikacji z wiersza polecenia.

## Rozwiązanie

Zarówno Cygwin dla Windows, jak i systemy operacyjne Linux, takie jak Ubuntu, pozwalają na instalację i wykorzystanie GDB – debugera trybu wiersza polecenia dla aplikacji C++.

## Jak to działa

Do zainstalowania debugera GDB możemy wykorzystać instalator Cygwin w systemie Windows albo menedżer pakietów, zawarty w preferowanej dystrybucji systemu Linux. Zapewni to dostęp do debugera C++ wiersza polecenia, którego będziesz mógł używać do badania funkcjonalności programów napisanych w C++. Można to przećwiczyć, wykorzystując plik źródłowy, makefile i aplikację wygenerowaną w przepisie 1-5.

Aby wygenerować informacje debugowania dla tego programu, trzeba uaktualnić plik makefile, aby zawierał zawartość listingu 1-3, po czym ponownie uruchomić `make`, aby wygenerować debugowalny plik wykonywalny.

**Listing 1-3.** *Makefile generujący debugowalny program*

```
HelloWorld: HelloWorld.cpp
        clang++ -g -std=c++1y HelloWorld.cpp -o HelloWorld
```

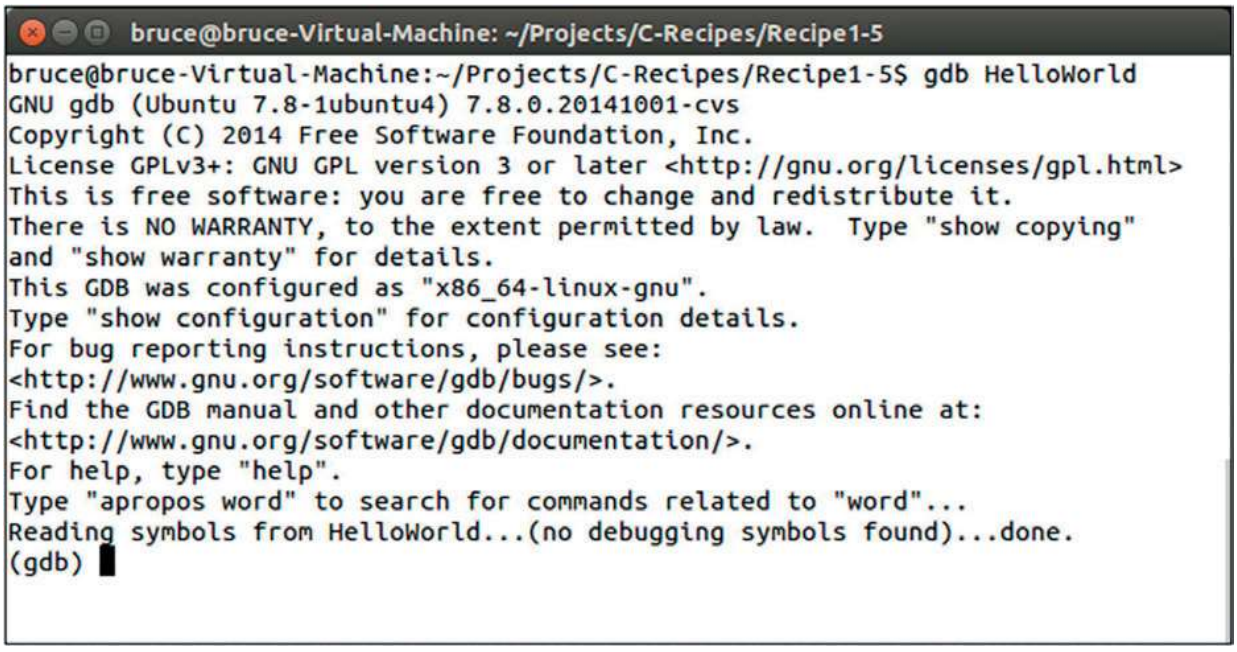
Po wykonaniu przepisu 1-5, zmodyfikowaniu makefile, aby jego zawartość była zgodna z listingiem 1-3 i ponownym wygenerowaniu pliku wykonalnego, możemy uruchomić GDB względem tej aplikacji, przechodząc do tego folderu w wierszu polecenia i wpisując `gdb HelloWorld`. Nowy przełącznik `-g` przekazany do Clang w makefile z listingu 1-3 żąda od kompilatora wygenerowania dodatkowych informacji dołączanych do aplikacji, które pomogą debugerowi dostarczyć dokładne informacje o programie, który jest wykonywany wewnątrz debugera.

---

**Uwaga** Może pojawić się komunikat informujący, że program jest już aktualny, jeśli został wcześniej skompilowany. W takim przypadku wystarczy po prostu usunąć istniejący plik wykonywalny przed wywołaniem **make**.

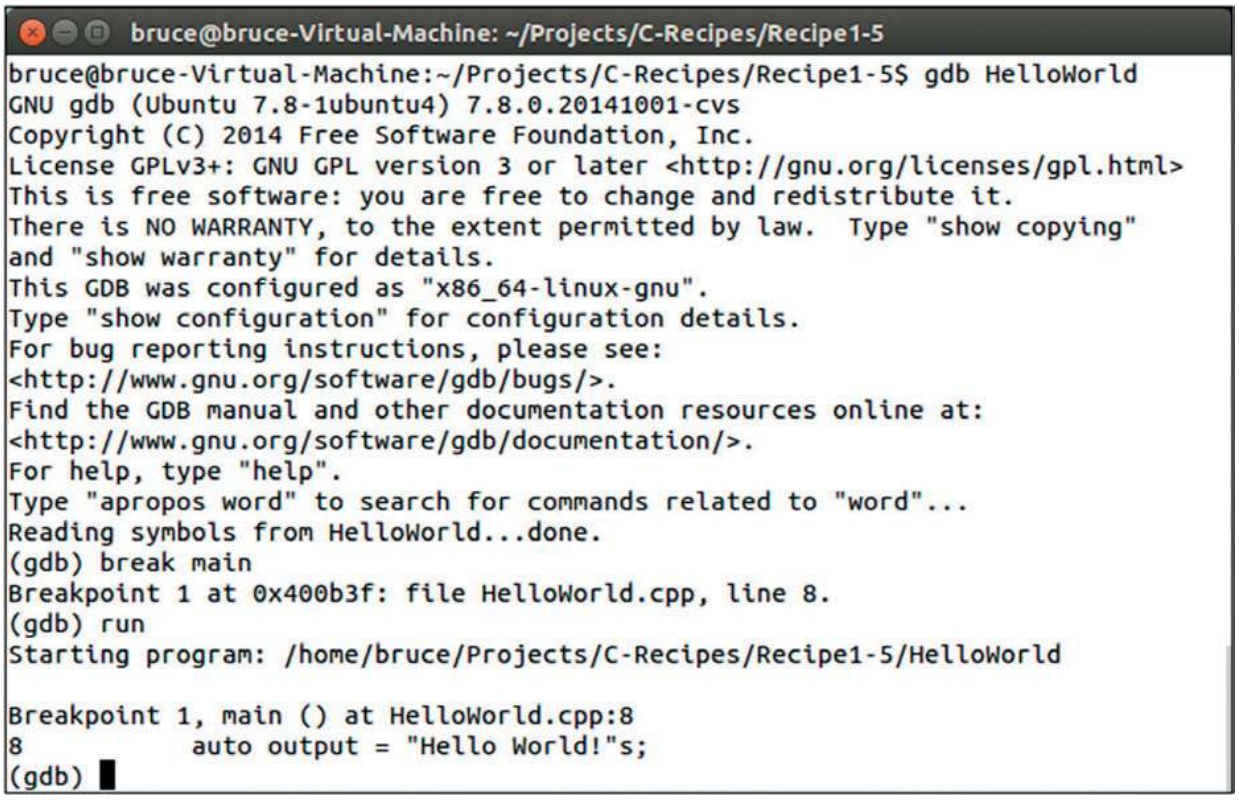
---

Uruchomienie GDB dla HelloWorld powinno dać wynik pokazany na rysunku 1-9.




**Rysunek 1-9.** *Działające wystąpienie GDB*

Mamy teraz działający debuger, którego możemy użyć w celu badania programu w trakcie jego działania. Program ten po starcie GDB jeszcze nie został uruchomiony. Pozwala to skonfigurować jakieś punkty przerwań, zanim rozpoczniemy analizę. Do ustawienia punktu przerwania możemy użyć polecenia `break` (albo skrótu `b` tego polecenia). Wpisujemy `break main` w znaku zachęty GDB i naciskamy Enter. Spowoduje to przekazywanie przez GDB kolejnych poleceń na ekran wraz z adresem programu, w którym został ustawiony punkt przerwania, a także nazwą pliku i numerem wiersza wykrytym dla przekazanej funkcji. Teraz możemy wpisać `run` w oknie terminala, aby rozpocząć wykonywanie programu i zastopować GDB w punkcie przerwania. Wyjście powinno odpowiadać pokazanemu na rysunku 1-10.




**Rysunek 1-10.** *Wyjście widoczne w chwili zastopowania GDB na punkcie przerwania ustawionego w procedurze main*

W tym momencie dostępnych jest kilka opcji pozwalających na kontynuację wykonywania programu. Poniżej pokazana jest lista najbardziej popularnych poleceń:

`step`

Polecenie to służy do przejścia do funkcji, która ma być wywołana w bieżącym wierszu.

`next`

To polecenie powoduje wykonanie bieżącego wiersza i zatrzymanie w następnym wierszu tej samej funkcji.

`finish`

Polecenie `finish` służy do wykonania całego kodu pozostałego w bieżącej funkcji i zatrzymania w kolejnym wierszu funkcji, która wywołała bieżącą funkcję.

`print <name>`

Polecenie `print` uzupełnione nazwą zmiennej pozwala wypisać wartość zmiennej w programie.

`break`

Polecenie `break` może zostać użyte w połączeniu z numerem wiersza, nazwą funkcji albo nazwą pliku źródłowego i numeru wiersza w celu ustawienia punktu przerwania w kodzie źródłowym programu.

`continue`

Polecenie `continue` powoduje wznowienie wykonywania kodu po jego zastopowaniu w punkcie przerwania.

`until`

Polecenie `until` powoduje kontynuowanie wykonywania pętli i zatrzymanie w pierwszym wierszu bezpośrednio po ukończeniu tej pętli.

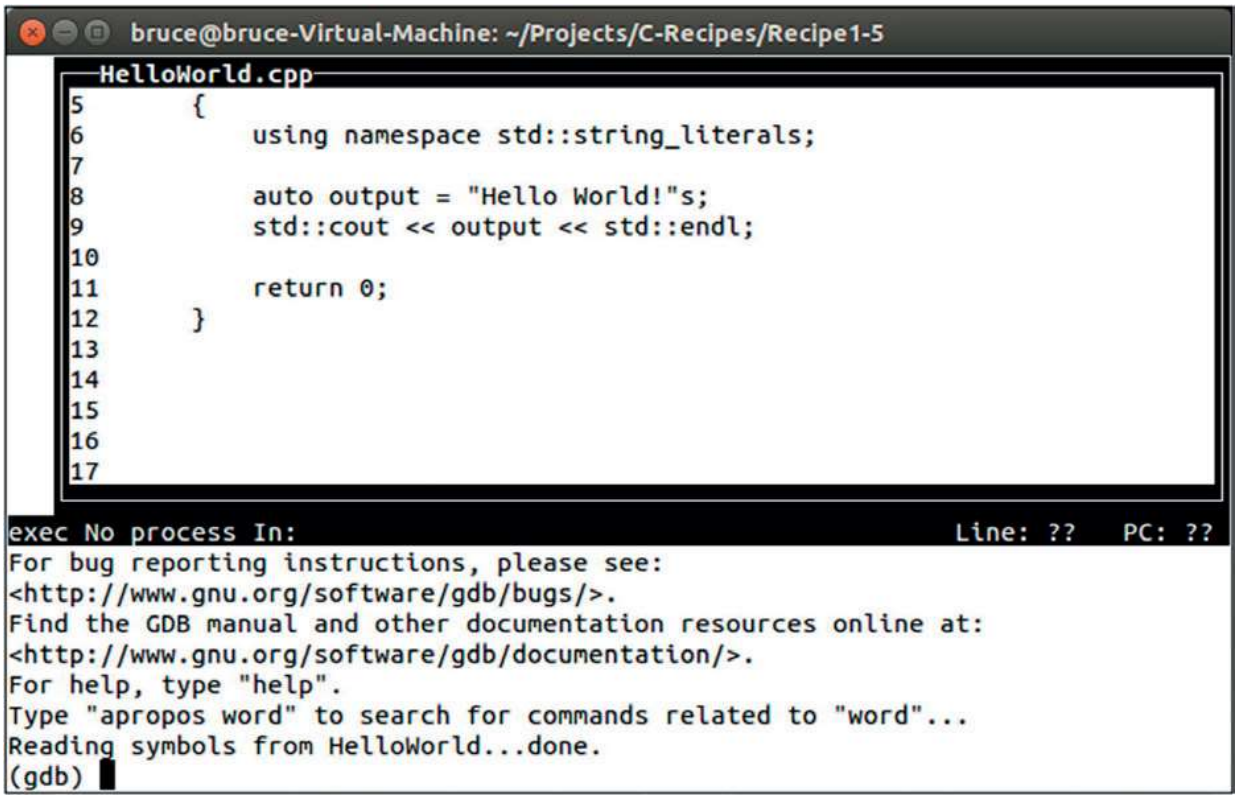
`info`

Polecenie to może być używane albo z poleceniem `locals`, albo `stack` i powoduje wyświetlenie informacji o bieżących zmiennych lokalnych albo o stanie stosu programu.

`help`

Wpisanie `help` uzupełnionego o dowolne polecenie wyświetla informacje o różnych sposobach wykorzystywania tego polecenia.

Debuger GDB można uruchamiać również z poleceniem `-tui`. Zapewnia to wyświetlenie w górnej części okna pliku źródłowego, który jest aktualnie debugowany. Rysunek 1-11 pokazuje, jak to może wyglądać.




**Rysunek 1-11.** *GDB z oknem źródła*

# Przepis 1-7. Debugowanie programów C++ w OS X

## Problem

System operacyjny OS X nie udostępnia żadnej prostej metody instalowania i posługiwania się GDB.

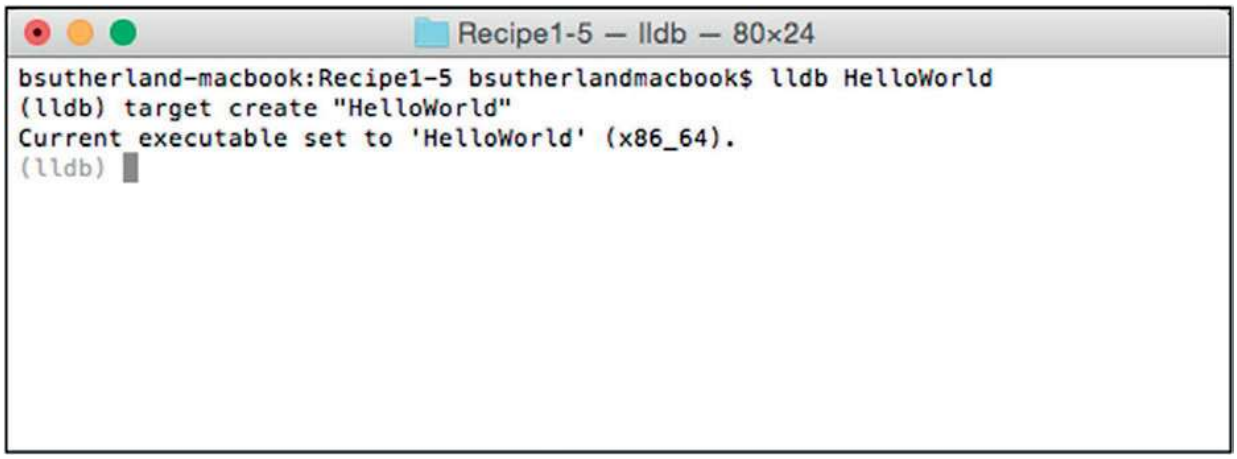
## Rozwiązanie

Xcode wyposażony jest w debuger LLDB, którego można używać w trybie wiersza polecenia zamiast GDB.

## Jak to działa

Debuger LLDB jest zasadniczo bardzo podobny do debugera GDB użytego w przepisie 1-6. Przechodzenie pomiędzy GDB a LLDB sprowadza się jedynie do nauczenia się, jak wykonywać te same proste zadania w każdym z nich, używając poleceń udostępnianych przez poszczególne narzędzia.

Można uruchomić LLDB dla pliku wykonywalnego HelloWorld, przechodząc w Terminalu do katalogu zawierającego HelloWorld i wpisując `lldb HelloWorld`. W rezultacie zobaczymy wyjście podobne do pokazanego na rysunku 1-12.




**Rysunek 1-12.** *Debuger LLDB działający w terminalu OS X*

---

**Uwaga** Konieczne jest skompilowanie programu przy użyciu przełącznika `-g`. Listing 1-3 pokazuje, gdzie należy go umieścić.

---

Po uruchomieniu LLDB, jak na rysunku 1-12, możemy ustawić punkt przerwania w pierwszym wierszu procedury main, wpisując `breakpoint set -f HelloWorld.cpp -l 8` albo skrótowo `b main`. Następnie możemy użyć polecenia `run`, aby rozpocząć wykonywanie programu – zostanie ono wstrzymane w dopiero co ustawionym punkcie przerwania. Gdy program się zatrzyma, możemy użyć polecenia `next`, aby przejść przez bieżący wiersz i zatrzymać się w kolejnym. Można również użyć polecenia `step`, aby wejść do funkcji wywoływanej w bieżącym wierszu i zastopować działanie w pierwszym wierszu tej funkcji. Polecenie `finish` spowoduje wykonanie i wyjście z bieżącej funkcji.

Działanie LLDB można zakończyć wpisując `q` i naciskając Enter. Ponownie uruchamiamy LLDB i wpisujemy `breakpoint set -f HelloWorld.cpp -l 9`. Następnie

wpisujemy polecenie `run`, a LLDB wypisze kod źródłowy sąsiadujący z miejscem, w którym aplikacja została zatrzymana.

Możemy teraz wpisać polecenie `print output`, aby zobaczyć wartość przechowywaną przez zmienną wyjściową. Możemy również użyć polecenia `frame variable`, aby zobaczyć wszystkie zmienne lokalne w bieżącej ramce stosu.

Te proste polecenia pozwalają używać debugera LLDB wystarczająco dla potrzeb pracy z przykładami przedstawionymi w tej książce. Poniższej listy można używać jako poręcznej ściągawki przy korzystaniu z LLDB:

`step`
: Polecenie `step` służy do wejścia do funkcji, która ma być wywołana w bieżącym wierszu.

`next`
: Polecenie `next` jest używane do przejścia przez bieżący wiersz i zatrzymania w kolejnym wierszu tej samej funkcji.

`finish`
: Polecenie `finish` pozwala wykonać cały kod pozostały w bieżącej funkcji i zastopowanie w następnym wierszu funkcji, która wywołała bieżącą funkcję.

`print <name>`
: Polecenie `print` uzupełnione nazwą zmiennej służy do wypisania wartości tej zmiennej w naszym programie.

`breakpoint set --name <name>`
`breakpoint set –file <name> --line <number>`
: Polecenia `breakpoint` można użyć w połączeniu z numerem wiersza, nazwą funkcji albo nazwą pliku źródłowego i numerem wiersza w celu ustawienia punktu przerwania w kodzie źródłowym programu.

`help`
: Wpisanie `help` uzupełnionego o dowolną nazwę polecenia zwraca informacje o działaniu tego polecenia i różnych sposobach jego stosowania.

# Przepis 1-8. Przełączanie trybów kompilacji C++

## Problem

Chcemy mieć możliwość przełączania się pomiędzy różnymi standardami C++ przed kompilowaniem programów.

## Rozwiązanie

Przełącznik `std` udostępniany przez kompilator Clang pozwala wyspecyfikować standard C++, który ma być użyty podczas kompilacji.

## Jak to działa

Kompilator Clang domyślnie pracuje zgodnie ze standardem C++98. Możemy użyć argumentu `std` dla polecenia `clang++`, aby nakazać kompilatorowi użycie innego standardu, niż domyślny. Listing 1-4 pokazuje zawartość pliku makefile, który został skonfigurowany do budowania programu przy użyciu standardu C++17. W tym przypadku kompilator Clang będzie obsługiwać standard C++17. Użycie trybu `2a` powoduje wykorzystanie wersji C++20.

**Listing 1-4.** *Budowanie przy użyciu C++17*

```
HelloWorld: HelloWorld.cpp
      clang++ -std=c++17 HelloWorld.cpp -o HelloWorld
```

Plik makefile z listingu 1-4 pokazuje, jak możemy nakazać Clang wykonanie kompilacji pliku źródłowego z wykorzystaniem standardu C++17. Ten przykład został napisany przy użyciu wersji Clang 5. W celu włączenia eksperymentalnej obsługi standardu C++20 (`2a`) w Clang 5 należy użyć opcji `-std=c++2a`.

Listing 1-5 pokazuje, jak możemy skompilować program, używając standardu C++11.

**Listing 1-5.** *Budowanie przy użyciu C++11*

```
HelloWorld: HelloWorld.cpp
      clang++ -std=c++11 HelloWorld.cpp -o HelloWorld
```

W tym listingu użyliśmy opcji `c++11` przełącznika `std`, aby wykonać kompilację zgodnie ze standardem C++11. Na koniec listing 1-6 pokazuje, jak jawnie skonfigurować Clang, aby wykonywał kompilację zgodnie ze standardem C++98.

**Listing 1-6.** *Budowanie przy użyciu C++98*

```
HelloWorld: HelloWorld.cpp
      clang++ -std=c++98 HelloWorld.cpp -o HelloWorld
```

Plik makefile z listingu 1-6 można wykorzystać, aby jawnie nakazać budowanie kodu wynikowego zgodnie ze standardem C++98. Ten sam rezultat uzyskamy, całkowicie pomijając przełącznik `std`, jako że Clang domyślnie wykonuje kompilację przy użyciu standardu C++98.

---

**Uwaga** Nie ma gwarancji, że każdy kompilator będzie domyślnie używać standardu C++98. Jeśli nie ma pewności, jaki standard jest domyślny, należy sprawdzić dokumentację używanego kompilatora. Trzeba również pamiętać, że możemy używać kompilatora Clang dla wielu funkcjonalności C++20; włączenie eksperymentalnego trybu C++20 (2a) umożliwia polecenie wspomniane we wcześniejszym tekście.

---

# Przepis 1-9. Budowanie z użyciem biblioteki Boost

## Problem

Chcemy napisać program wykorzystujący bibliotekę Boost.

## Rozwiązanie

Biblioteka Boost jest dostarczana jako kod źródłowy, który można dołączyć i skompilować wraz z naszą aplikacją.

## Jak to działa

Boost to wielka biblioteka C++, która zawiera doskonałe funkcje najrozmaitszych rodzajów. Omówienie całej biblioteki wykracza poza zakres tej książki – w istocie

wymagałoby oddzielnej pozycji, zapewne obszerniejszej niż ta! Tym niemniej będziemy wykorzystywać jej różne części, a w szczególności bibliotekę formatowania ciągów tekstowych. Bibliotekę Boost można uzyskać z witryny Boost, dostępnej pod adresem *www.boost.org/*.

Z witryny tej możemy pobrać skompresowany folder, który zawiera najświeższą wersję biblioteki Boost. Jedyny folder, który jest bezwzględnie konieczny, aby dołączyć podstawową funkcjonalnośc Boost, jest sam folder `boost`. Pobieramy bieżącą wersję Boost 1.71.0 i tworzymy folder wewnątrz folderu projektu, nazywając go `boost_1_xxx_x` (gdzie *x* oznaczają odpowiedni numer wersji), po czym kopiujemy do tej lokalizacji folder boost z pobranej wersji.

Po przygotowaniu folderu projektu i umieszczeniu w nim pobranej kopii biblioteki Boost możemy dołączyć do swojego kodu źródłowego pliki nagłówkowe Boost. Listing 1-7 pokazuje program wykorzystujący funkcję `boost::format`.

**Listing 1-7.** *Korzystanie z funkcji* `boost::format`

```
#include <iostream>
#include "boost/format.hpp"

using namespace std;

int main()
{
    std::cout << "Enter your first name: " << std::endl;
    std::string firstName;
    std::cin >> firstName;

    std::cout << "Enter your surname: " << std::endl;
    std::string surname;
    std::cin >> surname;

    auto formattedName = str( boost::format("%1% %2%"s) % firstName %
                              surname );
    std::cout << "You said your name is: " << formattedName << std::endl;

    return 0;
}
```

Kod w listingu 1-7 pokazuje, jak dołączyć plik nagłówkowy Boost do pliku źródłowego, a także użyciu funkcji z tego pliku w naszym programie.

**Uwaga** Nie trzeba się martwić tym, że nie wiemy, jak działa funkcja `format`, jeśli nie jest to od razu zrozumiałe; zostanie ona omówiona w rozdziale 3.

Musimy również poinformować kompilator w pliku makefile, gdzie ma szukać plików nagłówkowych; w przeciwnym razie nasz program się nie skompiluje. Listing 1-8 pokazuje zawartość pliku makefile, którego można użyć do zbudowania tego programu.

**Listing 1-8.** *Makefile dla kompilacji z użyciem biblioteki Boost*

```
main: main.cpp
      clang++ -g -std=c++1y -Iboost_1_55_0 main.cpp -o main
```

Plik makefile pokazany w listingu 1-8 przekazuje do programu clang++ opcję `-I`. Opcja ta służy informowaniu Clang, że chcemy dodać wskazany folder do ścieżki przeszukiwania, używanej do lokalizowania plików dołączanych dyrektywą `#include` w kodzie źródłowym. Jak można zauważyć, przekazałem tu folder `boost_1_55_0`, który utworzyłem w folderze mojego projektu. Folder ten zawiera folder boost, co można zauważyć po tym, jak plik nagłówkowy Boost został dołączony w listingu 1-7.

**Uwaga** Jeśli Czytelnik ma trudności z wykonaniem tego przykładu i nie ma pewności, gdzie umieścić pliki nagłówkowe Boost, całość kodu towarzyszącego tej książce można pobrać, klikając przycisk **Download Source Code** na stronie *www.apress.com/9781484257128*.

# Przepis 1-10. Instalowanie Microsoft Visual Studio

## Problem

Chcielibyśmy móc używać na komputerze Mac lub PC (Windows) oprogramowania MS Visual Studio 2019, zapewniającego dobrą obsługę standardu C++20. Visual Studio oferuje obsługę wielu języków programowania, projektów zespołowych, a także integrację z systemem kontroli wersji kodu GitHub. Umiejętność posługiwania się Visual Studio będzie cennym elementem w CV.

## Rozwiązanie

Pobieramy bezpłatne Visual Studio Community Edition dla platformy Mac lub Windows.

## Jak to działa

Firma Microsoft oferuje za opłatą wiele różnych wersji Visual Studio; jednak wersja Community Edition jest bezpłatna. Użytkownicy korporacyjni zapewne będą wykorzystywali wersję komercyjną wyposażoną w dodatkowe funkcje, ale po opanowaniu wydania darmowego przejście do wersji płatnej powinno być bardzo łatwe.

**Procedura 1-1.** *Instalowanie Visual Studio Community Edition*

1. Upewnij się, że masz co najmniej 2,3 gigabajtów wolnego miejsca na dysku twardym (może być potrzebne aż do 60 gigabajtów, zależnie od wybranych funkcjonalności) i uprawnienia administracyjne.
2. Pobierz ze strony *https://visualstudio.microsoft.com/vs/community/* instalator online dla odpowiedniej platformy.
3. Uruchom instalator. Na zakończenie instalacji konieczne będzie ponowne uruchomienie komputera.

**Procedura 1-2.** *Testowanie instalacji Visual Studio*

1. Uruchom Visual Studio i wybierz opcję **Start without Code** (uruchom bez kodu) w prawym dolnym rogu. Jeśli pojawi się oferta aktualizacji, zainstaluj ją w pierwszej kolejności.
2. Rozwiązanie (*Solution*) w Visual Studio może zawierać jeden lub więcej projektów, a każdy projekt może obejmować jeden lub więcej plików. Aby utworzyć rozwiązanie, wybierz polecenie **File → New Project**. Zaznacz opcję **Empty Project** (pusty projekt), kliknij **Next**, następnie nadaj projektowi nazwę i zanotuj albo zmień lokalizację dla swoich plików, po czym kliknij **Create**.
3. W panelu **Solution Explorer** (eksplorator rozwiązań) po prawej kliknij prawym klawiszem myszy opcję **Source Files**, po czym wybierz polecenie **Add → New Item → C++ file**, po czym nadaj plikowi nową nazwę (zachowując rozszerzenie .cpp).
4. Wpisz poniższy przykład do nowo utworzonego pliku źródłowego, aby przetestować VS 2019.

```
#include <iostream>
using namespace std;
int main()
{
    string word;
    cout << "Type in World " << endl;
    cin >> word;
    cout << "Hello " << word << "!!!"  << endl;
    cout << "Press any key to exit\n";
    cin >> word;
    return 0;
}
```

5. Wybierz polecenie **Build → Build Solution** (zbuduj rozwiązanie). Następnie, jeśli w dolnym panelu wyjściowym nie pojawią się żadne błędy, wybierz **Debug → Start Without Debugging** (uruchom bez debugowania), aby uruchomić program testowy.

W przyszłości będziemy zwykle chcieli eksperymentować z uruchamianiem programów pod kontrolą debugera, ale dla tego prostego testu nie jest to konieczne. Ważną częścią posługiwania się VS jest znajomość tego, gdzie przechowywane są nasze rozwiązania/projekty/pliki źródłowe. Aby przyśpieszyć proces kompilacji, należy ją zawsze wykonywać na lokalnym dysku twardym i jedynie kopiować wynikowe pliki na dyski przenośne lub zdalne; w przeciwnym razie czas budowania może być znacznie dłuższy od normalnego.

Rozdział 2

# Nowoczesny C++

Rozwój języka programowania C++ rozpoczyna się w roku 1979 od powstania języka nazywanego *C with Classes* (C z klasami). Nazwa C++ została nieformalnie przyjęta w roku 1983, a prace nad językiem trwały przez całe lata osiemdziesiąte i dziewięćdziesiąte bez przyjęcia formalnego standardu. Zmieniło się to dopiero w roku 1998, gdy pojawił się pierwszy standard ISO opisujący język programowania C++. Od tego czasu opublikowanych zostało kilka aktualizacji standardu, na przykład w roku 2003, 2011 i 2014; zaś co do najnowszej wersji, czyli C++20, w chwili pisania tych słów (czyli pod koniec roku 2019) jest ona na końcu drogi do finalnej akceptacji.

---

**Uwaga** Standard opublikowany w roku 2003 był niewielką aktualizacją standardu 1998, która nie wprowadzała nowych funkcjonalności. Istotne zmiany zostały wprowadzone w standardzie C++11 i późniejszych. W tej książce jednak zajmować się będziemy głównie nowymi funkcjonalnościami dostępnymi w wersjach C++17 i C++20.

---

W książce tej przede wszystkim zamierzamy skupić się na najnowszym standardzie programowania C++, czyli C++20. Ilekroć mówimy tu o języku programowania C++, można mieć pewność, że mamy na myśli język zgodny z opisem zawartym w bieżącym standardzie ISO. Przy omawianiu funkcji wprowadzonych w roku 2011 będziemy się jawnie odwoływać do wersji C++11; w przypadku dowolnych funkcjonalności, które pojawiły się przed rokiem 2011, będziemy używać desygnatu C++98 i tak dalej.

W tym rozdziale przyjrzymy się funkcjom programistycznym dodanym do języka w najnowszym oficjalnym standardzie i w wersji C++20. Wiele nowoczesnych funkcjonalności C++ zostało dodanych w standardach C++11 i C++17, a później rozszerzonych w proponowanym standardzie C++20. Biorąc to pod uwagę, ważna jest umiejętność

identyfikowania różnic, gdy posługujemy się kompilatorami wspierającymi standard, który nie jest jeszcze zatwierdzony w 100%. W istocie właśnie dlatego VS 2019 i inne wersje są oznaczane jako nie posiadające pełnej obsługi wszystkich proponowanych uzupełnień C++ 20, przynajmniej według stanu z końca roku 2019.

# Przepis 2-1. Inicjowanie zmiennych

## Problem

Chcielibyśmy móc inicjować wszystkie zmienne w standardowy sposób.

## Rozwiązanie

Ujednolicona składnia inicjowania została wprowadzona w standardzie C++11 i możemy jej używać do inicjowania zmiennych dowolnego typu.

## Jak to działa

Aby docenić, dlaczego ujednolicone inicjowanie wprowadzone w C++11 jest ważną funkcjonalnością języka, trzeba poznać niedostatki mechanizmu inicjowania zmiennych występującego w C++98. Listing 2-1 pokazuje program zawierający pojedynczą klasę `MyClass`.

**Listing 2-1.** *Problem vexing parse kodu C++*

```
class MyClass
{
private:
    int m_Member;

public:
    MyClass() = default;
    MyClass(const MyClass& rhs) = default;
};

int main()
{
    MyClass objectA;
```

```
    MyClass objectB(MyClass());
    return 0;
}
```

Kod przedstawiony w listing 2-1 wygeneruje błąd kompilacji. Problem występuje w definicji `objectB`. Kompilator C++ nie rozpozna tego wiersza jako definicji zmiennej `objectB` typu `MyClass`, wywołującej konstruktor, który przyjmuje obiekt skonstruowany przez wywołanie konstruktora `MyClass`. Moglibyśmy oczekiwać, że to właśnie kompilator powinien tu zobaczyć; jednak w rzeczywistości postrzega on deklarację funkcji. Kompilator sądzi, że wiersz ten deklaruje funkcję nazwaną `objectB`, która zwraca obiekt `MyClass` i ma pojedynczy, nienazwany wskaźnik funkcyjny do funkcji, zwracającej obiekt `MyClass` i nie przekazującej żadnych parametrów.

Próba skompilowania programu pokazanego w listingu 2-1 powoduje, że Clang i Visual Studio generują ostrzeżenie. Poniżej pokazany jest komunikat zwracany przez Clang. W Visual Studio będzie on brzmiał podobnie:

```
main.cpp:14:20: warning: parentheses were disambiguated as a function
      declaration [-Wvexing-parse]
    MyClass objectB(MyClass());
            ^~~~~~~~~~
main.cpp:14:21: note: add a pair of parentheses to declare a variable
    MyClass objectB(MyClass());
            ^
            (        )
```

Kompilator Clang poprawnie rozpoznał, że kod wprowadzony w listingu 2-1 zawiera problem *vexing parse*[2] i nawet sugeruje opakowanie konstruktora `MyClass` przekazywanego jako parametr w kolejną parę nawiasów, aby rozwiązać ten problem. Standard C++11 udostępnia alternatywne rozwiązanie w postaci *ujednoliconego inicjowania*. Można je zobaczyć w listingu 2-2.

2 Dosłownie „irytująca analiza”. Termin ukuty przez Scotta Meyersa, opisujący niejednoznaczności związane ze stosowaniem zwykłych nawiasów w odmiennych rolach. Został on oficjalnie wprowadzony do standardu w wersji C++11. Jak dotąd, nie doczekał się polskiego odpowiednika i dlatego używany jest termin angielski (przyp. tłum.).

**Listing 2-2.** *Użycie jednolitego inicjowania w celu rozwiązania problemu vexing parse*

```
class MyClass
{
private:
    int m_Member;

public:
    MyClass() = default;
    MyClass(const MyClass& rhs) = default;
};

int main()
{
    MyClass objectA;
    MyClass objectB{MyClass{}};
    return 0;
}
```

Jak widzimy w listingu 2-2, jednolite inicjowanie zastępuje zwykłe nawiasy klamrowymi. Ta zmiana składni informuje kompilator, że zamierzamy używać inicjowania ujednoliconego do inicjowania naszej zmiennej. Składnia ta może być używana do inicjowania zmiennych niemal wszystkich typów.

---

**Uwaga** W poprzednim akapicie stwierdziliśmy, że inicjowanie ujednolicone może być stosowane do inicjowania *niemal* wszystkich zmiennych. Mogą pojawić się problemy przy inicjowaniu agregatów lub prostych, starych typów danych; jednak na razie nie musimy się nimi martwić.

---

Inną korzyścią ze stosowania ujednoliconego inicjowania jest uchronienie się przed występowaniem *konwersji zawężających*. Kod pokazany w listingu 2-3 nie zdoła się skompilować przy używaniu inicjowania ujednoliconego.

**Listing 2-3.** *Użycie inicjowania ujednoliconego w celu uniknięcia konwersji zawężających*

```
int main()
{
    int number{ 0 };
    char another{ 512 };
```

```
    double bigNumber{ 1.0 };
    float littleNumber{ bigNumber };

    return 0;
}
```

Kompilator Clang, podobnie jak kompilator VS, zgłoszą błędy przy kompilowaniu kodu z tego przykładu, gdyż w kodzie źródłowym występują dwie konwersje zawężające. Pierwsza następuje w chwili, gdy próbujemy zdefiniować zmienną typu `char` o wartości literału 512. Maksymalna wartość, jaką może przechować typ `char`, to 255; tym samym wartość 512 zostałaby obcięta do maksymalnej wartości tego typu danych. Kompilator zgodny z C++11 lub wersją nowszą nie skompiluje tego kodu z powodu tego błędu. Inicjowanie zmiennej typu `float` wartością typu `double` również jest konwersją zawężającą. Konwersje zawężające występują wtedy, gdy dane są przekazywane z jednego typu do innego i typ docelowy nie może pomieścić wszystkich wartości reprezentowanych przez typ źródłowy. W przypadku konwersji typu `double` na `float` następuje utrata dokładności; z tego względu kompilator (poprawnie) nie skompiluje tego kodu, tak jak jest on aktualnie napisany. Kod pokazany w listingu 2-4 używa `static_cast` w celu poinformowania kompilatora, że konwersja zawężająca nie jest błędem lub niedopatrzeniem, ale jest zamierzona – tym razem kod się skompiluje bez błędów.

**Listing 2-4.** *Wykorzystanie static_cast w celu skompilowania konwersji zawężających*

```
int main()
{
    int number{ 0 };
    char another{ static_cast<char>(512) };

    double bigNumber{ 1.0 };
    float littleNumber{ static_cast<float>(bigNumber) };

    return 0;
}
```

# Przepis 2-2. Inicjowanie obiektów przy użyciu list inicjalizacyjnych

## Problem

Chcielibyśmy skonstruować obiekt z wielu obiektów określonego typu.

## Rozwiązanie

Nowoczesny C++ udostępnia listy inicjalizacyjne, których można użyć w celu przekazania do konstruktora wielu obiektów tego samego typu.

## Jak to działa

Listy inicjalizacyjne wprowadzone w standardzie C++11 oparte są na inicjowaniu ujednoliconym i mają na celu ułatwienie inicjowania złożonych typów. Klasycznym przykładem typu złożonego, którego zainicjowanie danymi może być trudne, jest wektor. Listing 2-5 pokazuje dwa różne wywołania standardowego konstruktora wektora.

**Listing 2-5.** *Konstruowanie obiektów wektorów*

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    using MyVector = vector<int>;

    MyVector vectorA( 1 );
    cout << vectorA.size() << " " << vectorA[0] << endl;

    MyVector vectorB( 1, 10 );
    cout << vectorB.size() << " " << vectorB[0] << endl;

    return 0;
}
```

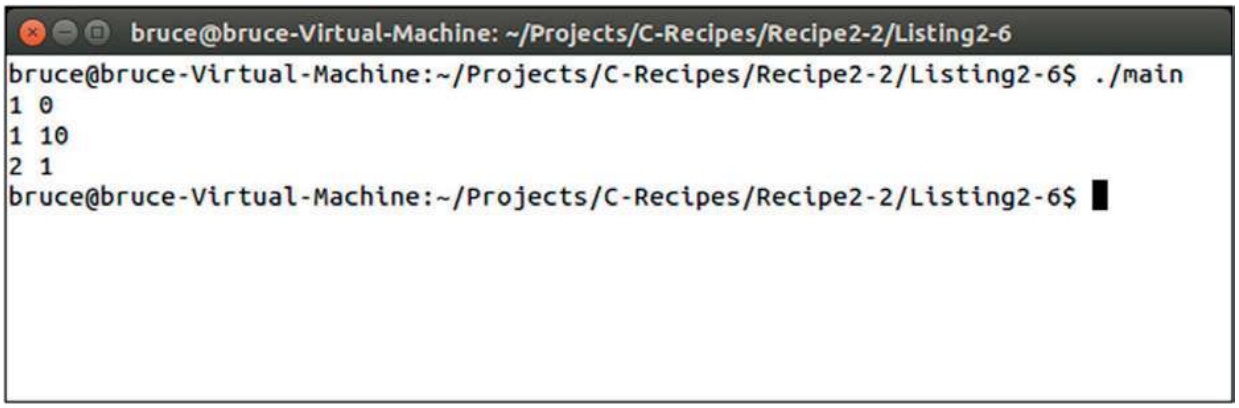
Kod pokazany w listingu 2-5 nie robi jednak tego, czego moglibyśmy oczekiwać na pierwszy rzut oka. Zmienna vectorA zostanie zainicjowana pojedynczą wartością typu int, zawierającą 0. Ktoś mógłby oczekiwać, że będzie zawierać pojedynczą liczbę całkowitą o wartości 1, ale to byłoby nieprawidłowe. Pierwszy parametr konstruktora vector określa, jak wiele wartości będzie mógł przechowywać początkowy obiekt typu vector, a w tym przypadku żądamy przechowania pojedynczej zmiennej. Analogicznie moglibyśmy oczekiwać, że vectorB będzie zawierać dwie wartości, a mianowicie 1 i 10; jednak to, co tu mamy, to obiekt typu vector zawierający jedną wartość i tą wartością jest 10. Zmienna vectorB jest budowana przy użyciu tego samego konstruktora, co vectorA; jednak tym razem specyfikowana jest wartość użyta do zainicjowania elementu wektora, zamiast użycia wartości domyślnej.

Kod przedstawiony w listingu 2-6 używa listy inicjalizacyjnej w połączeniu z inicjowaniem ujednoliconym, aby skonstruować wektor zawierający dwa elementy o wskazanych wartościach.

**Listing 2-6.** *Wykorzystanie inicjowania ujednoliconego do konstruowania zmiennej typu* vector

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    using MyVector = vector<int>;

    MyVector vectorA( 1 );
    cout << vectorA.size() << " " << vectorA[0] << endl;

    MyVector vectorB( 1, 10 );
    cout << vectorB.size() << " " << vectorB[0] << endl;

    MyVector vectorC{ 1, 10 };
    cout << vectorC.size() << " " << vectorC[0] << endl;

    return 0;
}
```

Kod z listingu 2-6 tworzy trzy różne obiekty typu vector. Rysunek 2-1 pokazuje wyjście generowane przez ten program.




**Rysunek 2-1.** *Wyjście generowane przez program z listingu 2-6*

Wyjście konsolowe widoczne na rysunku 2-1 pokazuje rozmiar oraz wartość przechowywaną w pierwszym elemencie każdego wektora. Można zauważyć, że pierwszy wektor zawiera pojedynczy element o wartości 0. Drugi wektor również zawiera jeden element; jednak tym razem wartość tego elementu to 10. Trzeci wektor został skonstruowany przy użyciu inicjowania ujednoliconego i zawiera dwie wartości, przy czym wartość pierwszego elementu to 1. Jak można się domyślić, wartością drugiego elementu będzie 10. Nie zadbanie o właściwy rodzaj inicjowania naszych typów może powodować znaczącą różnicę w zachowaniu naszych programów. Kod pokazany w listingu 2-7 pokazuje jawne użycie typu `initializer_list` do konstruowania obiektu typu `vector`.

**Listing 2-7.** *Jawne użycie* `initializer_list`

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    using MyVector = vector<int>;

    MyVector vectorA( 1 );
    cout << vectorA.size() << " " << vectorA[0] << endl;

    MyVector vectorB( 1, 10 );
    cout << vectorB.size() << " " << vectorB[0] << endl;
```

```
    initializer_list<int> initList{ 1, 10 };
    MyVector vectorC(initList);
    cout << vectorC.size() << " " << vectorC[0] << endl;

    return 0;
}
```

Kod w listingu 2-7 zawiera jawnie zadeklarowany obiekt `initializer_list`, który zostanie użyty do skonstruowania wektora. Kod z listingu 2-6 utworzył taki obiekt niejawnie przy konstruowaniu wektora za pomocą inicjowania ujednoliconego. Zwykle nie ma potrzeby jawnego tworzenia list inicjalizujących, jak w tym przykładzie; trzeba jednak mieć świadomość, co kompilator wykonuje w tle, gdy piszemy kod z użyciem inicjowania ujednoliconego.

# Przepis 2-3. Używanie dedukowania typu

## Problem

Chcemy napisać kod przenośny, który nie będzie wymagał ponoszenia wysokich kosztów konserwacji przy zmianach typów.

## Rozwiązanie

C++ udostępnia słowo kluczowe `auto`, którego można użyć, aby kompilator spróbował automatycznie wydedukować typ zmiennej.

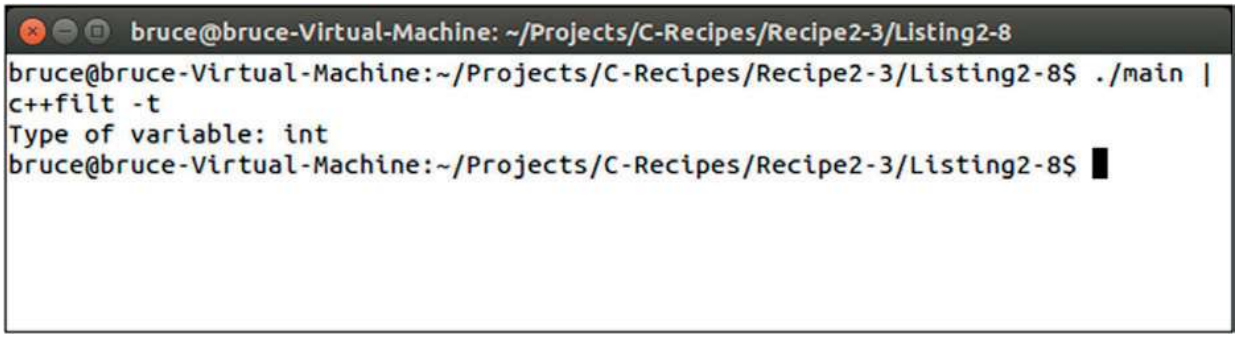
## Jak to działa

Kompilatory C++98 miały możliwość automatycznego dedukowania typu zmiennej. Jednak funkcjonalność ta była dostępna jedynie wówczas, gdy pisaliśmy kod wykorzystujący szablony i pominęliśmy specyfikację typu. W nowoczesnym C++ to dedukowanie typu zostało rozszerzone na znacznie więcej scenariuszy. Kod w listingu 2-8 pokazuje użycie słowa kluczowego `auto` i metody `typeid` w celu rozpoznania typu zmiennej.

**Listing 2-8.** *Używanie słowa kluczowego* auto

```
#include <iostream>
#include <typeinfo>

using namespace std;

int main()
{
    auto variable = 1;
    cout << "Type of variable: " << typeid(variable).name() << endl;

    return 0;
}
```

Kod ten pokazuje tworzenie zmiennej o automatycznie wykrywanym typie w języku C++. Kompilator automatycznie ustali, że w tym kodzie chcieliśmy utworzyć zmienną typu int i będzie to typ, który pojawi się na wyjściu programu, a raczej coś podobnego. W istocie kompilator Clang zwróci swoją wewnętrzną reprezentację typu całkowitoliczbowego, którą jest i. Można przekazać to wyjście do programu o nazwie c++filt, aby przekonwertować ten wynik na normalną nazwę typu. Rysunek 2-2 pokazuje, jak można to osiągnąć.




**Rysunek 2-2.** *Wykorzystanie* c++filt *w celu uzyskania poprawnej nazwy typu z kompilatora Clang*

Program c++filt z powodzeniem przekształcił wewnętrzny typ kompilatora Clang na czytelny dla człowieka format typu int.

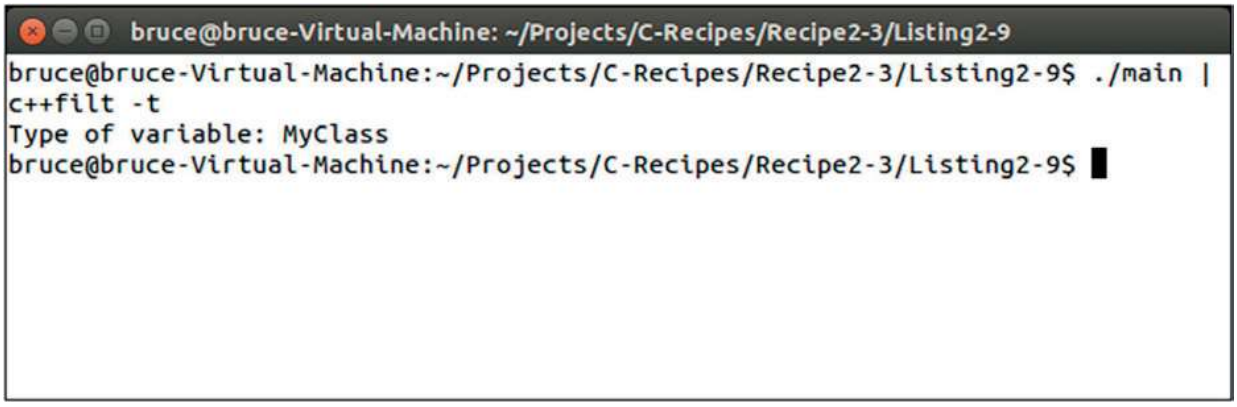
Słowo kluczowe auto działa również w odniesieniu do klas. Pokazuje to listing 2-9.

**Listing 2-9.** *Używanie* auto *w połączeniu z* class

```
#include <iostream>
#include <typeinfo>
```

```
using namespace std;

class MyClass
{
};

int main()
{
    auto variable = MyClass();
    cout << "Type of variable: " << typeid(variable).name() << endl;

    return 0;
}
```

Program ten wypisze jako wyjście nazwę *MyClass*, co można zobaczyć na rysunku 2-3.




**Rysunek 2-3.** *Używanie* auto *dla klas*

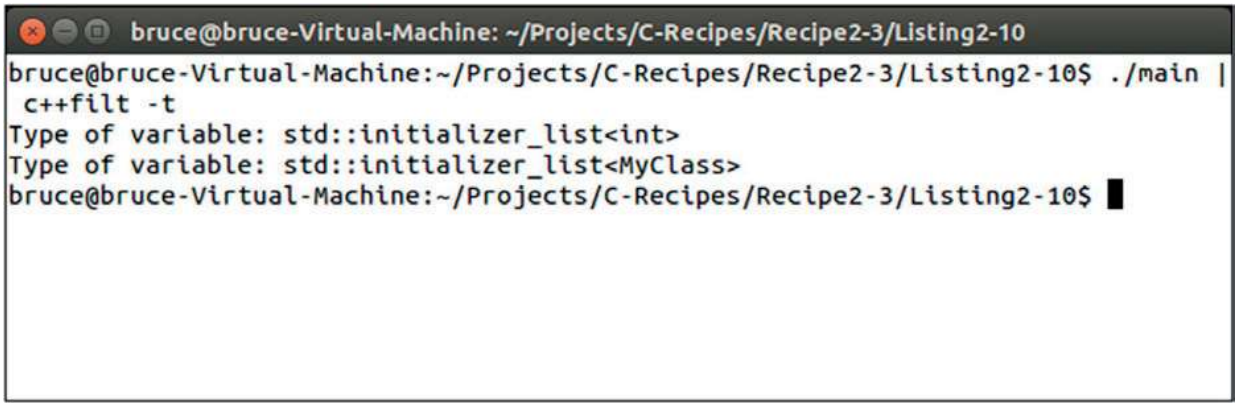
Nieszczęśliwie istnieją sytuacje, w których słowo kluczowe auto może prowadzić do mniej pożądanych rezultatów. Na pewno wypadniemy z gry, jeśli będziemy próbować łączyć słowo kluczowe auto z inicjowaniem ujednoliconym. Listing 2-10 pokazuje przykład takiego użycia.

**Listing 2-10.** *Używanie* auto *w połączeniu z inicjowaniem ujednoliconym*

```
#include <iostream>
#include <typeinfo>

using namespace std;
```

```
class MyClass
{
};

int main()
{
    auto variable{ 1 };
    cout << "Type of variable: " << typeid(variable).name() << endl;

    auto variable2{ MyClass{} };
    cout << "Type of variable: " << typeid(variable2).name() << endl;

    return 0;
}
```

Moglibyśmy oczekiwać, że kod z listingu 2-10 utworzy zmienną typu `int` oraz drugą typu `MyClass`; tak jednak się nie stanie. Rysunek 2-4 pokazuje wyjście generowane przez ten program.




**Rysunek 2-4.** *Wyjście generowane przy stosowaniu* `auto` *z inicjowaniem ujednoliconym*

Szybkie spojrzenie na rysunek 2-4 pokazuje problem, który natychmiast pojawia się, gdy próbujemy używać słowa kluczowego `auto` wraz z inicjowaniem ujednoliconym. Funkcja inicjowania ujednoliconego w C++ automatycznie tworzy zmienną typu `initializer_list`, zawierającą wartość pożądanego przez nas typu, a nie sam typ i wartość. Prowadzi to do względnie prostej rady: nie należy używać inicjowania ujednoliconego, jeśli definiujemy zmienne przy użyciu `auto`. Zalecane jest nie używanie składni `auto`, nawet jeśli pożądanym typem faktycznie jest `initializer_list`, gdyż kod będzie znacznie łatwiejszy do zrozumienia i mniej podatny na błędy, jeśli nie będziemy mieszać ze sobą

i dopasowywać stylów inicjowania zmiennych. Wynika stąd jeszcze jedna, finalna porada: używać auto dla zmiennych lokalnych, gdy tylko jest to możliwe. Niemożliwe jest zadeklarowanie zmiennej o automatycznym typie i jej nie zdefiniowanie; tym samym niemożliwa jest sytuacja, gdy mamy niezdefiniowaną zmienną lokalną, o ile używamy składni auto. To spostrzeżenie może wyeliminować przynajmniej jedno potencjalne źródło błędów w naszych programach.

# Przepis 2-4. Używanie auto w funkcjach

## Problem

Chcemy utworzyć bardziej ogólne funkcje przy użyciu dedukcji typu, aby ułatwić późniejsze utrzymywanie kodu.

## Rozwiązanie

Nowoczesny C++ pozwala nam używać dedukcji typów dla parametrów funkcji i typów zwracanych.

## Jak to działa

C++ pozwala wykorzystać dedukcję typu przy pracy z funkcjami, wykorzystując dwie metody. Możemy dedukować typy parametrów funkcji poprzez utworzenie funkcji szablonowej i wywołanie jej bez jawnej specjalizacji. Typ zwracany może zostać wydedukowany poprzez użycie słowa kluczowego auto w miejsce typu funkcji. Listing 2-11 pokazuje użycie auto do wydedukowania typu zwracanego dla funkcji.

**Listing 2-11.** *Dedukowanie typu zwracanego funkcji przy użyciu* auto

```
#include <iostream>

using namespace std;

auto AutoFunctionFromReturn(int parameter)
{
    return parameter;
}
```

```
int main()
{
    auto value = AutoFunctionFromReturn(1);
    cout << value << endl;

    return 0;
}
```

Typ zwracany przez funkcję `AutoFunctionFromReturn` w listingu 2-11 jest dedukowany automatycznie. Kompilator zbada typ zmiennej zwracanej z tej funkcji i użyje go do wydedukowania typu, jaki ma zostać zwrócony. Działa to poprawnie, gdyż kompilator dysponuje wewnątrz funkcji wszystkimi informacjami, jakich potrzebuje do wydedukowania typu. Zwracana jest zmienna `parameter`; tym samym kompilator może użyć jej typu jako typu zwracanego przez tę funkcję.

Sprawy nieco się komplikują, jeśli potrzebujemy zbudować program przy użyciu kompilatora zgodnego z C++11. Zbudowanie kodu z listingu 2-11 przy użyciu C++11 daje w wyniku następujący błąd:

```
main.cpp:5:1: error: 'auto' return without trailing return type
auto AutoFunctionFromReturn(int parameter)
```

Listing 2-12 zawiera funkcję z automatyczną dedukcją typu zwracanego, która działa w C++11.

**Listing 2-12.** *Dedukowanie typu zwracanego w C++11*

```
#include <iostream>

using namespace std;

auto AutoFunctionFromReturn(int parameter) -> int
{
    return parameter;
}

int main()
{
    auto value = AutoFunctionFromReturn(1);
    cout << value << endl;
```

```
    return 0;
}
```

Patrząc na kod z listingu 2-12 ktoś mógłby się zastanawiać, dlaczego w ogóle mielibyśmy robić coś takiego. Dedukowanie typu zwracanego funkcji ma niewielkie zastosowanie, jeśli zawsze określamy, że będzie to `int` – i to jest prawda. Dedukowanie typu zwracanego jest znacznie bardziej przydatne w przypadku funkcji, które w swojej sygnaturze nie mają zadeklarowanych typów parametrów. Listing 2-13 pokazuje dedukcję w działaniu dla funkcji szablonowej.

**Listing 2-13.** *Dedukowanie typów zwracanych dla funkcji szablonowych C++11*

```
#include <iostream>

using namespace std;

template <typename T>
auto AutoFunctionFromParameter(T parameter) -> decltype(parameter)
{
    return parameter;
}

int main()
{
    auto value = AutoFunctionFromParameter(2);
    cout << value << endl;

    return 0;
}
```

Listing 2-13 pokazuje użyteczne zastosowanie dedukowania typu zwracanego. Tym razem funkcja jest specyfikowana jako szablon (`template`); tym samym kompilator nie może określić typu zwracanego, opierając się na typie parametru. W standardzie C++11 wprowadzono słowo kluczowe `decltype` jako dopełnienie słowa kluczowego `auto`. `decltype` służy do nakazania kompilatorowi użycia typu w danym wyrażeniu. Wyrażeniem tym może być nazwa zmiennej; tym niemniej możemy tu również wstawić funkcję, a `decltype` pozwoli wydedukować typ zwracany z tej funkcji.

W tym miejscu kod zatoczył pełne koło. Standard C++11 zezwala na użycie `auto` dla funkcji w celu dedukowania typu zwracanego, ale wymaga, aby typ był nadal specyfikowany jako typ wartości zwracanej (*trailing return type*). Typ ten może być dedukowany

przy użyciu decltype; jednak prowadzi to do nadmiernie rozwlekłego kodu. Standard C++14 wyprostował tę sytuację, pozwalając na użycie auto w funkcji bez konieczności posiadania typu wartości zwracanej, nawet przy stosowaniu w szablonach, co pokazuje listing 2-14.

**Listing 2-14.** *Wykorzystanie* auto *do dedukowania typu zwracanego funkcji szablonowej*

```
#include <iostream>

using namespace std;

template <typename T>
auto AutoFunctionFromParameter(T parameter)
{
    return parameter;
}

int main()
{
    auto value = AutoFunctionFromParameter(2);
    cout << value << endl;

    return 0;
}
```

# Przepis 2-5. Używanie stałych czasu kompilacji

## Problem

Chcielibyśmy zoptymalizować działanie w czasie wykonywania naszego programu przy użyciu stałych czasu kompilacji.

## Rozwiązanie

C++ udostępnia słowo kluczowe constexpr, którego można użyć w celu zagwarantowania, że wyrażenie będzie mogło zostać obliczone w trakcie kompilacji.

## Jak to działa

Słowo kluczowe constexpr można wykorzystać do tworzenia zmiennych i funkcji, które z pewnością będą mogły zostać obliczone w trakcie kompilacji. Kompilator zgłosi błąd, jeśli dodamy do takich wyrażeń dowolny kod, który uniemożliwi obliczenie w czasie kompilacji. W istocie standard C++20 rozszerza funkcjonalność constexpr o zezwolenie na stosowanie bloków try/catch wewnątrz is_constant_evaluated; jednak obsługa tej funkcjonalności nie jest spójna pomiędzy różnymi kompilatorami, a przynajmniej nie do czasu pełnego zatwierdzenia standardu. Listing 2-15 pokazuje program używający zmiennej wykorzystującej constexpr do zdefiniowania rozmiaru tablicy (array).

**Listing 2-15.** *Używanie* constexpr *do definiowania wielkości* array

```
#include <array>
#include <cstdint>
#include <iostream>

int main()
{
    constexpr uint32_t ARRAY_SIZE{ 5 };
    std::array<uint32_t, ARRAY_SIZE> myArray{ 1, 2, 3, 4, 5 };

    for (auto&& number : myArray)
    {
        std::cout << number << std::endl;
    }

    return 0;
}
```

Zmienna constexpr w listingu 2-15 gwarantuje, że wartość będzie mogła zostać obliczona w czasie kompilacji. Jest to niezbędne w tym przypadku, gdyż rozmiar tablicy jest czymś, co musi zostać ustalone podczas kompilowania programu. Listing 2-16 pokazuje, jak możemy rozszerzyć ten przykład, aby zawierał funkcję typu constexpr.

**Listing 2-16.** *Funkcja typu* constexpr

```
#include <array>
#include <cstdint>
#include <iostream>
```

```
constexpr uint32_t ArraySizeFunction(int parameter)
{
    return parameter;
}

int main()
{
    constexpr uint32_t ARRAY_SIZE{ ArraySizeFunction(5) };
    std::array<uint32_t, ARRAY_SIZE> myArray{ 1, 2, 3, 4, 5 };

    for (auto&& number : myArray)
    {
        std::cout << number << std::endl;
    }

    return 0;
}
```

Możemy pójść jeszcze dalej i utworzyć klasę z konstruktorem constexpr. Pokazuje to listing 2-17.

**Listing 2-17.** *Tworzenie konstruktorów klas* constexpr

```
#include <array>
#include <cstdint>
#include <iostream>

class MyClass
{
private:
    uint32_t m_Member;

public:
    constexpr MyClass(uint32_t parameter)
        : m_Member{parameter}
    {
    }

    constexpr uint32_t GetValue() const
    {
        return m_Member;
```

```
    }
};

int main()
{
    constexpr uint32_t ARRAY_SIZE{ MyClass{ 5 }.GetValue() };
    std::array<uint32_t, ARRAY_SIZE> myArray{ 1, 2, 3, 4, 5 };

    for (auto&& number : myArray)
    {
        std::cout << number << std::endl;
    }

    return 0;
}
```

Kod z listingu 2-17 jest w stanie utworzyć obiekt i wywołać metodę w wyrażeniu constexpr. Jest to możliwe, gdyż konstruktor MyClass został zadeklarowany jako konstruktor constexpr. Kod wykorzystujący słowo kluczowe constexpr pokazany do tego miejsca jest w pełni kompatybilny z kompilatorami C++11. Standard C++17 złagodził wiele ograniczeń, które istniały w wersji C++11, a standard C++20 dodaje nowe funkcjonalności. Instrukcje constexpr zgodne z C++11 nie mogą robić wielu rzeczy, które mógłby wykonywać normalny kod C++. Przykłady takich ograniczeń to tworzenie zmiennych i używanie wyrażeń warunkowych (if). Kod w listingu 2-18 pokazuje funkcję constexpr standardu C++14, której można użyć do ograniczenia maksymalnego rozmiaru tablicy.

**Listing 2-18.** *Używanie funkcji typu* constexpr *w C++14*

```
#include <array>
#include <cstdint>
#include <iostream>

constexpr uint32_t ArraySizeFunction(uint32_t parameter)
{
    uint32_t value{ parameter };
    if (value > 10 )
    {
        value = 10;
    }
```

```
    return value;
}

int main()
{
    constexpr uint32_t ARRAY_SIZE{ ArraySizeFunction(15) };
    std::array<uint32_t, ARRAY_SIZE> myArray{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};

    for (auto&& number : myArray)
    {
        std::cout << number << std::endl;
    }

    return 0;
}
```

Jako końcowy, bardzo prosty przykład, użycie tej funkcjonalności w istocie nie zwraca stałej, ale (oprócz innych rzeczy) zapewnia większą czytelność kodu i przyszłą kompatybilność. Poniższy przykład pokazuje, że stała nie jest w rzeczywistości tym, czym się wydaje.

**Listing 2-19.** *Przykład stałej, która wcale nie jest stałą*

```
#include <iostream>
using namespace std;
int reg_const()
{     return 999; }
constexpr int new_const()
{     return 999; }
int main() {
      const int first = reg_const();
      int second = new_const();
      second = 1;  // z technicznego punktu widzenia nie powinno być
możliwości zmiany stałej!
      cout << first << " != to " << second << endl;
      return 0; }
```