Podstawy języka C

Skrypt akademicki

Paweł Kłeczek

pkleczek@agh.edu.pl
home.agh.edu.pl/~pkleczek

v0.9.1
(2020-05-25)

Spis treści

Rozdział 1

Wstęp

W tym rozdziale poznasz historię i cechy języka C, a także przyjrzysz się istniejącym standardom języka.

1.1 Krótka historia języka C

Język C został stworzony w 1972 roku przez Dennisa Ritchie'go z firmy Bell Labs w trakcie prowadzonych wspólnie z Kenem Thompsonem prac nad systemem operacyjnym UNIX. Ritchie nie wymyślił jednak C zupełnie od podstaw – oparł go o języki ALGOL, BCPL i B. Język ALGOL wprowadził do programowania podejście strukturalne[1], przy czym został zaprojektowany z myślą o środowisku naukowców zajmujących się informatyką. Z kolei język BCPL i jego „potomek" B zostały zaprojektowane z myślą o komercyjnym rozwoju oprogramowania – były to języki programowania systemowego. Ważną rzeczą jest fakt, iż C został pomyślany jako narzędzie pracy programistów aplikacji i systemów operacyjnych (a nie np. jako narzędzie do efektywnego przeprowadzania obliczeń inżynierskich – jak w przypadku języka FORTRAN, albo jako narzędzie dydaktyczne – jak w przypadku języków Pascal oraz BASIC).

Pierwsze próby standaryzacji

Początkowo język C nie był ustandaryzowany, przez co w miarę pojawiania się jego coraz to nowszych wersji tego języka programistom coraz trudniej było utrzymywać programy pisane w języku C (szczególnie, gdy zmiany dotyczyły istniejącej funkcjonalności lub istniejących reguł języka). „Protoplastą" takiego standardu języka C, określanym zwykle jako „K&R C" (od pierwszych liter nazwisk autorów tego języka), było pierwsze wydanie książki „*The C Programming Language*" Briana Kemighana i Dennisa Ritchie'go, które pojawiło się na rynku w 1978 r. (a więc 6 lat po opracowaniu pierwszej wersji języka!). Był to pierwszy szeroko dostępny podręcznik języka C. W szczególności dodatek zatytułowany „*C Reference Manual*" służył jako przewodnik dla twórców różnych kompilatorów języka C, a wiele tych kompilatorów było reklamowanych jako „w pełni zgodne z K&R C". Jednak chociaż wspomniany dodatek definiował reguły języka C, nie określał on biblioteki języka (a język C jest bardzo zależny od swojej biblioteki), stąd wobec braku standardu oficjalnego faktycznym standardem stała się biblioteka dostarczana z implementacją języka C dla systemu UNIX.

ANSI C lub C89

W miarę jak język C ewoluował i wchodził do coraz szerszego użycia, w społeczności jego użytkowników ujawniała się coraz silniejsza potrzeba istnienia bardziej wyczerpującego, aktualnego i ścisłego standardu. Aby sprostać tym oczekiwaniom, w 1983 r. Amerykański Narodowy Instytut Standardów (American National Standards Institute, ANSI) powołał komitet w celu opracowania nowego, komercyjnego standardu języka. Efektem jego wieloletnich prac było formalne przyjęcie w 1989 r. standardu *ANSI X3.159-1989 "Programming Language C"*, określanego potocznie jako „ANSI C" lub „C89", który definiował zarówno język, jak i standardową bibliotekę. Komitet ANSI kierował się kilkoma ogólnymi zasadami, z których być może najbardziej interesująca myśl przewodnia brzmiała: „zachować ducha języka C" . Komitet uznał, że część tego ducha jest wyrażana przez następujące idee:

[1] Paradygmat strukturalny opiera się na podziale kodu źródłowego programu na procedury i hierarchicznie ułożone bloki z wykorzystaniem struktur kontrolnych w postaci instrukcji wyboru i pętli. Rozwijał się w opozycji do programowania wykorzystującego proste instrukcje warunkowe i skoki.

- zaufanie do programisty
- nieutrudnianie programiście pracy
- zwięzłość i prostota języka
- tylko jeden sposób wykonania każdej operacji
- szybkość, nawet kosztem przenośności

Ostatni z punktów oznacza, że dana implementacja powinna definiować daną operację w taki sposób, aby działała ona jak najlepiej na docelowym komputerze, zamiast próbować narzucić abstrakcyjną, jednolitą definicję.

C90

Standard ANSI C został w 1990 r. zaakceptowany przez Międzynarodową Organizację Normalizacyjną (ang. International Organization for Standardization, ISO) jako standard *ISO/IEC 9899:1990*, określany potocznie jako „C90" (co istotne, z technicznego punktu widzenia C89 i C90 to ten sam język). Od tej pory kolejne wersje standardu języka C są opracowywane i uchwalane przez ISO.

C95

W 1995 r. ISO opublikowało standard *ISO/IEC 9899:1990/AMD1:1995*, określany potocznie jako „C95". Standard ten przede wszystkim poprawiał błędy standardu C90[2], ale wprowadził też kilka nowości, m.in.:

- Wsparcie biblioteczne dla tzw. **szerokich znaków** (ang. wide characters), czyli zestawów znaków wymagających do zakodowania więcej niż ośmiu bitów[3] – np. Unicode.
- Standardowe makra-aliasy dla niektórych operatorów (np. `and` jako alias dla `&&`).
- Standardowa stała symboliczna `__STDC_VERSION__` określająca standard języka użyty przy kompilacji programu.

C99

W 2000 r. ISO opublikowało standard *ISO/IEC 9899:1999*, określany potocznie jako „C99". Standard ten wprowadził m.in.:

- Nowe wbudowane typy danych: `long long`, `_Bool`, `_Complex` i `_Imaginary`.
- Nowe funkcjonalności jądra języka, takie jak literały złożone czy czytelniejszy sposób inicjalizacji składowych struktur i unii.
- Nowe standardowe pliki nagłówkowe, umożliwiające m.in. pisanie bardziej przenośnych programów (np. dzięki możliwości stosowania typów o stałej szerokości) oraz czytelniejszych (np. dzięki stosowaniu aliasów dla typu logicznego i jego wartości)[4].
- Poprawa kompatybilności z językiem C++ m.in. poprzez wprowadzenie funkcji wplatanych, jednolinijkowych komentarzy z użyciem `//`, oraz możliwości przeplatania deklaracji z innymi instrukcjami w obrębie bloku.
- Usunięcie pewnych niebezpiecznych cech języka C90, takich jak niejawne deklaracje funkcji oraz domyślne niejawne stosowanie typu `int`.

C11

W 2011 r. ISO opublikowało standard *ISO/IEC 9899:2011*, określany potocznie jako „C11". Standard ten wprowadził m.in.:

- Wsparcie dla programowania generycznego[5].
- Lepsze wsparcie dla systemu kodowania znaków Unicode.
- Wsparcie dla wielowątkowości i operacji atomicznych – zarówno w bibliotece standardowej, jak i w jądrze języka[6].

---

[2] Jak to ujął komitet standaryzacyjny, jednym z celów jego działania jest „kodyfikacja istniejących rozwiązań praktycznych w celu usunięcia ewidentnych niedostatków"

[3] zob. pliki `wchar.h` i `wctype.h` oraz wielobajtowe funkcje wejścia-wyjścia

[4] zob. pliki `stdint.h`, `stdbool.h`

[5] zob. słowo kluczowe `_Generic`

[6] zob. pliki `threads.h` i `stdatomic.h` oraz specyfikator `_Thread_local`

- Zastąpienie niebezpiecznej funkcji `gets()` jej bezpiecznym odpowiednikiem `gets_s()`.

C18

W 2018 r. ISO opublikowało standard *ISO/IEC 9899:2018*, określany potocznie jako „C18". Standard ten nie wprowadził nowych elementów języka, a jedynie poprawiał wybrane problematyczne fragmenty standardu C11[7]

Uwagi odnośnie terminologii w kontekście standardów języka C

Możesz spotkać się również m.in. z następującymi nieformalnymi terminami:

- „CXX strict" (np. „C99 strict") oznacza użycie takich opcji kompilacji, aby wymusić ścisłe stosowanie danego standardu języka C (w szczególności: bez żadnych niestandardowych rozszerzeń).
- „GNU C" oznacza jedną z dwóch rzeczy – albo kompilator języka C wchodzący w skład pakietu GNU Compiler Collection (GCC), albo użycie niestandardowych ustawień stosowanych domyślnie przez kompilator języka C wchodzący w skład GCC[8].

Ważne

Używając zwrotu „standard języka C" (bez podawania z którego roku) należy mieć na myśli najnowszy standard przyjęty przez ISO.

Standardy języka C a aktualność materiałów dydaktycznych

Obecnie, w dobie takich języków jak Python, Java i C++, popularność języka C zmalała. Oznacza to między innymi, że na rynku wydawniczym pojawia się mniej pozycji poświęconych *współczesnemu* językowi C. W praktyce godne uwagi są dwie pozycje, zaktualizowane do standardu C11 (zwróć uwagę na numer wydania!):

- „*Język C. Kompendium wiedzy. Wydanie IV*" (Stephen G. Kochan)
- „*Język C. Szkoła programowania. Wydanie VI*" (Stephen Prata)

Natomiast książkę „*Język ANSI C. Programowanie*" autorstwa Briana Kernighana i Dennisa Ritchie'go (niezależnie od wydania) można uznać za pozycję przestarzałą – od momentu opublikowania standardu „ANSI C" (w 1989 r.) zostało opublikowanych kilka kolejnych wersji standardu, wprowadzających do jądra języka C oraz do biblioteki standardowej istotne nowe funkcjonalności lub zmieniających pewne funkcjonalności względem ANSI C.

Ważne

Wystrzegaj się nauki z nieaktualnych materiałów (zarówno podręczników, jak i stron internetowych i odpowiedzi udzielanych w serwisie *Stack Overflow*) – uważnie sprawdzaj datę ich opublikowania, aby mieć pojęcie, jaki standard języka obwiązywał (lub raczej: był najbardziej popularny) w momencie publikowania tych materiałów.

1.2 Cechy języka C

Oto wybrane istotne cechy języka C (i napisanych w nim programów):

- Język C to *język ogólnego przeznaczenia* – nadaje się m.in. do pisania aplikacji systemowych, systemów operacyjnych, kompilatorów, programowania systemów wbudowanych[9]. . .
- Język C jest **przenośny** (ang. portable), czyli niezależny od platformy sprzętowej oraz systemu operacyjnego – programista ma możliwość napisania kodu programu w taki sposób, aby dało się go użyć na różnych konfiguracjach sprzętowych i systemowych bez konieczności wprowadzania dalszych zmian w kodzie (lub po minimalnych modyfikacjach).

[7] zob. Clarification Request Summary for C11

[8] Przykładowo, całe jądro systemu Linux zostało napisane z użyciem niestandardowych rozszerzeń GNU C.

[9] W języku C zostały napisane m.in. systemy operacyjne UNIX i Windows oraz interpreter języka Python.

- Język C to tzw. **język programowania wysokiego poziomu** (ang. high-level programming language) – czyli jednej instrukcji w kodzie odpowiada wiele instrukcji dla procesora[10].
- Język C z jednej strony umożliwia operowanie blisko warstwy sprzętowej (m.in. na poszczególnych bitach pamięci operacyjnej), a precyzja jego instrukcji jest zbliżona do precyzji asemblerów (w szczególności często spotykanym rozszerzeniem języka C jest możliwość umieszczania w kodzie tzw. wstawek asemblerowych za pomocą instrukcji `asm`[11]), z drugiej strony posiada elastyczne struktury sterowania przebiegiem wykonania programu (np. pętle i instrukcje warunkowe) umożliwiające wygodne rozwijanie złożonych programów.
- Język C umożliwia programowanie strukturalne – jest oparty o moduły (tzw. funkcje), stanowiące podstawowe „klocki" z których budowany jest program; użytkownik myśli o rozwiązaniu problemu poprzez rozbicie go na (mniejsze) podproblemy.
- Język C jest językiem kompilowanym (wymaga kompilatora, dedykowanego dla danego systemu operacyjnego).
- Język C jest językiem **statycznie typowanym** (ang. statically typed) – zgodność typów danych występujących w kodzie programu jest sprawdzana na etapie kompilacji, a nie na etapie wykonywania programu. W przypadku niezgodności typów kompilator zgłasza błąd, co uniemożliwia wygenerowanie pliku wykonywalnego programu; błąd zostanie zgłoszony nawet wówczas, gdy w praktyce problematyczna instrukcja nigdy nie zostanie wykonana.
- Język C posiada oficjalny standard (przyjęty przez ISO).
- Programy napisane w języku C mają niewielki rozmiar oraz cechują się dużą wydajnością.
- Filozofia języka C polega na nienakładaniu na programistę niepotrzebnych ograniczeń, ale równocześnie na obarczeniu go odpowiedzialnością za właściwe wykorzystanie tej swobody.
- Co do zasady język C zachowuje **kompatybilność wsteczną** (ang. backward compatibility) – oznacza to, że programy napisane z użyciem starszych standardów języka C mogą być poprawnie budowane z użyciem kompilatorów zgodnych z nowszymi standardami języka C, a ich działanie będzie identyczne.

1.3 Zastosowania języka C

Szczyt popularności języka C przypada na lata 80. XX w., kiedy to był on dominującym językiem programowania w świecie minikomputerowych systemów uniksowych. Obecnie, w dobie takich innych języków programowania wysokiego poziomu jak Java, Python czy C++, udział języka C w „rynku" programistycznym maleje – choć wciąż jest on często stosowany m.in. do programowania systemów wbudowanych i do programowania systemowego oraz. . . do implementacji kompilatorów i interpreterów innych języków programowania (np. interpretera języka Python). W szczególności język C++ stanowi niemal dokładny nadzbiór języka C, rozszerzając go o mechanizm obiektowości ułatwiający pracę nad złożonymi systemami – nauka języka C stanowi zatem dobry wstęp do nauki języka C++.

1.4 Czym jest pseudokod?

Pewne zagadnienia związane z językami programowania wygodniej jest omawiać z użyciem **pseudokodu** (ang. pseudocode), czyli takiego zapisu, który zachowuje strukturę charakterystyczną dla kodu zapisanego w języku programowania, lecz jednocześnie rezygnuje ze ścisłych reguł składniowych na rzecz prostoty i czytelności. Pseudokod nie zawiera szczegółów implementacyjnych (jak np. inicjalizacja zmiennych, alokacja pamięci itp.), często też opuszcza się w nim opis działania podprocedur (jeśli powinien być on oczywisty dla czytelnika), zaś nietrywialne kroki algorytmu opisywane są z pomocą formuł matematycznych lub zdań w języku naturalnym. Innymi słowy, pseudokod nie jest związany z żadnym językiem programowania, a więc ten sam pseudokod można przetłumaczyć na różne języki.

Nie istnieją w chwili obecnej szerzej przyjęte standardy zapisu pseudokodu. Większość autorów używa przyjętej ad hoc składni, często opierając się na składni istniejących języków programowania (np. Pascal, ALGOL, C).

[10] W przeciwieństwie do **języków programowania niskiego poziomu** (ang. low-level programming languages), takich jak asemblery, w których każdej instrukcji w kodzie odpowiada dokładnie jedna instrukcja dla procesora. Innymi przykładami języków wysokiego poziomu są m.in. C++, Java, Ada i Fortran.

[11] zob. dodatek J w standardzie C11

Rozdział 2

Wprowadzenie do języka C

W tym rozdziale zapoznasz się ze szczegółami procesu zamiany kodu w języku C na kod zrozumiały dla komputera. Nauczysz się też znajdować i usuwać błędy w kodzie programu.

2.1 Etapy tworzenia programu: Sześć kroków

Oto sześć kroków, na jakie można podzielić proces pisania programu w języku C. Wybrane z nich zostaną omówione dokładniej w kolejnych podrozdziałach.

2.1.1 Określenie celów programu

Rozpoczynając pisanie programu powinieneś mieć jasny pogląd na cele, jakie ma on realizować. Myśl w kategoriach: (1) danych, jakich Twój program będzie potrzebował, (2) zadań obliczeniowych i innych operacji, które będzie musiał wykonać, oraz (3) informacji, jakie powinien przedstawić użytkownikowi. Na tym poziomie planowania powinieneś myśleć w ogólnych kategoriach, a nie w kategoriach określonego języka programowania.

2.1.2 Projektowanie programu

Po naszkicowaniu myślowego obrazu zadań stojących przed programem zadecyduj o sposobie ich realizacji, m.in.:

- Jak powinien wyglądać interfejs użytkownika?
- Jaki powinien być sposób organizacji programu?
- Kto będzie użytkownikiem programu?
- Ile czasu pozostało na jego ukończenie?

Musisz również określić sposób reprezentacji danych w programie (i ewentualnie w plikach zewnętrznych), a także wybrać metody, które zostaną użyte do ich przetwarzania. Wybór dobrego sposobu reprezentacji danych potrafi znacznie ułatwić ich przetwarzanie, przyspiesza także sam proces projektowania programu. Również w tym kroku myśl ogólnikowo, bez skupiania się na konkretnym języku programowania – uwzględnij jednak możliwości poszczególnych języków programowania i wybierz ten, który najlepiej nadaje się do implementacji projektu (np. język C daje możliwość manipulowania bitami, a język Python – nie).

2.1.3 Pisanie kodu programu

Mając opracowany jasny projekt programu możesz rozpocząć jego implementację przez pisanie kodu źródłowego. Kod źródłowy umieszcza się w plikach tekstowych przechowujących zawartość w „surowej” postaci[1] (zob. listing 2.1). Na chwilę obecną będziesz własnoręcznie tworzyć wyłącznie tzw. **pliki źródłowe** (o zwyczajowym rozszerzeniu `.c`), natomiast w dalszych rozdziałach poznasz również pliki nagłówkowe (o zwyczajowym rozszerzeniu `.h`).

Listing 2.1. Przykład kodu źródłowego w języku C

```
/* main.c */
#include <stdlib.h>
```

[1] W języku angielskim taką postać nazywa się plain text.

```c
#include <stdio.h>

int main(void) {
    printf("Hello world!");

    return EXIT_SUCCESS;
}
```

W ramach tego kroku powinieneś także dokumentować swoją pracę. Najprostszym sposobem jest wykorzystanie komentarzy, czyli objaśnień uzupełniających kod źródłowy.

Przydatnym narzędziem wspomagającym efektywne pisanie kodu jest *zintegrowane środowisko programistyczne* (ang. integrated development environment, IDE), czyli program komputerowy udostępniający szereg kluczowych dla programisty funkcjonalności, takich jak edytor kodu z kolorowaniem składni i zaznaczaniem błędów, automatyzacja procesu budowania programu, czy debugowanie. Przykładem zintegrowanego środowiska programistycznego dla języka C jest CLion IDE.

2.1.4 Budowanie programu

Jedynym kodem zrozumiałym dla komputera jest tzw. **kod maszynowy** (ang. machine code), czyli zapisany z użyciem zer i jedynek (w postaci binarnej) ciąg instrukcji dla procesora – przykładowo, w przypadku procesora z rodziny x86 instrukcja `10000011110000000000001010` oznacza „dodaj 10 do wartości przechowywanej w rejestrze EAX” (jej postać szesnastkowa to `83C00A` – 83 to kod operacji dodawania, C0 to kod rejestru EAX, a 0A to dodawana wartość). Taka postać jest skrajnie nieczytelna dla człowieka, a co gorsze zestaw dozwolonych instrukcji jest właściwy dla danego procesora, przez co kod maszynowy jest nieprzenośny.

Pewnym ułatwieniem jest pisanie programu z użyciem **języka asemblera** (ang. assembly language) *właściwego dla danej architektury procesora* – w języku tym polecenia kodu maszynowego mają nadane zrozumiałe dla człowieka etykiety (np. w języku asemblera dla procesorów x86 o składni Intela wspomniana wcześniej instrukcja maszynowa przyjmuje postać `add eax, 10`), natomiast wciąż jedno polecenie w kodzie asemblera odpowiada ściśle jednej instrukcji w kodzie maszynowym. Programy napisane w innym języku niż kod maszynowy muszą być przetłumaczone na kod maszynowy – proces ten nosi nazwę **translacji kodu** (ang. code translation). W przypadku kodu napisanego w języku asemblera translacją zajmuje się ... program nazywany asemblerem. Programowanie w języku asemblera jest czasochłonne (gdyż nie udostępnia on złożonych konstrukcji dostępnych, np. pętli) oraz uciążliwe (np. konieczność pamiętania mnemoników), a napisany kod wciąż nie jest przenośny (gdyż korzysta z zestawu instrukcji właściwych dla danego procesora).

Języki wysokiego poziomu, takie jak język C, pozwalają iść krok dalej w poprawie wygody pisania programów i ich czytelności – jedna instrukcja w języku wysokiego poziomu odpowiada wielu instrukcjom procesora (np. istnieje możliwość konstruowania złożonych wyrażeń matematycznych), przy czym zestaw instrukcji jest niezależny od konfiguracji sprzętowej i systemu operacyjnego. Oznacza to jednak, że kod takiego języka musi zostać odpowiednio przetworzony na kod maszynowy – z użyciem kompilatora lub interpretera:

- **Kompilator** (ang. compiler) dokonuje translacji tzw. **kodu źródłowego** (ang. source code) na **kod obiektowy** (ang. object code) zawierający polecenia w kodzie maszynowym zrozumiałym dla zadanego procesora (lub danej maszyny wirtualnej[2]), przy czym *cały* kod źródłowy jest od razu tłumaczony na kod obiektowy. W szczególności kompilator sprawdza, czy program jest napisany zgodnie z regułami języka programowania i w przypadku znalezienia błędów daje stosowne komunikaty (a plik obiektowy nie zostaje utworzony).
- **Interpreter** (ang. interpreter) również dokonuje translacji kodu źródłowego na kod obiektowy, tyle że proces ten zachodzi *instrukcja po instrukcji* w miarę wykonywania programu bezpośrednio z kodu źródłowego (zatem może się zdarzyć, że tylko część kodu zostanie przetłumaczona).

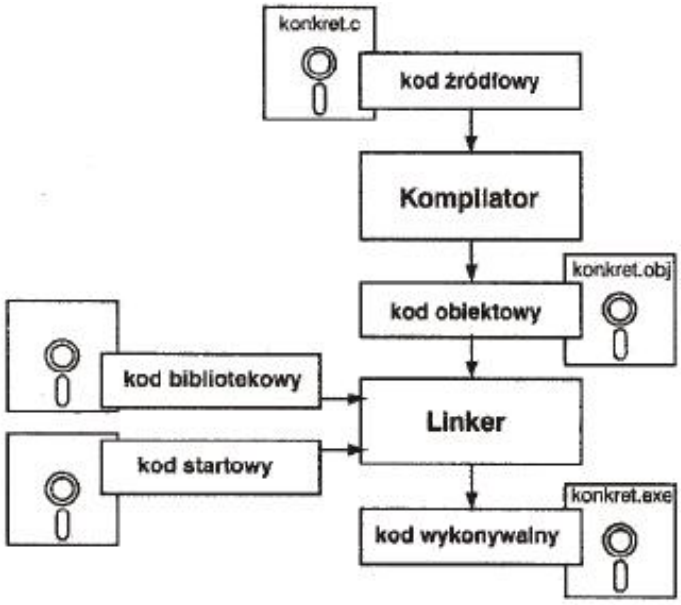
Język C korzysta z kompilatora, natomiast językiem interpretowanym jest np. Python. Choć sam proces kompilacji może być czasochłonny, program skompilowany działa szybciej niż analogiczny program napisany w języku interpretowanym (gdyż kompilator zna całość programu i jest w stanie dokonać pewnych optymalizacji w celu efektywniejszego wykorzystania zasobów sprzętowych).

[2] Przykładem kompilatora tłumaczącego na kod maszyny wirtualnej jest kompilator języka Java.

W przypadku języka C, po napisaniu przez programistę kodu źródłowego programu (w postaci plików tekstowych) wykonywane są kolejno cztery procesy – nazwane łącznie **budowaniem programu** (ang. program build) – prowadzące do otrzymania pliku możliwego do uruchomienia jako program na danym systemie operacyjnym (co istotne, każdy proces wymaga użycia osobnego programu):

1. **Preprocessing.** Preprocesor dokonuje zmian w *tekście* plików źródłowych programu zgodnie z umieszczonymi w nim dyrektywami preprocesora (m.in. dołącza pliki nagłówkowe) – każdy taki plik źródłowy po przetworzeniu go przez preprocesor nazywa się **jednostką translacji** (ang. translation unit).
2. **Kompilacja (do języka assemblera).** Kompilator dokonuje translacji jednostek translacji na język assemblera.
3. **Kompilacja (do kodu maszynowego).** Assembeler dokonuje translacji plików zawierających kod assemblera jednostek translacji na **pliki obiektowe** (ang. object files)[3]. Plik obiektowy jest zrozumiały przez maszynę, ale jednocześnie nie da się go bezpośrednio uruchomić w systemie operacyjnym, gdyż w pliku tym brakuje:
   - kodów obiektowych funkcji z biblioteki standardowej oraz
   - tzw. **kodu startowego** (ang. startup code) odpowiadającego za przygotowanie „środowiska pracy" programu na danym systemie operacyjnym.
4. **Konsolidacja (linkowanie).** Konsolidator, lub inaczej linker, dokonuje **konsolidacji** (ang. linking), czyli dołącza kody obiektowe funkcji z biblioteki standardowej i ewentualnych bibliotek stworzonych przez użytkownika oraz dodaje odpowiedni kod startowy; w efekcie powstaje gotowy do wykonania plik zawierający **kod wykonywalny** (ang. executable code) – tzw. **plik wykonywalny** (ang. executable file)[4].

Dla końcowego użytkownika etapy kompilacji do kodu assemblera i następnie do kodu maszynowego są zwykle widoczne jako jeden etap (choć np. GCC umożliwia podgląd kodu asemblera). W niektórych systemach kompilator i konsolidator uruchamiane są osobno, w innych konsolidator jest wywoływany automatycznie przez kompilator (w takim przypadku do użytkownika należy jedynie wydanie polecenia kompilacji). Schemat procesu budowania został przedstawiony na rysunku 2.1.



Rysunek 2.1. Uproszczony schemat procesu budowania programu w języku C

Język C wykorzystuje to wieloetapowe podejście po to, aby ułatwić stosowanie programowania modularnego. Możesz więc oddzielnie skompilować poszczególne moduły, a później połączyć je przy pomocy konsolidatora. Dzięki temu jeśli zechcesz zmodyfikować tylko jeden z modułów, nie będziesz musiał ponownie kompilować wszystkich pozostałych.

[3] W przypadku systemu operacyjnego Windows jest to plik o rozszerzeniu `.obj`.
[4] W przypadku systemu operacyjnego Windows jest to plik o rozszerzeniu `.exe`.

Ważne

Pliki binarne i wykonywalne są przeznaczone dla danego systemu operacyjnego i dla danej architektury komputera. Aby stworzyć program dla innego systemu trzeba go ponownie zbudować, z innymi parametrami.

Kompilatory języka C nie stanowią części systemu operacyjnego komputera – są dodatkowymi programami, które muszą być zainstalowane, zanim możliwe będzie tworzenie programów w języku C.

Ważne

W niniejszym skrypcie autor zakłada użycie kompilatora języka C wchodzącego w skład pakietu GCC (dostępnego m.in. na systemy operacyjne z rodziny UNIX) lub jego klona MinGW (przeznaczonego dla systemu operacyjnego Windows), w angielskiej wersji językowej.

Użycie innego kompilatora (lub w innej wersji językowej) spowoduje m.in. wyświetlanie innych komunikatów podczas procesu budowania programu oraz różnice w dostępnych flagach kompilacji – dlatego nie jest zalecane w przypadku pracy z niniejszym skryptem.

Ponieważ proces budowania może być bardzo złożony (program może się składać z wielu plików źródłowych, może istnieć konieczność dołączania niestandardowych bibliotek itd.) do zarządzania tym procesem i do jego automatyzacji opracowano specjalne narzędzia, takie jak program *make* w systemach z rodziny UNIX. Program *make* wykorzystuje specjalne pliki konfiguracyjny, tzw. **makefile**, który dla przykładowego programu (składającego się z pojedynczego pliku `myprog.c`) może mieć postać taką, jak w listingu 2.2[5].

Listing 2.2. Przykład pliku konfiguracyjnego (`Makefile`) dla narzędzia *make*

```
# Kompilator: gcc
CC = gcc

# Flagi kompilacji:
#  -std=c11 wymusza kompilację w zgodzie ze standardem C11
#  -Wall włącza większość ostrzeżeń kompilatora
CFLAGS  = -std=c11 -Wall

# Nazwa wynikowego pliku wykonywalnego:
TARGET = myprog

# Reguła powodująca zbudowanie "wszystkiego"
all: $(TARGET)

# Reguła powodująca zbudowanie konkretnego celu.
$(TARGET): $(TARGET).c
  $(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c

# Reguła usuwająca pliki wygenerowane podczas budowania programu.
clean:
  $(RM) $(TARGET)
```

W praktyce wygodniej jest stosować narzędzia generujące pliki konfiguracji budowania dla danego systemu automatycznie, na podstawie własnych plików konfiguracyjnych zawierających bardziej ogólne (i przyjaźniejsze użytkownikowi) polecenia. Przykładem takiego generatora jest narzędzie *CMake* (dostępne na różnych platformach), które umożliwia m.in. generowanie plików makefile. Listing 2.3 przedstawia przykład prostego skryptu konfiguracyjnego dla programu *CMake*. Więcej o narzędziu *CMake* przeczytasz tu.

Listing 2.3. Przykład pliku konfiguracyjnego (`CMakeLists.txt`) dla narzędzia *CMake*

```
cmake_minimum_required(VERSION 3.13)
project(c_playground C)

set(CMAKE_C_STANDARD 11)
```

[5] Więcej o narzędziu *make* i makefile'ach przeczytasz tu.

```
add_compile_options(-Wall)

add_executable(myprog myprog.c)
```

2.1.5 Testowanie programu i usuwanie błędów

Plik wykonywalny (będący wynikiem całego procesu budowania programu) jest programem, który można uruchomić. To, że Twój program udało się uruchomić, jest dobrym znakiem, ale w dalszym ciągu możliwe jest, że będzie on działał w niewłaściwy sposób. Dlatego powinieneś sprawdzić, czy spełnia on wytyczone przez Ciebie założenia. Niektóre Twoje programy będą miały błędy (ang. pot. bugs – „pluskwy", jak określa się je w komputerowym żargonie). Okazji do popełnienia błędu jest wiele, przykładowo:

- Można niewłaściwie opracować projekt.
- Można błędnie zaimplementować dobre pomysły.
- Można przeoczyć nieoczekiwane dane wejściowe, które mogą zakłócić pracę programu.
- Można nieprawidłowo użyć języka C.
- ...

Aby zminimalizować ryzyko popełnienia takich błędów do dobrego zwyczaju należy pisanie tzw. **testów jednostkowych** (ang. unit tests), czyli testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu (np. procedur w programowaniu proceduralnym). Testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik (np. zwrócone wartości, stan obiektu) z oczekiwanymi wynikami. Testy jednostkowe zwykle można wykonywać automatycznie, co pozwala efektywnie testować w drobiazgowy sposób nawet złożone programy. Framework umożliwiający pisanie i uruchamianie testów jednostkowych nie stanowi części języka C – istnieje wiele konkurencyjnych rozwiązań, przy czym jednym z najpopularniejszych jest Google Testing Framework.

Więcej o sposobach usuwania błędów przeczytasz w rozdziale *2.2 Usuwanie błędów*.

2.1.6 „Pielęgnowanie" i modyfikacja programu

Niewykluczone, że napisany przez Ciebie program będzie intensywnie używany przez Ciebie lub kogoś innego. W takim wypadku przypuszczalnie będziesz mieć szereg powodów, aby z czasem wprowadzać w nim zmiany, przykładowo:

- Być może zawiera on niewielki błąd, który ujawnia się po otrzymaniu pewnych szczególnych danych.
- Być może wymyślisz lepszą metodę wykonania jednej z czynności w programie.
- Być może zechcesz rozszerzyć go o jakąś przydatną funkcję lub przystosować do użytku na innym komputerze.

Wszystkie te czynności wykonasz łatwiej, jeśli kod programu pisany był sposób przejrzysty, został odpowiednio udokumentowany oraz zawiera wystarczająco bogaty zestaw testów jednostkowych:

- Wracając po pewnym czasie do przejrzystego i udokumentowanego kodu nie musisz poświęcać dużo czasu na przypomnienie sobie „który fragment kodu realizował daną funkcjonalność" oraz ponowne zrozumienie „o co mi w tym fragmencie kodu właściwie chodziło".
- Po wprowadzeniu zmian w kodzie istniejącego programu wystarczy, że uruchomisz komplet (być może odpowiednio zmodyfikowanych) testów jednostkowych i sprawnie otrzymasz odpowiedź, czy przypadkiem nie został wprowadzony przez Ciebie do programu nowy błąd.

O takich dobrych praktykach programistycznych, które pomogą Ci zwiększyć czytelność kodu oraz zminimalizować ryzyko błędów, poczytasz tu – w szczególności zapoznaj się z częściami poświęconymi nazewnictwu, komentarzom oraz zasadzie DRY (pozostałe zasady wymagają najpierw poznania przez Ciebie dalszych zagadnień języka C). Pamiętaj również, aby odpowiednio formatować kod – przydatne zasady zostały zebrane tu.

2.1.7 Komentarz

Programowanie nie zawsze jest tak liniowym procesem, jak opisano to powyżej. Czasami konieczne jest przeskakiwanie tam i z powrotem między krokami. Na przykład pisząc kod źródłowy możesz zorientować się, że przyjęty wcześniej plan jest mało praktyczny lub możesz zauważyć lepszy sposób wykonania

jakiegoś zadania. Również po uruchomieniu programu i ujrzeniu go w działaniu możesz zdecydować się na modyfikację projektu wyjściowego. W przechodzeniu między etapami niezwykle pomocna jest jasna i kompletna dokumentacja oraz testy jednostkowe.

Większość początkujących programistów pomija kroki *Określenie celów programu* i *Projektowanie programu*, przechodząc od razu do kroku *Pisanie kodu programu*. Istotnie, w przypadku bardzo prostych programów projekt można opracować w pamięci, a ewentualne błędy są łatwe do znalezienia. W miarę jednak wzrostu złożoności programów wyobraźnia zaczyna zawodzić, a tropienie błędów staje się coraz trudniejsze. Ostatecznie tych, którzy zaniedbali etap planowania, czekają godziny straconego czasu, zamieszania i frustracji wywołanej przez źle działające i zawiłe programy. Wyrób w sobie nawyk planowania (choć zgrubnego) przed rozpoczęciem pisania programu; korzystaj ze staroświeckiej, lecz sprawdzonej technologii – papieru i ołówka – aby zapisywać cele Twoich programów i szkicować zarysy ich struktury.

2.2 Usuwanie błędów

Aby być w stanie usunąć dany błąd w programie najpierw trzeba zorientować się, z jakim typem błędu ma się do czynienia, a następnie umiejętnie użyć dostępnych narzędzi w celu zdiagnozowania problematycznego fragmentu kodu.

2.2.1 Błędy a ostrzeżenia

Błędy i ostrzeżenia mogą pojawić się na każdym z trzech etapów budowania programu (preprocessingu, kompilacji i konsolidacji):

- Wystąpienie **błędu** (ang. error) sprawia, że plik wykonywalny nie zostanie utworzony.
- Wystąpienie **ostrzeżenia** (ang. warning) sygnalizuje jedynie możliwe problemy – np. użyliśmy konstrukcji nie należącej do standardu języka i dlatego część (innych) kompilatorów może nie potrafić skompilować naszego kodu – natomiast jego wystąpienie nie uniemożliwia utworzenia pliku wykonywalnego.

Wszystkie błędy zgłoszone przez kompilator i konsolidator *muszą* być poprawione, natomiast *do dobrej praktyki* należy również poprawa wszystkich ostrzeżeń.

2.2.2 Błędy składniowe a semantyczne oraz błędy kompilacji a konsolidacji

Próba zbudowania programu z listingu 2.4 z flagami kompilacji[6] `-Wall` i `-Wextra` pokaże Ci, na czym polega różnica między błędami a ostrzeżeniami, między błędami składniowymi a semantycznymi oraz między błędami kompilacji a konsolidacji.

Listing 2.4. Listing demonstrujący ostrzeżenia i błędy

```
1  #include <stdlib.h>
2
3  void foo();
4
5  int main(void) {
6      double n = 3
7      int n3 = n * n;  // oblicz "n do potęgi 3"
8
9      foo();
10
11     return EXIT_SUCCESS;
12 }
```

Na etapie *kompilacji* zgłoszone zostały następujące problemy (o tym, że błędy dotyczą etapu kompilacji (a nie konsolidacji) świadczy rozszerzenie pliku, w którym znaleziono błędy – `.c` to rozszerzenie plików źródłowych, a nie nagłówkowych czy obiektowych):

```
(...)\main.c: In function 'main':
(...)\main.c:7:5: error: expected ',' or ';' before 'int'
     int n3 = n * n;
```

[6] Flagi kompilacji zostały dokładniej omówione w rozdziale *2.2.4 Flagi kompilacji*

```
     ^~~
(...)\main.c:6:12: warning: unused variable 'n' [-Wunused-variable]
     double n = 3
            ^
```

- Pierwszy problem (błąd) stanowi przykład **błędu składniowego** (ang. syntax error), czyli niezgodności z regułami języka określonymi w jego standardzie – to błąd analogiczny do błędu gramatycznego w języku polskim. Zwróć uwagę, że został on zgłoszony w linii 7 (na pozycji 5), choć do dobrej praktyki należy umieszczanie średnika na końcu linii zawierającej instrukcję – czyli w linii 6.
- Drugi problem (ostrzeżenie) wynika stąd, że w programie nie powinny występować zmienne w istocie niewykorzystywane. Jak to – spytasz – zmienna `n` nie jest wykorzystywana? Cóż. . . ponieważ w obecnym kodzie brakuje średnika, kompilator nie przetwarza poprawnie instrukcji `int n3 = n * n;` i odnosi wrażenie, że faktycznie zmienna `n` nie jest wykorzystywana – i zgłasza to ostrzeżenie w linii 6.

Po dodaniu brakującego średnika w linii 6 i ponownej próbie zbudowania programu otrzymasz komunikat:

```
(...)\main.c: In function 'main':
(...)\main.c:7:9: warning: unused variable 'n3' [-Wunused-variable]
     int n3 = n * n;
         ^~
(...)/objects.a(main.c.obj): In function `main':
(...)/main.c:9: undefined reference to `foo'
```

- Ostrzeżenie o nieużywanej zmiennej wciąż się pojawia, jednak tym razem pojawia się w linii 7 i dotyczy zmiennej `n3` (czyli jest niejako zgodne z oczekiwaniami).
- Błąd „`undefined reference to `foo'`" to błąd konsolidacji (świadczy o tym rozszerzenie pliku, w którym znaleziono błędy – `.a` to rozszerzenie plików bibliotek statycznych zawierających kod obiektowy, a nie plików źródłowych) – kod programu jest poprawny od strony zgodności ze składnią języka C, jednak nie zawiera implementacji funkcji `foo()`, przez co nie mógłby zostać poprawnie wykonany.

Jak widzisz, często poprawa jednych błędów powoduje pojawienie się „nowych" błędów, będących w istocie błędami cały czas istniejącymi w programie, tyle że nie zgłoszonymi od razu przy pierwszej próbie jego budowania.

Zauważ, że program wciąż zawiera **błąd semantyczny** lub inaczej **błąd znaczeniowy** (ang. semantic error), czyli nie jest poprawny od strony logicznej[7] – w linii 7 zmienna `n3` otrzymuje wartość n^2 zamiast zakładanej n^3. Ani kompilator ani konsolidator nie wychwycą błędów semantycznych, gdyż nie są one związane z niezgodnością z regułami języka C – oznaczają jedynie, że program wykona się, lecz nie w pełni zgodnie z intencją jego twórcy. Aby zmniejszyć ryzyko wystąpienia błędów semantycznych warto pisać wyczerpujące testy jednostkowe.

2.2.3 Dobre praktyki usuwania błędów i ostrzeżeń

Oto dobre praktyki związane z błędami i ostrzeżeniami:

- W pierwszej kolejności usuwaj błędy (w takiej kolejności, w jakiej się pojawiają – od pierwszej linii pliku do ostatniej), a dopiero potem zajmuj się ostrzeżeniami.
- Błąd składniowy w jednym miejscu jest w stanie sprawić, że kompilator doszuka się błędów także w innych, zupełnie poprawnych miejscach (przykładowo: nieprawidłowa deklaracja zmiennej wywoła dalsze błędy we wszystkich miejscach, w których zmienna ta została użyta).
- Kompilator zgłasza niektóre błędy o jedną linię kodu za późno (przykładowo: kompilator może nie zorientować się, że w danej linijce brakuje średnika, dopóki nie spróbuje skompilować kolejnego wiersza programu), dlatego w usunięciu błędu często pomaga kontrola otoczenia miejsca jego wystąpienia.
- Czytaj opisy błędów i ostrzeżeń! Bardzo często to wystarczy, by zorientować się, jak je usunąć.

[7] Przykładem błędu semantycznego w języku polskim może być zdanie „Czarna inflacja myśli powoli." Jest to zdanie poprawne gramatycznie, jednak czy inflacja może być czarna oraz czy to istota myśląca?

2.2.4 Flagi kompilacji

Flagi kompilacji umożliwiają do pewnego stopnia sterowanie procesem kompilacji – w przypadku kompilatora języka C z pakietu GCC flagi pozwalają określić m.in. stosowaną wersję standardu języka oraz „czujność" kompilatora (i linkera). Aby tworzyć kod w pełni zgodny ze standardami języka C (co znacznie uprości życie Tobie i Twoim współpracownikom) i możliwie jednoznaczny, kompiluj programy z użyciem co najmniej poniższego zestawu flag[8] (najlepiej ustaw je jako domyślne opcje kompilatora):

- `-std=c11` – korzystaj ze standardu C11 (ew. `-std=c99` dla C99)
- `-Wall` – włącz ostrzeżenia o wszystkich tych potencjalnych problemach z kodem, które część użytkowników uważa za dyskusyjne
- `-Wextra` – włącz pewnie dodatkowe ostrzeżenia, których nie włącza `-Wall` (wciąż nie wszystkie. . .)
- `-Werror` – traktuj wszystkie ostrzeżenia jako błędy
- `-Wpedantic` – włącz wszystkie ostrzeżenia wymagane przez standard ICO C; odrzuć programy korzystające z niedozwolonych rozszerzeń bądź nie stosujące się do standardów
- `-pedantic-errors` – zgłoś błąd w każdym przypadku, gdy `-Wpedantic` informuje o konieczności diagnostyki kodu

2.2.5 Debugowanie

Jednym z najprostszych sposobów wykrywania błędów semantycznych jest umieszczenie w programie dodatkowych instrukcji wypisujących komunikaty na standardowe wyjście, które pozwalają śledzić przebieg wykonania programu i stan zmiennych. Jednak często lepszym rozwiązaniem jest skorzystanie ze specjalistycznego narzędzia, tzw. **debuggera**, czyli programu służącego do wykonania innego programu w sposób krokowy (tj. instrukcja po instrukcji), z możliwością podglądu stanu programu (a więc m.in. wartości zmiennych) po każdym kroku.

Ważne

Debugger nie jest częścią języka – to narzędzie dostarczane (zazwyczaj) w ramach zintegrowanego środowiska programistycznego.

Więcej o debugowaniu w wybranych zintegrowanych środowiskach programistycznych przeczytasz tu.

[8] Pełny opis tych flag znajdziesz w dokumentacji kompilatora GCC.

Rozdział 3

Dane w języku C

W tym rozdziale dowiesz się o podstawowych *arytmetycznych* typach danych używanych w C, między innymi o różnicach pomiędzy reprezentacjami liczb całkowitych i rzeczywistych.

3.1 Bity, bajty, słowa

- Najmniejszą jednostką pamięci jest **bit** (ang. bit). Może on przechowywać jedną z dwóch wartości: 0 („wyłączony") lub 1 („włączony").
- **Bajt** (ang. byte) jest najczęściej używają jednostką pamięci komputera. Jeden bajt składa się *zazwyczaj* z 8 bitów[1].
- Naturalną jednostką pamięci dla danego typu komputera jest **słowo** (in. **słowo maszynowe**, ang. word), o długości zależnej od typu procesora. Dla komputerów z procesorem 32-bitowym będą to 4 bajty, dla procesora 64-bitowego – 8 bajtów itd.

3.2 Systemy pozycyjne

System pozycyjny to metoda zapisywania liczb (in. system liczbowy) w taki sposób, że w zależności od pozycji danej cyfry w ciągu, oznacza ona wielokrotność potęgi pewnej liczby uznawanej za bazę danego systemu:

$$x = p_n p_{n-1} \dots p_0 = \sum_{i=0}^{n} p_i B^i,$$

gdzie:

- p_i – i-ta pozycja liczby
- B (ang. base) – baza systemu liczbowego

W używanym powszechnie systemie dziesiętnym za bazę przyjmuje się liczbę dziesięć (przykładowo: $123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$). Ponieważ komputerowy bit może przyjmować tylko dwie wartości – 0 lub 1 – naturalnym dla komputera systemem liczbowym jest system o podstawie 2, czyli **system dwójkowy**. Liczby wyrażone w systemie dwójkowym nazywamy **liczbami binarnymi**. Na przykład, liczba dwójkowa 1101_2 oznacza:
$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$

Informatycy często wykorzystują systemy ósemkowy i szesnastkowy, ponieważ są one bliższe systemowi binarnemu niż system dziesiętny (dzięki temu, że 8 i 16 są potęgami dwójki), a jednocześnie są bardziej zwięzłe niż system dwójkowy:

- W systemie ósemkowym każda cyfra ósemkowa odpowiada trzem cyfrom binarnym. Ułatwia to przeliczanie wartości z jednego systemu na drugi, przykładowo: $0377_8 = 11111111_2$ – cyfrę 3 zamieniliśmy na `011`, a następnie każdą z siódemek zamieniliśmy na `111` (w końcowym zapisie opuściliśmy zero wiodące). W informatyce liczby ósemkowe zwyczajowo zapisuje się z przedrostkiem `0`, czyli np. `0377`.
- W systemie szesnastkowym cyfry od 10 do 15 są wyrażane za pomocą liter od A do F (przykładowo $\mathrm{A3F}_{16} = 10 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 = 2623$). W informatyce liczby szesnastkowe zwyczajowo

[1] Język C używa terminu „bajt" do określenia liczby bitów potrzebnej do przechowania całego zestawu znaków, a więc teoretycznie jeden bajt mógłby mieć długość 8, 9, 16 czy jakiejkolwiek innej liczby bitów. Jednak opisując pojemności kości pamięci lub prędkości transferu danych, mamy na myśli bajt o długości 8 bitów.

zapisuje się z przedrostkiem `0x`, czyli np. `0xA3F`. Każda cyfra szesnastkowa odpowiada 4-cyfrowej liczbie binarnej, a więc dwie cyfry szesnastkowe odpowiadają jednemu 8-bitowemu bajtowi. (Pierwsza cyfra przedstawia 4 bardziej znaczące bity.) Własność ta sprawia, że system szesnastkowy jest naturalnym sposobem reprezentacji wartości o rozmiarze będącym wielokrotnością jednego bajtu (czyli praktycznie wszystkich wartości, jakie wykorzystuje komputer).

3.3 Liczby binarne

3.3.1 Binarne liczby całkowite

Osiem bitów, jakie (zazwyczaj) składają się na jeden bajt, można ponumerować liczbami od 7 do 0, w kierunku od lewej do prawej: bit nr 7 nazywamy wówczas **bitem najbardziej znaczącym** (ang. most significant bit, MSB, high-order bit), a bit nr 0 – **bitem najmniej znaczącym** (ang. least significant bit, LSB, low-order bit). Numer bitu odpowiada pozycji w liczbie binarnej (czyli wykładnikowi potęgi liczby 2). Największa liczba, jaką może przechować 8-bitowy bajt, powstaje przez nadanie wszystkim bitom wartości 1; liczbą tą jest $11111111_2 = 2^7 + 2^6 + \ldots + 2^0 = 255$. Najmniejszą liczbą, jaką można zapisać w jednym bajcie, jest $00000000_2 = 0$. Jeden bajt może zatem przedstawiać liczby od 0 do 255, co daje w sumie 256 różnych wartości. Zmieniając interpretację liczb binarnych, można sprawić, aby bajt wyrażał liczby od -128 do $+127$ (tak jak poprzednio, istnieje 256 różnych wartości).

3.3.2 Liczby całkowite ze znakiem

Reprezentacja liczb ze znakiem jest określona sprzętowo, a nie przez język C. Prawdopodobnie najprostszym sposobem na wyrażenie znaku liczby jest przechowywanie go w jednym wydzielonym bicie, np. bicie najbardziej znaczącym. W przypadku wartości jednobajtowej, pozostawia to 7 bitów na samą liczbę. W takim zapisie, noszącym nazwę **reprezentacji znak-moduł**, `00000001` to 1, a `10000001` to -1. Dostępny zakres wartości obejmuje więc liczby od -127 do $+127$. Istotną wadą tego systemu jest to, iż występują w nim dwa zera: $+0$ i -0. Jest to kłopotliwe, a także nieoszczędne, ponieważ wyraża tę samą wartość za pomocą dwóch różnych układów bitów. Odporna na ten problem jest **metoda dopełnienia dwójkowego** (ang. twos complement method), która jest najpopularniejszym używanym obecnie systemem zapisu. Omówimy ją na przykładzie wartości 1-bajtowej. Wartości od 0 do 127 są reprezentowane przez liczby od `00000000` do `01111111` (najbardziej znaczący bit jest dla nich równy 0). Jeśli najbardziej znaczący bit jest włączony, wartość jest ujemna. Jak na razie system nie różni się od reprezentacji znak-moduł. Różnica tkwi w sposobie określania wartości liczb ujemnych. W metodzie dopełnienia dwójkowego moduł (wartość bezwzględną) wartości ujemnej otrzymujemy przez odjęcie jej od 9-bitowej liczby `100000000` (256). Na przykład, załóżmy, że naszą liczbą jest `10000000`. Jako wartość bez znaku byłaby ona równa 128. Jako wartość ze znakiem jest ona ujemna (ponieważ siódmy bit jest równy 1), a jej moduł wynosi `100000000 - 10000000`, czyli `10000000` (128). Stąd liczba binarna `10000000` jest równa -128. (W systemie znak-moduł byłaby ona równa -0.) Analogicznie `10000001` to -127, a `11111111` to -1. Metoda dopełnienia dwójkowego wyraża zatem liczby z przedziału od -128 do $+127$. Najprostszym sposobem na odwrócenie znaku liczby binarnej zapisanej w systemie dopełnienia dwójkowego jest odwrócenie każdego bitu (zmiana zer na jedynki, a jedynek na zera) i dodanie 1. Na przykład, ponieważ 1 to `00000001`, -1 jest równe `11111110 + 1`, czyli (jak przekonałeś się wcześniej) `11111111`.

3.3.3 Liczby zmiennoprzecinkowe

Wartość liczby zmiennoprzecinkowej jest obliczana według wzoru:

$$x = SMB^E,$$

gdzie:

- S (ang. sign) – znak liczby (1 lub -1),
- M (ang. mantissa) – znormalizowana mantysa (liczba ułamkowa),
- B (ang. base) – podstawa systemu liczbowego (np. 2, 10),
- E (ang. exponent) – wykładnik, cecha (liczba całkowita).

Mantysa jest znormalizowana, tj. należy do przedziału $[1, B)$ (przedział prawostronnie otwarty!).

3.4 Po co nam typy danych?

Określenie oczekiwanego rodzaju danych pozwala komputerowi przechowywać, pobierać i interpretować je we właściwy sposób. Język C pozwala na stosowanie wielu rodzajów (typów) danych, służących reprezentacji m.in. liczb całkowitych, liczb rzeczywistych oraz znaków.

3.5 Typ całkowity a typ zmiennoprzecinkowy

Dla człowieka różnica między **liczbami całkowitymi** (ang. integers) a **liczbami rzeczywistymi** (ang. real numbers) wyraża się w sposobie ich zapisu; dla komputera różnica między liczbami całkowitymi a **liczbami zmiennoprzecinkowymi** (ang. floating-point numbers) – które mniej lub bardziej odpowiadają temu, co matematycy nazywają liczbą rzeczywistą – polega głównie na odmiennym sposobie ich przechowywania.

Innymi słowy, dla człowieka (w ujęciu matematycznym) liczby 0 i 0.0 są zarówno równe, jak i *tożsame*, natomiast dla komputera – nie. Choć dla komputera te wartości są sobie równe, to należą do różnych typów danych, a więc: są inaczej reprezentowane w pamięci komputera.

Co więcej, ponieważ język C został zaprojektowany m.in. jako język programowania sprzętowego, umożliwia on efektywne wykorzystanie dostępnych zasobów systemowych – w tym zasobów pamięci operacyjnej. W związku z tym udostępnia on wiele różnych typów dla przechowywania wartości całkowitych oraz wiele różnych typów dla przechowywania wartości zmiennoprzecinkowych – w zależności od tego, jaki zakres wartości potrzebujemy reprezentować oraz (w przypadku liczb całkowitych) czy potrzebujemy reprezentować wyłącznie wartości nieujemne, czy również ujemne.

3.6 Literały

Literał (ang. literal) to jednostka leksykalna reprezentująca ustaloną wartość (liczbową, tekstową, itp.) wpisaną przez programistę bezpośrednio w danym miejscu w kod programu – wartość ta nie ulega zmianie podczas wykonywania programu, stąd literał jest również nazywany **stałą** (ang. constant).

Język C udostępnia literały dla podstawowych typów danych służących do reprezentowania wartości całkowitych i zmiennoprzecinkowych, znaków, łańcuchów znaków oraz etykiet typów wyliczeniowych.

Oto przykłady literałów:

- `4` – literał całkowity (w systemie dziesiętnym)
- `0xA7` – literał całkowity (w systemie szesnastkowym)
- `'A'` – literał znakowy

Literały odpowiadające poszczególnym typom danych będą wprowadzane sukcesywnie w kolejnych podrozdziałach.

3.7 Typy całkowite

W zależności od potrzeb język C udostępnia typy służące do reprezentacji **liczb całkowitych ze znakiem** (ang. signed integers) oraz **liczb całkowitych bez znaku** (ang. unsigned integers) – typy bez znaku umożliwiają reprezentowanie wyłącznie wartości nieujemnych. W ramach każdej z tych kategorii dostępne są typy służące do reprezentacji liczb z różnych zakresów, różniące się rozmiarem (czyli liczbą bajtów niezbędnych do przechowania obiektu danego typu).

3.7.1 Podstawowy typ całkowity: `int`

Podstawowym typem służącym do reprezentacji liczb całkowitych ze znakiem (czyli przykładowo -3 i 5) jest **`int`**.

Standard języka C określa jedynie, że typ **`int`** musi być w stanie przechowywać *co najmniej* wartości z zakresu $[-32,767, +32,767]$, a więc równoważnie z zakresu $[-2^{15}+1, +2^{15}-1]$, czyli musi mieć szerokość *co najmniej* 16 bitów (ale np. na większości komputerów PC typ **`int`** ma szerokość co najmniej 32 bitów).

W języku C literały wyrażające liczby całkowite nigdy nie zawierają kropki dziesiętnej (zatem literały `2` i `-3` oznaczają liczby całkowite ze znakiem, natomiast literał `2.0` oznacza liczbę zmiennoprzecinkową).

Literały wyrażające liczby całkowite w systemie ósemkowym posiadają przedrostek `0` (np. `0377`), natomiast w systemie szesnastkowym – przedrostek `0x` (np. `0xA3F`).

3.7.2 Modyfikatory rozmiaru

Czasem zachodzi potrzeba reprezentowania wartości całkowitych spoza zakresu typu **`int`** albo też mniejszy zakres byłby wystarczający. Wówczas język C udostępnia trzy modyfikatory – **`short`**, **`long`** i **`long long`**[2] – przy czym określa jedynie, że:

- Typ **`short int`** musi być w stanie przechowywać *co najmniej* wartości z zakresu $[-32,767, +32,767]$ (równoważnie: $[-2^{15}+1, +2^{15}-1]$), czyli musi mieć szerokość *co najmniej* 16 bitów.
- Typ **`long int`** (równoważnie **`long`**) musi być w stanie przechowywać *co najmniej* wartości z zakresu $[-2^{31}+1, +2^{31}-1]$, czyli musi mieć szerokość *co najmniej* 32 bitów.
- Typ **`long long int`** (równoważnie **`long long`**) musi być w stanie przechowywać *co najmniej* wartości z zakresu $[-2^{63}+1, +2^{63}-1]$, czyli musi mieć szerokość *co najmniej* 64 bitów.

Zwróć uwagę, że powyższe typy muszą umożliwiać przechowywanie wartości *co najmniej* z pewnego zakresu – zatem, przykładowo, wszystkie powyższe typy mogłyby reprezentować liczbę 64-bitową. Nie masz zatem gwarancji, że stosując typ **`short`** zamiast **`int`** oszczędzasz pamięć operacyjną (to zależy od Twojej architektury komputera i od kompilatora)! Jeśli stosowanie typu o ściśle określonej szerokości jest istotne, język C udostępnia tzw. typy o stałej szerokości (zob. rozdz. *18.5.3 Typy o stałej szerokości, typy o minimalnej szerokości...*). Takie luźne (jedynie minimalne) wymogi pozwalają lepiej dopasować typy do konkretnej platformy sprzętowej.

W przypadku podania modyfikatora rozmiaru można pominąć **`int`**, czyli **`short int`** jest równoważne **`short`** itd.

Literały dla typu **`long`** mają przyrostek `L` (równoważnie `l`), np. `4L`, natomiast literały dla typu **`long long`** – przyrostek `LL` (równoważnie `ll`), np. `4LL`. Zauważ, że wielką literę `L` trudniej pomylić z cyfrą `1` niż małą literę `l`.

3.7.3 Modyfikatory znaku

Do określenia tego, czy liczba całkowita ma posiadać znak, czy też nie, służą modyfikatory **`signed`** (tj. „ze znakiem") i **`unsigned`** (tj. „bez znaku"), przykładowo:

- **`signed int`** – typ **`int`** *ze znakiem*
- **`unsigned int`** – typ **`int`** *bez znaku*

W języku C typy całkowite domyślnie posiadają znak, zatem nie ma potrzeby dodawania modyfikatora **`signed`** w sytuacji, gdy pożądany jest właśnie typ całkowity ze znakiem.

O ile (hipotetyczny) n-bitowy typ całkowity ze znakiem `T` umożliwia przechowywanie *co najmniej* wartości z zakresu $[-2^{n-1}+1, +2^{n-1}-1]$, o tyle analogiczny (hipotetyczny) typ całkowity bez znaku **`unsigned`** `T` umożliwia przechowywanie *co najmniej* wartości z zakresu $[0, 2^n-1]$. Przykładowo, 16-bitowy typ **`int`** umożliwia przechowywanie *co najmniej* wartości z zakresu $[-2^{15}+1, +2^{15}-1]$, a typ **`unsigned int`** – *co najmniej* wartości z zakresu $[0, 2^{16}-1]$.

Aby utworzyć stałą całkowitą bez znaku, należy dodać przyrostek `U` (od *unsigned*), np. `6U`.

Literały wyrażające liczby całkowite *bez znaku* w systemie ósemkowym i szesnastkowym oprócz odpowiedniego *przedrostka* (odpowiednio `0` albo `0x`) muszą również posiadać przyrostek `U`, np. `0377U`, `0xA3FU`.

3.8 Typy zmiennoprzecinkowe

W języku C do reprezentacji liczb zmiennoprzecinkowych służą trzy typy danych:

- **`float`** – „zwykła" liczba **zmiennoprzecinkowa** (ang. floating point)
- **`double`** – liczba zmiennoprzecinkowa **o podwójnej precyzji** (ang. double-precision)
- **`long double`** – wg standardu języka C „dokładność typu **`long double`** jest co najmniej taka, jak typu **`double`**" (ale na razie zwykle oba te typy mają taką samą dokładność)

pozwalające na przechowywanie liczb zmiennoprzecinkowych z różną dokładnością (zob. tab. 3.1).

Różne formy notacji liczb rzeczywistych przedstawia tabela 3.2, przy czym do zapisu liczb zmiennoprzecinkowych w informatyce zwykle stosuje się notację naukową „E" (od *exponent* – wykładnik). Notacja naukowa „E" to skrócona forma podstawowej notacji naukowej – nie podaje się w niej symbolu mnożenia oraz podstawy potęgi (10), zatem ma ona postać:

$$xen,$$

[2] Typ **`long long`** został wprowadzony w standardzie C99.

Tablica 3.1. Liczba cyfr znaczących i zakres wykładnika typów zmiennoprzecinkowych (minima wg standardu języka C)

typ	cyfry znaczące	zakres wykładnika
`float`	6	−37 do 37
`double`	10	−37 do 37

gdzie:

- x – skończona liczba rzeczywista dziesiętna,
- n – wykładnik (liczba całkowita),

przy czym zamiast litery `e` można równoważnie użyć litery `E`. Przykładowo $3.16E7$ oznacza 3.16×10^7.

Tablica 3.2. Notacja liczb rzeczywistych

Liczba	Notacja naukowa	Notacja naukowa „E”
1,000,000,000	1.0×10^9	1.0e9
123.000	1.23×10^5	1.23e5
322.56	3.2256×10^2	3.2256e2
0.000,056	5.6×10^{-5}	5.6e−5

Podstawowa postać literału zmiennoprzecinkowego odpowiada notacji wykładniczej, przy czym w każdym przypadku możesz pominąć:

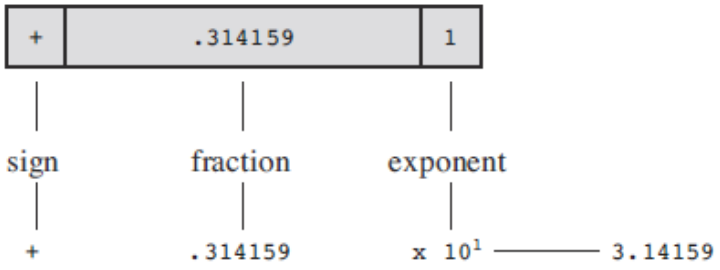
- znak `+` (np. `2.0e3` jest równoważne `+2.0e+3`), a także
- kropkę dziesiętną (np. `2e5` jest równoważne `2.0e5`) albo część wykładniczą (np. `19.28` jest równoważne `19.28e1`) – ale nie możesz pominąć obu tych elementów na raz (gdyż wówczas otrzymasz literał całkowity).

Powyższe literały wyrażają wartości typu `double` – aby uzyskać wartości typu `float` należy dodać przyrostek `f` (równoważnie `F`), np. `1.8f`. Literał dla typu `long double` otrzymuje się poprzez dodanie do literału dla typu `double` przyrostka `L` (równoważnie `l`), np. `4.3L`.

Ważne

W języku C przy zapisie liczb rzeczywistych używamy kropki dziesiętnej (a nie przecinka)!

W języku C reprezentację wartości zmiennoprzecinkowej w pamięci komputera określa standard IEC 60559:1989[3] – składa się ona z bitu znak, pewnej liczby bitów (różnej w zależności od systemu) przeznaczonej na mantysę i kilku dalszych bitów przechowujących wykładnik (zob. rys. 3.1).



Rysunek 3.1. Przechowywanie liczby π w postaci zmiennoprzecinkowej (dziesiętnej)

Do reprezentowania pewnych specjalnych wartości służą stałe symboliczne zdefiniowane w pliku nagłówkowym `math.h`:

- `NAN` – reprezentuje wartość nie będącą liczbą rzeczywistą (np. $\sqrt{-1}$)
- `INFINITY` – reprezentuje nieskończoność (dodatnią)

[3] równoważnie: standard IEEE 754-2008

3.9 Typy znakowe

Typ **char** reprezentuje pojedynczy znak (ang. character), np. literę „A” lub cyfrę „3”.

Oto jak wygląda korzystanie z typu znakowego w języku C:

- Ponieważ komputer operuje na liczbach (a nie znakach), każdy znak ma przyporządkowany kod numeryczny – znaki są przechowywane w pamięci jako liczby całkowite.
- Standard języka C gwarantuje, że typ **char** jest wystarczająco pojemny, aby przechować podstawowy zestaw znaków komputera, na który napisano kompilator. Historycznym standardem kodowania był standard ASCII, który w wersji podstawowej definiował 128 kodów znaków, a w wersji rozszerzonej (przez IBM PC) – 256 kodów znaków. W związku z tym do jego przechowywania należało użyć 8 bitów.
- Kompilator definiuje w pliku `limits.h` stałą symboliczną `CHAR_BIT`, która wyraża liczbę bitów składających się na bajt (czyli najmniejszą jednostkę pamięci umożliwiającą przechowanie całego zestawu znaków). Na większości współczesnych komputerów bajt składa się z 8 bitów[4].

Literały znakowe, lub inaczej **stałe znakowe** (ang. character constants) wyrażane są poprzez ujęcie znaku w apostrofy (np. `'A'`, `'3'`). Niektóre znaki mają specjalne znaczenie (np. przejście do nowego wiersza) i nie da się ich zapisać wprost z użyciem „zwykłego” zapisu z apostrofami. Można się do nich odwołać z użyciem kodu ASCII (rozwiązanie niezalecane) albo skorzystać z tzw. **sekwencji sterujących** (ang. escape sequences), czyli specjalnych ciągów symboli zawierających backslash, np.:

- `\n` (kod ASCII: 13) – przejście do nowego wiersza
- `\t` (kod ASCII: 9) – tabulator poziomy
- `\"` – cudzysłów
- `\'` – apostrof
- `\\` – backslash

Znak przejścia do nowego wiersza i znak tabulatora to przykłady **znaków niedrukowanych**.

Powiedzmy, że programista chcąc utworzyć zmienną `ch` przechowującą znak „A” zastosował następującą instrukcję (wykorzystującą kody numeryczne):

```
char ch = 65;  // OK dla kodu ASCII, ale ogólnie to kiepski styl
```

Założył on, że kompilator korzysta z kodowania ASCII, jednak ten kod nie będzie przenośny – nie będzie działał na innych komputerach wykorzystujących inny system kodowania znaków. Poprawny sposobem inicjalizacji zmiennej `ch` jest

```
char ch = 'A';  // rozwiązanie PRZENOŚNE
```

Standard języka C nie określa tego, czy typ **char** jest typem posiadającym znak, zatem można spotkać dwie implementacje (zależne od kompilatora):

- ze znakiem – **char** może przechowywać wartości z zakresu od -128 do $+127$
- bez znaku – **char** może przechowywać wartości z zakresu od 0 do 255

Informacja o zakresie typu **char** znajduje się m.in. w pliku nagłówkowym `limits.h`, jako stałe symboliczne `CHAR_MIN` i `CHAR_MAX`. Kompilatory zgodne ze standardem języka C pozwalają dodawać do **char** słowa kluczowe **signed** lub **unsigned**, co pozwala uniknąć wspomnianej niejednoznaczności.

3.10 Typ logiczny

Do reprezentowania wartości logicznych („prawda” i „fałsz”) służy typ `_Bool`, który umożliwia przechowywanie wartości 0 i 1[5]. Teoretycznie typ ten mógłby mieć rozmiar 1 bitu, lecz w praktyce (ze względów praktycznych) zajmuje zwykle 1 bajt.

Plik nagłówkowy `stdbool.h` udostępnia trzy wygodne makra:

- `bool` jako alias dla typu `_Bool`
- `false` (fałsz) jako alias dla wartości 0
- `true` (prawda) jako alias dla wartości 1

[4] Natomiast platforma sprzętowa wykorzystująca Unicode jako podstawowy zestaw znaków mogłaby posiadać typ **char** o długości 16 bitów.

[5] Typ ten został wprowadzony w standardzie C99.

Ważne

Wszystkie wartości poza 0 są traktowane jako „prawda”.

3.11 Kiedy stosować który typ arytmetyczny?

W swoich programach domyślnie korzystaj z typów:

- `int` – do przechowywania liczb całkowitych (pod kątem operacji arytmetycznych)
- `double` – do przechowywania liczb zmiennoprzecinkowych (pod kątem operacji arytmetycznych)
- `char` – do przechowywania znaków

O tym, kiedy stosować typ ze znakiem, a kiedy bez znaku przeczytasz tu.

3.12 Inne typy danych

W zakresie typów arytmetycznych, oprócz omówionych we wcześniejszych rozdziałach typów całkowitych i zmiennoprzecinkowych oraz typu znakowego i logicznego język C udostępnia typy służące do obsługi liczb zespolonych i urojonych (zob. plik nagłówkowy `complex.h`).

Język C udostępnia również typy wyliczeniowe oraz typy pochodne: typy funkcyjne, wskaźniki, tablice, struktury i unie. Typy te zostaną omówione w dalszych rozdziałach.

3.12.1 Typ `void`

Typ `void` posiada pusty zbiór wartości – jest to zatem tzw. **typ niekompletny** (ang. incomplete type) i nie ma możliwości utworzenia obiektu tego typu. Jest on wykorzystywany m.in. podczas pracy ze wskaźnikami i funkcjami (w tym drugim przypadku do oznaczania funkcji, które nie zwracają wartości oraz które nie przyjmują żadnych argumentów). Zastosowania typu `void` zostaną dokładniej omówione w dalszych rozdziałach.

3.13 Rozmiary i zakresy typów

Język C zawiera wbudowany operator o nazwie `sizeof`, który zwraca rozmiar argumentu w bajtach, np. `sizeof(int)` zwróci szerokość typu `int` w bajtach.

Wartość zwracana przez operator `sizeof` jest typu `size_t`[6]. Standard języka C ogranicza się do stwierdzenia, że „operator `sizeof` zwraca typ całkowity” – `size_t` może być więc w zależności od systemu typem `unsigned int` lub `unsigned long` itd. Definicja typu `size_t` dla Twojego kompilatora znajduje się m.in. w plikach nagłówkowych `stddef.h`, `stdio.h` i `stdlib.h`.

Oprócz tego w plikach nagłówkowych `limits.h` i `float.h` zdefiniowane są stałe symboliczne określające m.in. zakresy wartości reprezentowalne przez poszczególne typy na danej architekturze komputera odpowiednio dla typów całkowitych (`limits.h`) i zmiennoprzecinkowych (`float.h`). Mają one (w większości) ogólną postać `T_MIN` i `T_MAX`, oznaczające odpowiednio minimalne i maksymalne wartości reprezentowalne z użyciem typu `T`, przykładowo: `INT_MIN` i `INT_MAX`, `LONG_MIN` i `LONG_MAX`, `DBL_MIN` i `DBL_MAX` itd.

3.14 Arytmetyka liczb w C – pułapki

Język C udostępnia m.in. cztery podstawowe operatory matematyczne: dodawanie (`+`), odejmowanie (`-`), mnożenie (`*`) i dzielenie (`/`). Pomogą one pokazać wybrane problemy, które mogą wyniknąć podczas nieumiejętnego obchodzenia się z danymi poszczególnych typów.

3.14.1 Przekroczenie zakresu liczb całkowitych

Do **przekroczenia zakresu liczby całkowitej** (ang. integer overflow) dochodzi przy próbie wyjścia poza zakres wartości reprezentowalny przez dany typ całkowity *ze znakiem*. Termin „przepełnienie” może być tu nieco mylący, gdyż oznacza zarówno wyjście poza maksymalną, jak i poza minimalną wartość zakresu.

Standard jezka C stwierdza dwie rzeczy:

- W przypadku wartości będącej typu całkowitego *bez znaku* `T` nigdy nie dochodzi do przepełnienia, gdyż wartości spoza zakresu są redukowane za pomocą operacji „modulo wartość `T_MAX + 1`”.

[6] Typ `size_t` jest w istocie aliasem – o aliasach przeczytasz w rozdziale *18.5 Specyfikator `typedef`*.

- W przypadku wartości będącej typu całkowitego *ze znakiem* wykonanie operacji powodującej wyjście poza zakres reprezentowalnych wartości prowadzi do niezdefiniowanego zachowania programu.

W praktyce jednak większość kompilatorów stosuje dla typów całkowitych ze znakiem poniższe dwie zasady:

- `T_MAX + 1 == T_MIN`, `T_MAX + 2 == T_MIN + 1` itd.
- `T_MIN - 1 == T_MAX`, `T_MIN - 2 == T_MAX - 1` itd.

Poniższy program demonstruje przykłady przepełnienia typu całkowitego ze znakiem i bez znaku:

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

int main(void) {
    int v_int = INT_MAX;
    unsigned int v_uint = UINT_MAX;

    printf("          %12s %12s %12s\n", "val", "val + 1", "val + 2");

    // specyfikator %u wyświetla wartości typu `unsigned int`
    printf(" int max: %12d %12d %12d\n", v_int, v_int + 1, v_int + 2);
    printf("uint max: %12u %12u %12u\n", v_uint, v_uint + 1, v_uint + 2);

    v_int = INT_MIN;
    v_uint = 0U;

    printf(" int min: %12d %12d %12d\n", v_int, v_int - 1, v_int - 2);
    printf("uint min: %12u %12u %12u\n", v_uint, v_uint - 1, v_uint - 2);

    return EXIT_SUCCESS;
}
```

Wynik uruchomienia programu:

```
                    val       val + 1       val + 2
 int max:    2147483647  -2147483648  -2147483647
uint max:    4294967295             0             1
 int min:   -2147483648   2147483647   2147483646
uint min:             0   4294967295   4294967294
```

3.14.2 Przepełnienie i niedomiar liczb zmiennoprzecinkowych

W języku C reprezentację wartości zmiennoprzecinkowej w pamięci komputera oraz zachowanie w przypadku wystąpienia przepełnienia lub niedomiaru określa standard IEC 60559:1989:

- Do **przepełnienia liczby zmiennoprzecinkowej** dochodzi w sytuacji, gdy wartość *wykładnika* jest zbyt duża do reprezentowania z użyciem przeznaczonych do tego bitów – wówczas wynikiem jest specjalna wartość, odpowiednio `-INFINITY` ($-\infty$) albo `+INFINITY` ($+\infty$), w zależności od znaku mantysy.
- Do **niedomiaru liczby zmiennoprzecinkowej** (ang. underflow) dochodzi, gdy wartość *wykładnika* jest zbyt mała do reprezentowania z użyciem przeznaczonych do tego bitów *przy założeniu reprezentacji mantysy w postaci znormalizowanej* (tj. jako wartość z zakresu $[1, B]$, gdzie B oznacza podstawę systemu liczbowego). Wówczas wynikiem jest albo 0 albo tzw. wartość *subnormalna* posiadająca nienormalizowaną mantysę (tj. mantysę z zakresu $(0, B)$)[7].

Poniższy program demonstruje przykłady przepełnienia i niedomiaru typu zmiennoprzecinkowego:

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

[7] Informację o tym, czy dany typ zmiennoprzecinkowy wspiera wartości subnormalne, uzyskasz za pomocą makr `T_HAS_SUBNORM` w pliku nagłówkowym `float.h`

```c
int main(void) {
    double val_overflow = DBL_MAX * 2.0;
    double val_underflow = DBL_MIN * DBL_MIN;

    printf("overflow = %lf , underflow = %lf\n", val_overflow, val_underflow);

    return EXIT_SUCCESS;
}
```

Wynik uruchomienia programu:

```
overflow = 1.#INF00 , underflow = 0.000000
```

3.14.3 Utrata cyfr znaczących

Do **utraty cyfr znaczących** lub inaczej **obcięcia** (ang. truncation) liczby dochodzi w dwóch przypadkach.

Pierwszy przypadek dotyczy dzielenia przez siebie dwóch liczb całkowitych i polega na odrzuceniu części ułamkowej wyniku dzielenia[8]. Przykładowo, wynikiem dzielenia `7 / 4` (czyli liczby typu całkowitego przez liczbę typu całkowitego) jest liczba typu całkowitego o wartości 1. Dla porównania:

- Wynikiem dzielenia `7 / 4.0` (czyli liczby typu całkowitego przez liczbę typu zmiennoprzecinkowego) jest liczba typu zmiennoprzecinkowego o wartości najbliższej liczbie rzeczywistej $1,75$ – gdyż nie wszystkie liczby rzeczywiste są reprezentowalne z użyciem liczby zmiennoprzecinkowej (o skończonej liczbie bitów).
- Wynikiem dzielenia `7.0 / 4` (czyli liczby typu zmiennoprzecinkowego przez liczbę typu całkowitego) jest liczba typu zmiennoprzecinkowego o wartości najbliższej liczbie rzeczywistej $1,75$ – gdyż nie wszystkie liczby rzeczywiste są reprezentowalne z użyciem liczby zmiennoprzecinkowej (o skończonej liczbie bitów).

W przypadku liczb zmiennoprzecinkowych do utraty cyfr znaczących dochodzi, gdy szerokość mantysy nie wystarcza do przechowania wszystkich tych cyfr. Przykładowo, wynikiem sumy `1.0e100 + 1.0` będzie `1.0e100`, gdyż reprezentowanie wartości większej o 1 wymagałoby mantysy pozwalającej na reprezentację 100 miejsc po przecinku, natomiast 64-bitowy typ **`double`** udostępnia jedynie 52 bity mantysy, czyli jej maksymalna dokładność to $2^{-52} \approx 2,22 \times 10^{-16}$. Jak widzisz, typy zmiennoprzecinkowe nie pozwalają na reprezentację wszystkich liczb rzeczywistych, gdyż w dowolnym przedziale $[a, b]$ istnieje nieskończenie wiele liczb rzeczywistych.

Zwróć też uwagę, że w każdym systemie liczbowym pewnych ułamków zwykłych nie da się wyrazić z użyciem skończonej liczby pozycji, przykładowo:

- w systemie dziesiętnym: $1/3_{10} = 0.333\ldots_{10}$
- w systemie dwójkowym: $2/5_{10} = 0.4_{10} = 0.011001100110011\ldots_{2}$

W przypadku ułamków dziesiętnych posiadających w zapisie binarnym nieskończenie wiele cyfr po przecinku dojdzie do obcięcia. Aby się o tym przekonać, wykonaj poniższy program nadający zmiennej zmiennoprzecinkowej wartość początkową 0, a następnie dodający do niej dziesięć razy wartość 0.1_{10}:

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

int main(void) {
    double x = 0.0;
    for (int i = 0; i < 10; ++i) {
        x += 0.1;
    }

    printf("x = %.20lf\n", x);

    return EXIT_SUCCESS;
}
```

Wynik działania programu:

[8] Taka operacja nazwa się po angielsku *truncation toward zero*.

```
x = 0.99999999999999989000
```

Jak widzisz, choć oczekiwany wynik działania 0.1×10 to 1, wartość `x` wynosi mniej – obcięcie doprowadziło do wystąpienia tzw. **błędu zaokrąglenia** (ang. rounding error) zgodnie z definicją wynoszącego $1 - x$.

3.14.4 Dzielenie przez zero

W języku C obowiązują zasady dzielenia przez zero określone przez standard IEC 60559:1989 (x to wartość zmiennoprzecinkowa dodatnia):

- `x / 0.0` ma wartość `+INFINITY`
- `0.0 / 0.0` ma wartość `NAN`
- `-x / 0.0` ma wartość `-INFINITY`

Rozdział 4

Obiekty, zmienne i stałe

4.1 Czym są obiekty?

Język C pozwala na tworzenie i usuwanie obiektów, a także na dostęp do nich i manipulowanie ich wartością. W języku C **obiekt** (ang. object) to obszar w pamięci[1] w środowisku uruchomieniowym programu, którego zawartość może reprezentować wartości – przez *wartość* należy rozumieć znaczenie zawartości obiektu (tj. bajtów pamięci), gdy zawartość ta jest interpretowana jako posiadająca pewien ustalony typ. Obiekty są tworzone m.in. poprzez ich zdefiniowanie oraz poprzez użycie literału.

Każdy obiekt posiada m.in. poniższe cechy:

- rozmiar (który można odczytać z użyciem operatora `sizeof`)
- okres przechowywania
- typ efektywny
- wartość (która może być nieokreślona)
- (opcjonalnie) identyfikator odwołujący się do tego obiektu

Wybrane cechy obiektów będą omawiane w kolejnych rozdziałach.

Większość obiektów zajmuje ciągły obszar pamięci[2] o rozmiarze co najmniej jednego bajta. Interpretując ten obszar jako ciąg wartości typu `unsigned char` otrzymuje się **reprezentację obiektu** (ang. object representation). Jeśli dwa obiekty posiadają tą samą reprezentację, traktowane są jako równe[3]. Odwrotna zależność nie jest prawdziwa.

W przypadku obiektów typów całkowitych zajmujących więcej niż jeden bajt (np. `int`) język C nie prezycuje, który bajt przechowuje pozycje najbardziej znaczące – zależy to od architektury danego procesora. Dwie najpopularniejsze **kolejności bajtów** (ang. endianness) to:

- big-endian – najbardziej znaczący bajt jest przechowywany pod najniższym adresem w ramach obszaru zajmowanego przez obiekt typu całkowitego (m.in. architektury rodziny POWER, Sparc i Itanium)
- little-endian – najmniej znaczący bajt jest przechowywany pod najniższym adresem w ramach obszaru zajmowanego przez obiekt typu całkowitego (m.in. architektury rodziny x86 i x86_64)

4.2 Czym są zmienne?

Pamięć komputera składa się z komórek, z których każda posiada unikalny adres. Aby zaoszczędzić programistom konieczności pamiętania konkretnych fizycznych adresów używanych w programie obiektów (np. `0x146A3F67`), język C udostępnia tzw. **zmienne** (ang. variables) – przyjazne dla człowieka identyfikatory tych miejsc w pamięci, w których znajdują się utworzone w programie obiekty. Zmienna to pewien fragment pamięci o ustalonym rozmiarze, który posiada własny identyfikator (nazwę) oraz może przechowywać pewną wartość, zależną od typu zmiennej.

Oto analogia do życia codziennego: Możesz powiedzieć znajomemu z AGH „spotkajmy się w budynku o adresie al. Mickiewicza 30”, lecz wygodniej jest powiedzieć „spotkajmy się w budynku A–0”; w tym przypadku „A–0” pełni rolę identyfikatora.

[1] Nie chodzi tu jedynie o pamięć operacyjną RAM, ale o *data storage* – a więc również m.in. o rejestry procesora itp.

[2] Wyjątek stanowią pola bitowe (zob. rozdz. *19.2 Pola bitowe*).

[3] Wyjątek stanowią obiekty reprezentujące wartość zmiennoprzecinkową *NaN*.

4.3 Identyfikatory

Identyfikator (ang. identifier) to ciąg znaków używany jako nazwa np. obiektów, funkcji itd.
Oto zasady tworzenia prawidłowych identyfikatorów:

- Identyfikator może składać się wyłącznie z liter alfabetu łacińskiego (dużych i małych)[4], cyfr i znaków podkreślenia (_).
- Identyfikator nie może zaczynać się od cyfry.
- Identyfikator nie może zaczynać się od kombinacji dwóch znaków podkreślenia lub znaku podkreślenia i dużej litery (te kombinacje są zarezerwowane dla poprawnego działania bibliotek standardowych).
- Identyfikatorem nie może być ani słowo kluczowe, ani nazwa funkcji z bibliotek standardowych (w tym nazwa zastrzeżona przez bibliotekę standardową do użytku w przyszłości[5]), ani tzw. zewnętrzny identyfikator (wprowadzony np. przez Twój kompilator) zaczynający się symbolem podkreślenia.
- Identyfikator może mieć dowolną długość[6].

Ważne

Pamiętaj, że:

- W języku C wielkość liter użytych w identyfikatorze ma znaczenie!
- Zastosowanie identyfikatora kolidującego z identyfikatorami stosowanymi przez biblioteki standardowe może prowadzić do niezdefiniowanego zachowania programu.

Dobre praktyki

Identyfikator danej zmiennej powinien określać to, co ta zmienna reprezentuje.

Przykładowo, zmienna reprezentująca aktualną temperaturę (mierzoną w stopniach Celsiusza) mogłaby się nazywać `current_temperature_celcius` (z kolei nadanie jej identyfikatora `tc` będzie zwykle mało czytelne. . .).

4.4 Deklaracja a definicja

Deklaracja (ang. declaration) to konstrukcja językowa, która jedynie *wprowadza identyfikator*, czyli informuje kompilator o występowaniu danego identyfikatora oraz precyzuje jego znaczenie (typ danych) i właściwości (np. klasę zmiennej). W szczególności w przypadku zmiennych deklaracja nie powoduje przydzielenia miejsca w pamięci, niezbędnego do przechowania obiektu.

Definicja (ang. definition) to taka deklaracja, która udostępnia komplet informacji o deklarowanym identyfikatorze, przykładowo:

- w przypadku obiektów – powoduje przydzielenie miejsca w pamięci na przechowywanie obiektu
- w przypadku funkcji – zawiera ciało (tj. implementację) funkcji

Sposoby deklarowania i definiowania poszczególnych rodzajów elementów w języku C (zmiennych, funkcji, stałych, struktur itd.) a także ich możliwe własności, będą przedstawiane w kolejnych rozdziałach niniejszego skryptu.

4.5 Definicja i inicjalizacja zmiennej

Zmienne, z którymi będziesz się stykać w kilku pierwszych rozdziałach niniejszego skryptu będą wprowadzane z użyciem stosownych *definicji*. Definicja pojedynczej zmiennej – czyli definicja *indywidualna* – składa się kolejno z typu danych, identyfikatora zmiennej i średnika na końcu; jej ogólna postać to:

```
typ_danych identyfikator_zmiennej;
```

Aby zdefiniować więcej niż jedną zmienną tego samego typu – czyli dokonać definicji *grupowej* – można użyć albo kilku definicji (po jednej dla każdej zmiennej), albo jednej instrukcji zawierającej kilka identyfikatorów zmiennych rozdzielonych przecinkami. Obydwie poniższe deklaracje są więc prawidłowe:

[4] Standard C99 dopuścił możliwość stosowania również znaków Unicode, jednak do dobrych praktyk należy ograniczenie się do alfabetu łacińskiego.

[5] Wykaz takich nazw zastrzeżonych znajdziesz tu.

[6] w praktyce: do 63 znaków

```
int t;      // definicja indywidualna
int x, y;   // definicja grupowa (dwie zmienne typu `int`)
```

Dobre praktyki

W standardzie C90 pominięcie typu było traktowane jako użycie typu domyślnego – czyli typu `int`[7]. *Obecnie zaleca się każdorazowe jawne określanie typu danych.*

Efektem zdefiniowania zmiennej jest zawsze zarezerwowanie dla niej odpowiedniego bloku pamięci, ale niekoniecznie nadanie jej wartości początkowej[8]. Nadanie zmiennej *ustalonej* wartości początkowej nosi nazwę **inicjalizacji** zmiennej (ang. initialization). W przypadku zmiennych, z którymi będziesz się stykać w pierwszych rozdziałach niniejszego skryptu[9], ich zdefiniowanie nie pociąga za sobą automatycznej inicjalizacji, a przynajmniej język C tego nie wymusza. W związku z tym wartość początkowa tych zmiennych (w przypadku niektórych kompilatorów) może być przypadkowa – będzie wynikała z zawartości bloku pamięci zmiennej w momencie jego przydzielania.

Do przypisania wartości służy operator przypisania = (w języku C symbol = *nie oznacza* równości w sensie matematycznym).

Dobre praktyki

Dobrą praktyką programistyczną jest inicjalizowanie każdej zmiennej w momencie jej definiowania:

```
/* przykłady definiowania zmiennych wraz z ich inicjalizacją */
int t = 0;  // inicjalizacja z użyciem literału
int q = t;  // inicjalizacja z użyciem innego obiektu
```

Dzięki temu od początku swego istnienia ma ona przypisaną pewną wartość (wybraną przez programistę), która nie będzie się zmieniać pomiędzy kolejnymi uruchomieniami programu. Wybór właściwej wartości początkowej zależy od konkretnego problemu.

Dobre praktyki

Nie należy stosować grupowej inicjalizacji (np. `int i = 1, j, k = 2;`*), gdyż jest ona mało czytelna i łatwo prowadzi do błędów! W przytoczonym przykładzie zmienna* `j` *nie została zainicjalizowana.*

Pamiętaj, aby inicjalizując zmienną z użyciem literału korzystać z literału odpowiedniego typu – typ literału powinien być zgodny z typem inicjalizowanej zmiennej, przykładowo:

```
1 int x = 3;     // OK
2 int y = 3.0;   // ZŁA PRAKTYKA: 3.0 to literał typu `double`
3 unsigned int p = 1U;   // OK
4 unsigned int q = 1;    // ZŁA PRAKTYKA: 1 to literał typu `signed int`
```

Kompilatory C dopuszczają formę inicjalizacji przedstawioną w linii 2, ale mogą na nią narzekać, szczególnie jeśli włączona jest wysoka czułość na błędy – lepiej zatem od razu wyrobić sobie dobre nawyki.

4.6 Definiowanie zmiennych a błędy kompilacji

Ponieważ kompilator analizuje kod jednostki translacji tylko raz (w kolejności od pierwszego do ostatniego wiersza) zmienna musi zostać zadeklarowana przed pierwszym użyciem jej identyfikatora – próba skompilowania poniższego programu:

```
#include <stdlib.h>

int main(void) {
    x = 1;  // brak deklaracji symbolu `x`
```

[7] Uzasadnienie takiego podejścia znajdziesz w dokumencie JTC1/SC22/WG14 N661

[8] O tym, kiedy te dwie operacje są dokonywane automatycznie jedna po drugiej przeczytasz w rozdziale *13 Klasy zmiennych*.

[9] czyli tzw. zmiennych automatycznych

```
    return EXIT_SUCCESS;
}
```

spowoduje wystąpienie błędu „`error: 'x' undeclared`”.

Co więcej, w danej jednostce translacji może znajdować się tylko jedna definicja danego symbolu – próba skompilowania poniższego kodu:

```
#include <stdlib.h>

int main(void) {
  // Dwie definicje tej samej zmiennej, ale bez ich inicjalizacji.
    int x;
    int x;  // ponowna definicja

    return EXIT_SUCCESS;
}
```

spowoduje wystąpienie błędu „`error: redeclaration of 'x' with no linkage`”.
Z kolei próba skompilowania poniższego kodu:

```
#include <stdlib.h>

int main(void) {
  // Dwie definicje tej samej zmiennej, z ich inicjalizacją.
    int x = 1;
    int x = 1;  // ponowna definicja

    return EXIT_SUCCESS;
}
```

spowoduje wystąpienie błędu „`error: redefinition of 'x'`”.

4.7 Stałe

Niektóre wartości wykorzystywane w programie są z góry ustalone jeszcze przed uruchomieniem programu i pozostają niezmienione przez cały okres jego działania – są to **stałe** (ang. constants). Przykładem takiej stałej wartości może być wartość liczby π.

Język C udostępnia cztery sposoby na definiowanie stałych:

- literały – np. `1`, `'A'`, `3.5`
- dodanie kwalifikatora `const` do typu[10] – np. `const int`
- zdefiniowanie stałej symbolicznej z użyciem dyrektywy preprocesora `#define`[11]
- zdefiniowanie etykiet typu wyliczeniowego[12]

Korzystanie ze zmiennych o typie z kwalifikatorem `const` w zasadzie nie różni się od korzystania ze zmiennych o typie bez tego kwalifikatora – obecność kwalifikatora `const` uniemożliwia jedynie modyfikację wartości takiej zmiennej. W przypadku większości stałych wartości, z którymi się na razie będziesz stykać, najlepiej reprezentować je z użyciem literałów lub zmiennych z kwalifikatorem `const`.

Użycie zmiennych o typie z kwalifikatorem `const` pozwala na parametryzację kodu i unikanie tzw. **stałych magicznych** (ang. magic constants) lub inaczej **liczb magicznych** (ang. magic numbers) – czyli użycia nieopisanych wartości (głównie liczbowych) w kodzie programu. Przykładowo, w poniższym programie

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(void) {
5     double r = 2.0;
6     double area = 3.141592 * r * r; // korzysta ze stałej magicznej `3.141592`
7
8     printf("Pole kola o promieniu %.2lf wynosi %.2lf\n", r, area);
```

[10] Kwalifikator `const` został omówiony w rozdz. *13.7.1 Kwalifikator `const`*.
[11] Działanie preprocesora i dyrektywy `#define` zostało omówione w rozdz. *5 Preprocesor języka C*.
[12] Typ wyliczeniowy został omówiony w rozdz. *18.2 Typy wyliczeniowe*.

```
9
10      return EXIT_SUCCESS;
11  }
```

wartość liczby π została umieszczona jako stała magiczna. Wielokrotne użycie takiej stałej magicznej może powodować problemy, gdy np. postanowisz zmienić dokładność (liczbę miejsc po przecinku) i zapomnisz tego zrobić we wszystkich miejscach, gdzie użyłeś takiej stałej magicznej. Instrukcja w linii 6 może zostać zastąpiona poniższymi dwiema instrukcjami:

```
const double PI = 3.141592;  // definicja stałej PI
double area = PI * r * r;
```

Każdorazowo wynik uruchomienia programu to:

```
Pole kola o promieniu 2.00 wynosi 12.57
```

Rozdział 5

Preprocesor języka C

W tym rozdziale poznasz lepiej możliwości preprocesora języka C, a w szczególności funkcje-makra i kompilację warunkową.

5.1 Czym jest preprocesor?

Preprocesor (ang. preprocessor) to specjalny program dokonujący zmian *tekstu* kodu programu jeszcze przed faktyczną kompilacją programu (stąd przedrostek „pre"). Preprocesor wyszukuje w kodzie specjalnych instrukcji nazywanych **dyrektywami preprocesora** (ang. preprocessor directives), które informują go o konieczności dokonania zmian (np. dołączenia pliku nagłówkowego, kompilacji warunkowej itp.) – wszystkie dyrektywy preprocesora języka C zaczynają się symbolem `#`. Ponieważ preprocesor dokonuje jedynie transformacji leksykalnych, efektem jego działania jest wciąż rodzaj tekstu[1].

5.2 Dołączanie plików: `#include`

Gdy preprocesor zauważy dyrektywę `#include`, odnajduje on plik o podanej nazwie i zastępuje wystąpienie danej dyrektywy `#include` zawartością tego pliku, co można schematycznie zilustrować w poniższy sposób:

```
// Plik A
Ala ma kota.
```

```
// Plik B
#include A
Kot ma Alę.
```

Po zakończeniu działania preprocesora kod pliku `B` będzie wyglądał następująco:

```
// Plik B
// Plik A
Ala ma kota.
Kot ma Alę.
```

Istnieją dwa rodzaje dyrektywy `#include`:

```
#include <standardowy_plik_naglowkowy.h>  // nazwa pliku w nawiasach ostrych
#include "moj_plik_naglowkowy.h"  // nazwa pliku w cudzysłowie
```

Użycie nawiasów ostrych sprawia, że preprocesor poszukuje pliku w jednym lub kilku standardowych katalogach systemowych. Z kolei cudzysłowy sprawiają, że przeglądany jest najpierw katalog bieżący (lub inny katalog podany wraz z nazwą pliku nagłówkowego), a dopiero później katalogi standardowe:

```
#include <stdio.h>  // przeszukuje katalogi systemowe
#include "abc.h"    // przeszukuje najpierw bieżący katalog roboczy
#include "/usr/abc.h"  // przeszukuje najpierw katalog /usr/
```

Po co dołączamy inne pliki (nagłówkowe)? Ponieważ zawierają one informacje, których potrzebuje kompilator – przykładowo plik `stdio.h` zawiera definicję funkcji `printf()`. Należy unikać dołączania zbędnych

[1] ściślej: ciąg tzw. żetonów

plików nagłówkowych, gdyż wydłuża to czas kompilacji. Natomiast dołączenie dużego pliku nagłówkowego nie musi powodować dużego wzrostu objętości programu – zawartość plików nagłówkowych składa się bowiem głównie z informacji wykorzystywanych w procesie kompilacji, a nie z elementów dodawanych do kodu wykonywalnego.

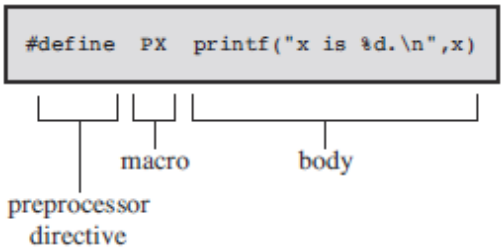
5.3 Stałe symboliczne: `#define`

Każdy wiersz (logiczny)[2] dyrektywy **`#define`** składa się z trzech części (zob. rys. 5.1):

- Pierwszą częścią jest sama dyrektywa **`#define`**.
- Drugą częścią jest wybrany przez nas skrót, znany w świecie komputerów jako makro lub zamiennik (alias). Nazwa skrótu nie może zawierać odstępów, a także musi być zgodna z zasadami nazewnictwa zmiennych – dozwolone są więc tylko litery (zwyczajowo: duże litery), cyfry i znak podkreślenia (_), przy czym pierwszy znak nazwy nie może być cyfrą.
- Trzecia część nosi nazwę **treści** (ang. body). Gdy preprocesor znajdzie jeden ze zdefiniowanych skrótów w kodzie źródłowym, niemal zawsze zastępuje go właśnie treścią (istnieje jeden wyjątek od tej reguły, który wkrótce poznasz).

Przykładowo, stałą symboliczną jest makro `EXIT_SUCCESS` zdefiniowane w pliku nagłówkowym `stdlib.h` jako

```
#define EXIT_SUCCESS 0
```



Rysunek 5.1. Części definicji makra

Proces przechodzenia od makra do jego ostatecznego rozwinięcia nosi nazwę **rozwijania makra** (ang. macro expansion). Zauważ, że wiersz **`#define`** może zawierać zwykłe komentarze języka C – są one po prostu ignorowane przez preprocesor. Ponadto, definicję makra można przenieść do następnego wiersza za pomocą lewego ukośnika (\). Dyrektywy **`#define`** umieszcza się na początku kodu programu, zaraz pod dyrektywami **`#include`**; każda dyrektywa **`#define`** powinna być umieszczona w osobnym wierszu (logicznym). Wszystko to pokazuje listing 5.1.

Listing 5.1. Proste przykłady użycia preprocesora

```
/* preproc.c - proste przykłady użycia preprocesora */
#include <stdio.h>
#include <stdlib.h>

#define DWA 2  // można korzystać z komentarzy
#define OW "Oto bardzo dlugi komunikat rozbity na " \
"dwa wiersze"   // lewy ukośnik kontynuuje definicję w następnym wierszu
#define CZTERY DWA*DWA
#define PX printf("X wynosi %d.\n", x)
#define FMT "X wynosi %d.\n"

int main(void) {
    int x = DWA;
    PX;
```

[2] Dyrektywy preprocesora obejmują obszar od symbolu # do najbliższego znaku końca linii. Długość dyrektywy jest więc ograniczona do jednego wiersza. Mimo to, kombinacja „lewy ukośnik i znak nowej linii" jest traktowana jako odstęp, a nie jako znacznik końca wiersza, zatem jedna dyrektywa może zajmować obszar kilku wierszy fizycznych. Z punktu widzenia preprocesora obszar ten stanowi jednak jeden wiersz logiczny.

```
    x = CZTERY;
    printf(FMT, x);
    printf("%s\n", OW);
    printf("DWA: OW\n");
    return EXIT_SUCCESS;
}
```

Wynik działania programu:

```
X wynosi 2.
X wynosi 4.
Oto bardzo dlugi komunikat rozbity na dwa wiersze
DWA: OW
```

Jak wygląda scenariusz zdarzeń, który doprowadził do takiego wyniku?

Instrukcja

```
int x = DWA;
```

zostaje zamieniona na

```
int x = 2;
```

Następny wiersz programu przynosi kolejną nowość. Być może wydaje Ci się, że stała `CZTERY` zostaje zastąpiona liczbą 4, ale w rzeczywistości proces przebiega następująco:

```
x = CZTERY;
```

zostaje zamienione na

```
X = DWA*DWA;
```

a następnie na

```
x = 2*2;
```

W tym miejscu kończy się proces rozwijania makra. Mnożenie zostaje wykonane nie w czasie pracy preprocesora, ale w momencie kompilacji, ponieważ wówczas obliczane są wszystkie wyrażenia stałe (wyrażenia złożone wyłącznie ze stałych) w języku C. Preprocesor nie wykonuje żadnych obliczeń, a jedynie dokonuje podstawień w *tekście* kodu programu, traktując dyrektywy w bardzo dosłowny sposób.

Zwróć uwagę, że definicja makra może składać się z innych makr.

Dobre praktyki

W praktyce stałe symboliczne będą Ci przydatne jedynie podczas pracy z tablicami – zwłaszcza tablicami wielowymiarowymi – do „elastycznego" określania ich rozmiaru (w tym przypadku: aby rozmiar tablicy był określony z użyciem identycznej wartości w całym programie). W pozostałych przypadkach, gdy potrzebujesz korzystać ze stałych w ujęciu nauk ścisłych (np. wartość liczby π, stała grawitacji G itp.) lepiej jest definiować odpowiednie „zwykłe" stałe – z użyciem kwalifikatora **`const`**.

5.3.1 Żetony

Z technicznego punktu widzenia, preprocesor postrzega makro nie jako ciąg znaków, a jako ciąg tzw. **żetonów** (ang. tokens). Żetony są „słowami", z których składa się treść definicji makra. Są one rozdzielone znakami niedrukowanymi. Na przykład, definicja

```
#define CZTERY 2*2
```

zawiera jeden żeton (ciąg `2 * 2`), za to definicja

```
#define SZESC 2 * 3
```

składa się z trzech żetonów: `2`, `*` oraz `3`.

5.3.2 Przedefiniowanie stałych symbolicznych

Raz zdefiniowanych stałych symbolicznych nie można wprost przedefiniować:

```
#define X 1
#define X 2  // BŁĄD: próba przedefiniowania stałej
```

Standard języka C dopuszcza ponowną definicję jedynie w przypadku, jeśli nowa definicja jest identyczna ze starą[3], natomiast inne definicje traktuje jako błędne.

5.3.3 Dyrektywa `#define` i argumenty

W języku C możliwe jest tworzenie makr, które wyglądają i działają podobnie do funkcji. Listing 5.2 ilustruje sposób definiowania i wykorzystywania takich funkcji-makr. Wskazuje on również na potencjalne pułapki.

Listing 5.2. Makra z argumentami

```
/* mak_arg.c - makra z argumentami */
#include <stdio.h>
#include <stdlib.h>

#define KWADR(X) X*X
#define PR(X) printf("Wynik wynosi %d.\n", X)

int main(void) {
    int x = 4;
    int z;
    z = KWADR(x);
    PR(z);
    z = KWADR(2);
    PR(z);
    PR(KWADR(x + 2));
    PR(100 / KWADR(2));
    printf("x wynosi %d.\n", x);
    PR(KWADR(++x));
    printf("Po zwiekszeniu x wynosi %x.\n", x);
    return EXIT_SUCCESS;
}
```

Wszystkie wystąpienia nazwy `KWADR(x)` w listingu 5.2 zostają zastąpione wyrażeniem `x*x`. W odróżnieniu od poprzednich przykładów możliwe jest korzystanie z symboli innych niż `x` – znak `x` w definicji makra zostaje bowiem zastąpiony symbolem użytym w wywołaniu makra w kodzie programu. Na przykład, `KWADR(2)` zostaje zastąpione wyrażeniem `2 * 2`, zatem `x` pełni rolę argumentu. Argument makra nie działa jednak dokładnie tak samo, jak argument funkcji – oto efekt uruchomienia programu:

```
Wynik wynosi 16.
Wynik wynosi 4.
Wynik wynosi 14.
Wynik wynosi 100.
x wynosi 4.
Wynik wynosi 30.
Po zwiekszeniu x wynosi 6.
```

Pierwsze dwa wiersze są zgodnie z przewidywaniami, ale kolejne zawierają dość osobliwe (na pierwszy rzut oka) wyniki. Skąd się one biorą?

Ponieważ preprocesor nie dokonuje obliczeń, a jedynie podstawia łańcuchy, poniższe użycie makra

```
KWADR(x + 2)
```

zostaje rozwinięte w

[3] Identyczność definicji oznacza, że ich treści muszą składać się z tych samych żetonów podanych w tej samej kolejności.

```
x + 2*x + 2
```

przy czym wartość tego wyrażenia dla $x = 4$ wynosi $4+2\times4+2 = 14$. Z kolei wyrażenie `100 / KWADR(2)` zostaje rozwinięte w `100 / 2 * 2`, przy czym wartość tego wyrażenia wynosi $100/2 \times 2 = 50 \times 2 = 100$.

Przykład ten wskazuje na istotną różnicę między wywołaniem funkcji a wywołaniem makra:

- Wywołanie *funkcji* przekazuje wartość argumentu do funkcji w trakcie działania programu.
- Wywołanie *makra* podstawia argument przed kompilacją.

Powyższe procesy przebiegają w różnym momencie cyklu budowania programu.

Aby uniknąć podobnych niespodzianek do dobrej praktyki należy otaczanie nawiasami zarówno *każdego* użycia nazwy argumentu w makrze, jak i całego makra:

```
#define KWADR(X) ((X)*(X))  // poprawne makro
```

Jednak nawet poprawne stosowanie nawiasów nie pomaga w przypadku korzystania z operatorów pre-inkrementacji i predekrementacji: `KWADR(++x)` zostaje rozwinięte w `((++x)*(++x))`, zatem zmienna `x` zostanie zwiększona dwukrotnie. Zauważ, że wyrażenie `++x` działałoby prawidłowo jako argument funkcji, ponieważ funkcja otrzymałaby tylko wartość tego wyrażenia (tj. 5), obliczoną przed jej wywołaniem.

Między nazwą makra a listą parametrów nie może występować odstęp:

```
#define KWADR (X) ((X)*(X))  // BŁĄD: pierwsze `(X)` jest traktowane jako
                             //   token a nie lista parametrów
```

5.3.4 Argumenty makr w łańcuchach

Aby uzyskać informację o argumencie makra (czyli tym, co zostało przekazane w momencie jego wywołania) w postaci tekstowej, należy skorzystać z symbolu `#` (zob. listing 5.3).

Listing 5.3. Stringification

```
/* stringify.c - token stringification */
#include <stdio.h>
#include <stdlib.h>

#define PKW(x) printf("Kwadratem liczby `%s` jest %d.\n", #x, ((x)* (x)))

int main(void) {
    int y = 5;
    PKW(y);
    PKW (2 + 4);
    return EXIT_SUCCESS;
}
```

Oto dane wyjściowe:

```
Kwadratem liczby `y` jest 25.
Kwadratem liczby `2 + 4` jest 36.
```

Powyższy mechanizm jest przydatny podczas tworzenia makr pod kątem debugowania.

5.3.5 Makro czy funkcja?

Wiele zadań może zostać wykonanych zarówno przy pomocy makra, jak i funkcji. Którego z tych mechanizmów należy użyć? Nie istnieje w tej kwestii żadna żelazna zasada.

Z pewnością makra są trudniejsze w użyciu niż funkcje, ponieważ mogą dawać nieoczekiwane i niewidoczne (na pierwszy rzut oka) wyniki i efekty uboczne. Z drugiej strony makra nie interesują typy przekazywanych argumentów (gdyż operują na tekście kodu programu), zatem można za ich pomocą definiować np. przydatne proste funkcje matematyczne działające zarówno dla typów całkowitych, jak i zmiennoprzecinkowych:

```
#define MAX(X,Y) ((X) > (Y) ? (X) : (Y))
#define ABS(X) ((X) < 0 ? -(X) : (X))
```

5.4 Dyrektywa #undef

Dyrektywa `#undef IDENTIFIER` usuwa stałą symboliczną `IDENTIFIER`:

```
#define X 1
#undef X      // usunięcie stałej symbolicznej
#define X 2   // OK: ponowne zdefiniowanie stałej symbolicznej
```

Użycie dyrektywy `#undef IDENTIFIER` jest dozwolone, nawet jeśli stała `IDENTIFIER` nie była wcześniej zdefiniowana.

5.5 Kompilacja warunkowa

Dyrektywy `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` oraz `#endif` służą zwykle do tworzenia programów, które mogą być kompilowane na kilka sposobów, czyli pozwalają uzyskać tzw. **kompilację warunkową** (polega ona na akceptowaniu lub pomijaniu przez kompilator fragmentów kodu w zależności od warunków w momencie kompilacji).

Przykłady w kolejnych rozdziałach zakładają właśnie taki scenariusz – konieczność umieszczenia w programie kodu zależnego od docelowego systemu operacyjnego – przy czym stałe symboliczne `_WIN64` i `__ANDROID__` są definiowane automatycznie przez kompilator (w zależności od systemu operacyjnego, na którym przebiega kompilacja).

5.5.1 Dyrektywy #ifdef, #else i #endif

Zasadę działania kompilacji warunkowej wyjaśni poniższy przykład:

```
#ifdef _WIN64
    // 64-bit Windows code
#else
    // code for other platforms
#endif
```

Dyrektywa `#ifdef` nakazuje – w przypadku, jeśli następująca po niej nazwa została zdefiniowana – wykonanie wszystkich dyrektyw i skompilowanie całego kodu aż do najbliższej dyrektywy `#else` lub `#endif`. Dyrektywa `#else` (jeśli jest obecna) oznacza początek kodu, który zostaje wykonany, jeśli wspomniana nazwa nie została zdefiniowana.

Nawiasem mówiąc, „pusta” definicja, taka jak:

```
#define ABC
```

wystarcza, aby dyrektywa `#ifdef` uznała nazwę `ABC` za zdefiniowaną.

Struktura `#ifdef`–`#else` przypomina budową instrukcję `if`–`else`. Główna różnica polega na tym, że preprocesor nie rozpoznaje klamer, a więc granice bloków oznaczane są słowami dyrektywami `#else` i `#endif`. Dyrektywy kompilacji warunkowej mogą być zagnieżdżane oraz mogą występować w dowolnym miejscu kodu programu.

5.5.2 Dyrektywa #ifndef

Dyrektywa `#ifndef` powoduje wykonanie określonego fragmentu kodu, jeśli następujący po niej identyfikator *nie został* zdefiniowany – stanowi więc przeciwieństwo dyrektywy `#ifdef` (natomiast podobnie jak `#ifdef` może być stosowana w połączeniu z dyrektywą `#else` oraz cały blok musi kończyć się dyrektywą `#endif`).

Jest ona często wykorzystywana do zdefiniowania stałej w przypadku, jeśli nie została ona zdefiniowana już wcześniej (co wykorzystuję się np. do definiowania tzw. strażnika nagłówka[4]).

```
#ifndef MY_HEADER_H_
#define MY_HEADER_H_
#endif
```

[4] zob. rozdz. *14.2 Strażnik nagłówka (header guard)*

5.5.3 Dyrektywy `#if` i `#elif`

Dyrektywa `#if` przypomina zwykłą instrukcję `if` – następuje po niej stałe wyrażenie całkowite (jego wartość niezerowa jest uznawana za prawdę), które może zawierać operatory logiczne i relacyjne języka C. Może ona zawierać także dyrektywy `#elif` oraz `#else` odpowiadające odpowiednio członom `else if` i `else` w przypadku instrukcji warunkowej.

Jeśli chcesz dokonać sprawdzania, czy dana stała została zdefiniowana, w ramach większego wyrażenia, zamiast dyrektywy `#ifdef X` możesz użyć polecenia

```
#if defined(X)  // sprawdź, czy stała symboliczna `X` została zdefiniowana
```

Operator `defined` jest operatorem preprocesora zwracającym wartość 1, jeśli jego argument jest zdefiniowany, a wartość 0 w przeciwnym przypadku.

Przykładowo, jeśli potrzebujesz umieścić w programie kod zależny od docelowego systemu operacyjnego, możesz to zrobić w poniższy sposób (makra `_WIN64` i `__ANDROID__` są definiowane przez kompilator):

```
#if defined _WIN64
    // 64-bit Windows code
#elif defined __ANDROID__
    // Android code
#else
    // code for other platforms
#endif
```

Dzięki wykorzystaniu kompilacji warunkowej możesz niewielkim wysiłkiem uczynić swój program bardziej przenośnym.

Rozdział 6

Formatowane wyjście: funkcja `printf()`

W tym rozdziale nauczysz się, jak wyświetlać dane przy pomocy funkcji `printf()`.

6.1 Funkcja `printf()`: ogólna postać

Główną funkcją służącą do wypisywania danych na standardowe wyjście (zazwyczaj oznaczające wyświetlanie danych na ekranie terminala) jest funkcja `printf()` o poniższym prototypie:

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

przy czym zgodnie ze standardem języka C zapis `...` oznacza, że funkcja może przyjmować tzw. zmienną liczbę argumentów[1].

Pierwszym parametrem jest tzw. **łańcuch sterujący** (ang. format string), który określa sposób wyświetlenia elementów. W najbardziej podstawowym przypadku służy on do wyświetlania tekstu, z uwzględnieniem sekwencji sterujących [2], przykładowo instrukcja:

```
printf("Ala ma kota!\n");
```

spowoduje wypisanie na standardowe wyjście ciąg znaków

```
Ala ma kota!
```

(wraz z przejściem do nowego wiersza).

Chcąc wyświetlić (odpowiednio sformatowane) dane należy zawrzeć w łańcuchu sterującym tzw. **specyfikatory konwersji** (ang. format specifiers), w miejsce których zostaną podstawione wartości, przy czym każdemu specyfikatorowi konwersji powinna odpowiadać dokładnie jedna podstawiana wartość przekazana jako dalszy argument wywołania funkcji `printf()`. Przykładowo, instrukcje:

```
const double x = 3.24;
printf("Czesc calkowita liczby %f to %d.\n", x, (int) x);
```

spowodują wypisanie na standardowe wyjście poniższego komunikatu (łącznie z przejściem do nowego wiersza):

```
Czesc calkowita liczby 3.240000 to 3.
```

Specyfikatory konwersji zostały omówione w kolejnym rozdziale.

6.2 Specyfikatory konwersji

Funkcja `printf()` *interpretuje bajty* każdego z przekazanych argumentów zgodnie z odpowiadającym mu specyfikatorem konwersji w łańcuchu sterującym. Wybrane najczęściej używane specyfikatory konwersji przedstawia tabela 6.1.

Mimo istnienia specyfikatorów konwersji dla wartości zmiennoprzecinkowych typu `double` i `long double`, w języku C nie ma specyfikatora dla typu `float`. Powodem tego jest fakt, iż w K&R C

[1] zob. variadic functions

[2] zob. rozdz. *3.9 Typy znakowe*

Tablica 6.1. Wybrane specyfikatory konwersji funkcji `printf()`

Specyfikator	Interpretacja wartości jako. . .
`%c`	pojedynczy znak
`%d`	dziesiętna liczba całkowita ze znakiem
`%e`	liczba zmiennoprzecinkowa w notacji z literą „e”
`%f`	liczba zmiennoprzecinkowa w zapisie dziesiętnym
`%o`	ósemkowa liczba całkowita bez znaku
`%u`	dziesiętna liczba całkowita bez znaku
`%x`	szesnastkowa liczba całkowita bez znaku (cyfry szesnastkowe pisane małą literą)

wartości typu **`float`** były automatycznie przetwarzane na wartości typu **`double`** w momencie użycia ich w wyrażeniu lub przekazania ich do funkcji. Choć w standardzie języka C co do zasady nie ma to miejsca, *w celu zachowania kompatybilności wstecznej* wszystkie argumenty typu **`float`** przekazywane do funkcji `printf()` są, wzorem K&R C, rozszerzane do typu **`double`**.

Ponieważ pojedynczy symbol `%` w łańcuchu sterującym oznacza – błędnie zapisany – specyfikator konwersji, do wyświetlenia znaku procentu należy użyć kombinacji `%%`.

Przykład: Wyświetlanie znaków

Ponieważ zmienna znakowa jest przechowywana jako wartość całkowita, wyświetlając zmienną typu **`char`** przy pomocy specyfikatora `%d` otrzymasz kod dziesiętny tego znaku:

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

int main(void) {
    char ch = 'C';
    printf("Kod znaku %c to %d (szesnastkowo: %#x).\n", ch, ch, ch);
    return EXIT_SUCCESS;
}
```

Wynik wykonania programu:

```
Kod znaku C to 67 (szesnastkowo: 0x43).
```

6.3 Modyfikatory specyfikatorów konwersji

Działanie podstawowych specyfikatorów konwersji można zmieniać, wstawiając między znak `%` a literę określającą format tzw. **modyfikator** (ang. modifier) lub **znacznik** (ang. flag). Tabele 6.2 i 6.3 zawierają listę standardowych symboli, jakie mogą służyć do tego celu. W przypadku stosowania kilku modyfikatorów na raz, należy je zapisać w tej samej kolejności, w jakiej przedstawiono je w tabeli 6.2, przy czym nie wszystkie kombinacje modyfikatorów są dozwolone.

Tablica 6.2. Modyfikatory funkcji `printf()`

Modyfikator	Znaczenie
znacznik	Pięć dostępnych znaczników (`-`, `+`, odstęp, `#` i `0`) jest opisane w tabeli 6.3. Można stosować kilka znaczników na raz (lub też nie stosować żadnego).
liczba	Minimalna szerokość pola. Jeśli wyświetlana wartość lub łańcuch nie zmieści się w polu o podanej szerokości, zostanie użyte większe pole.
.liczba	Dokładność. W przypadku konwersji typów `%E` i `%f` jest to liczba cyfr po przecinku.
ciąg liter	Długość typu formatowanej wartości, np. `l` – wartość długości **`long`**, `z` – wartość długości `size_t` itp.

Tablica 6.3. Znaczniki funkcji `printf()`

Znacznik	Znaczenie
–	Wyrównuje wartość do lewej krawędzi pola.
+	Wyświetla wartość ze znakiem plus lub minus, w zależności od tego, czy jest dodatnia czy ujemna.
odstęp	Wyświetla wartość z odstępem na początku (ale bez znaku plus), jeśli jest dodatnia, a ze znakiem minus, jeśli jest ujemna.
#	Wyświetla wartość w postaci alternatywnej, zależnej od specyfikatora. Poprzedza wartości typu `%o` zerem, a wartości typu `%x` / `%X` – odpowiednio symbolem `0x` lub `0X`.

W przypadku modyfikatorów określających szerokości pola i dokładność zamiast stosować z góry ustaloną wartość możesz użyć symbolu `*` i przekazać pożądaną wartość jako argument funkcji `printf()`.

Oto przykłady działania wybranych modyfikatorów i znaczników (dla zwiększenia czytelności początek i koniec każdego pola zostały oznaczone gwiazdką):

Listing 6.1. Użycie wybranych modyfikatorów i znaczników funkcji `print()`

```
#define __USE_MINGW_ANSI_STDIO 1
/* MinGW korzysta domyślnie z biblioteki "Visual C Runtime" (msvcrt)
 * firmy Microsoft, która jest zgodna jedynie z C89 - nie wspiera zatem
 * specyfikatora konwersji `z`. Powyższe makro sprawia, że MinGW korzysta
 * z alternatywnej biblioteki dostarczonej przez twórców MinGW.
*/

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

int main(void) {
    printf("*%d*\n", -123);
    printf("*% d*\n", 123);  // ` ` : miejsce na ew. znak minus
    printf("%x %#x\n", 0xABC, 0xABC);  // `#` : przedrostek formatu

    // szerokość pola
    printf("*%2d*\n", 123);
    printf("*%6d*\n", 123);
    printf("*%*d*\n", 8, 123);  // `*` : "zmienna" szerokość
    printf("*%-6d*\n", 123);  // `-` : wyrównanie do lewej

    printf("%.2f\n", 0.1);  // dokładność
    printf("%.2f\n", 0.123);  // dokładność
    printf("%.*f\n", 1, 0.123);  // `*` : "zmienna" dokładność

    // modyfikator długości
    printf("sizeof(int) = %zu bajtow\n", sizeof(int));

    return EXIT_SUCCESS;
}
```

Wynik uruchomienia powyższego programu:

```
*-123*
* 123*
abc 0xabc
*123*
*   123*
*     123*
*123   *
```

```
0.10
0.12
0.1
sizeof(int) = 4 bajtow
```

6.4 Niezgodność konwersji

Jak wspomniano, funkcja `printf()` *interpretuje bajty* każdego z przekazanych argumentów zgodnie z odpowiadającym mu specyfikatorem konwersji w łańcuchu sterującym. Przykładowo dla poniższego programu:

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

int main(void) {
    unsigned int ui = 0x1234;
    float f = 1.0f;
    unsigned int ui_max = UINT_MAX;

    printf("%f (float) = %d (%%d)\n", f, f);
    printf("%u (unsigned int) = %d (%%d)\n", ui_max, ui_max);
    printf("%#x (unsigned int) = %#hhx (%%x)\n", ui, ui);

    return EXIT_SUCCESS;
}
```

wynik jego uruchomienia wygląda następująco:

```
1.000000 (float) = 0 (%d)
4294967295 (unsigned int) = -1 (%d)
0x1234 (unsigned int) = 0x34 (%x)
```

Dlaczego?

- Jeśli do wyświetlenia wartości typu **`float`** (liczby zmiennoprzecinkowej) zostanie użyty specyfikator `%ud`, służący do wyświetlania liczby całkowitej bez znaku, wyświetlone zostaną „śmieci”, gdyż liczby te są zupełnie inaczej reprezentowane w pamięci komputera[3] – bity znaku, cechy i mantysy zostaną zinterpretowane jako kolejne pozycje „zwykłej” liczby binarnej.
- Użycie specyfikatora `%d`, odpowiadającego typowi `(signed) int`, do wyświetlenia wartości `UINT_MAX` (typu **`unsigned int`**) spowoduje wyświetlenie `-1`, gdyż liczby całkowite ze znakiem są przechowywane z użyciem dopełnienia dwójkowego[4].
- Użycie specyfikatora `%hhx`, odpowiadającego typowi **`unsigned char`**, do wyświetlenia wartości typu **`unsigned int`** spowoduje wyświetlenie jedynie pojedynczego bajtu tej wartości[5].

Funkcja `printf()` nie wymusza stosowania specyfikatorów konwersji zgodnych z typem przekazanych argumentów – ewentualne niezgodności nie zostaną również zgłoszone jako błąd przez kompilator (chyba, że użyjesz np. flagi kompilacji `-Werror=format`; domyślnie kompilator nie zgłosi nawet ostrzeżenia). Nawet jeśli kompilator zgłosi „jedynie” ostrzeżenie (czy też po prostu zauważysz niezgodność), odpowiednio:

- Jeśli brak zgodności wynikał z Twojego niedopatrzenia – popraw specyfikator konwersji.
- Jeśli celowo wypisujesz dane z użyciem danego specyfikatora konwersji – do dobrych praktyk należy dokonanie rzutowania takiego niezgodnego argumentu do typu zgodnego ze specyfikatorem konwersji.

6.5 Buforowanie

W którym momencie dane sformatowane z użyciem instrukcji `printf()` są faktycznie przekazywane na standardowe wyjście? Standardowe wyjście jest wyjściem buforowanym, co oznacza, że dane wyjściowe

[3] zob. rozdz. *3.3 Liczby binarne*

[4] zob. rozdz. *3.3.2 Liczby całkowite ze znakiem*

[5] Którego bajtu? – To zależy od kolejności przechowywania bajtów na danej konfiguracji sprzętowej (zob. rozdz. *4.1 Czym są obiekty?*).

wysyłane są do przejściowego obszaru pamięci zwanego buforem. Od czasu do czasu zawartość bufora przesyłana jest na terminal – operacja ta nazywa się **opróżnianiem bufora** (ang. buffer flush).

Standard języka C określa trzy sytuacje, w których dochodzi do automatycznego opróżnienia bufora:

- w momencie przepełnienia bufora,
- w momencie wystąpienia znaku nowej linii,
- gdy program oczekuje na dane wejściowe od użytkownika.

Aby mieć gwarancję, że instrukcja `printf()` bez zwłoki prześle dane na standardowe wyjście, dodaj na końcu łańcucha sterującego sekwencję sterującą `\n`.

Ważne

Upewnij się, że na końcu łańcucha sterującego znajduje się znak nowej linii `\n` (inaczej może się zdarzyć, że tekst do wyświetlenia utknie w buforze).

6.6 Liczba specyfikatorów konwersji a liczba argumentów

Upewnienie się, że liczba specyfikatorów formatu odpowiada liczbie podanych wartości, należy do Ciebie – ewentualne niezgodności nie zostaną domyślnie wychwycone przez kompilator (chyba, że użyjesz np. flagi kompilacji `-Werror=format`), gdyż funkcja `printf()` dopuszcza przekazywanie tzw. zmiennej liczby argumentów. Przekazanie niewłaściwej liczby argumentów do funkcji dopuszczającej ich zmienną liczbę powoduje niezdefiniowane zachowanie programu:

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

int main(void) {
    printf("Wlasciwy sposob: ");
    printf("%d - %d = %d\n", 10, 2, 10 - 2);
    printf("Bledny sposob:   ");
    printf("%d - %d = %d\n", 10); // brakuje 2 argumentów
    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
Wlasciwy sposob: 10 - 2 = 8
Bledny sposob:   10 - 2290648 = 2292769
```

6.7 Praktyczne wskazówki

Pola o stałej szerokości są użyteczne przy wyświetlaniu danych w układzie kolumnowym. Pamiętaj o pozostawieniu odstępu pomiędzy kolejnymi specyfikatorami konwersji – wówczas wyświetlane wartości nie będą zachodzić jedna na drugą, nawet gdyby przekroczyły one szerokość pola (dzieje się tak dlatego, że zwykłe znaki w łańcuchu sterującym, włącznie z odstępami, nie są nigdy pomijane).

Informacja

Jeśli Twój program nie wyświetla oczekiwanej liczby wartości lub wyświetla wyniki, których nie oczekiwałeś, sprawdź, czy przekazałeś funkcji `printf()` właściwą liczbę argumentów oraz odpowiednie specyfikatory konwersji (zgodne z typem argumentów).

Rozdział 7

Operatory, wyrażenia i instrukcje

Ten rozdział zapozna Cię z ważniejszymi operatorami dostępnymi w języku C włącznie z tymi, które odpowiadają prostym działaniom arytmetycznym. Dowiesz się o priorytetach operatorów i poznasz znaczenie terminów *instrukcja* i *wyrażenie*.

7.1 Czym jest operator?

Język C (podobnie jak większość innych języków programowania) umożliwia stosowanie zbioru **operatorów** (ang. operators) – konstrukcji, które zachowują się jak zwykłe funkcje, lecz różnią się od nich składnią lub semantyką. Przykłady popularnych (podstawowych) operatorów w języku C to m.in. operatory arytmetyczne (np. `+`), porównania (np. `<`) i logiczne (np. `&&` czyli operator koniunkcji); do bardziej złożonych operatorów należy m.in. operator przypisania (`=`).

Główne cechy opisujące operator to liczba i typy argumentów, typ wartości zwracanej, wykonywane działanie, priorytet, łączność (lub jej brak) oraz umiejscowienie operatora względem operandów. **Operand** (ang. operand) to po prostu argument operatora (przykładowo, operandami operatora `+` są składniki sumy).

W zależności od liczby operandów wymaganych przez operator wyróżniamy m.in.:

- operatory jednoargumentowe (ang. unary operators) – np. $\neg x$ (logiczna negacja)
- operatory dwuargumentowe (ang. binary operators) – np. $a + b$ (dodawanie)
- operatory trójargumentowe (ang. ternary operators) – np. operator warunkowy[1]

W zależności od umiejscowienia operatora względem operandów wymaganych przez operator wyróżniamy:

- operator przedrostkowym lub prefiksowy (ang. prefix operator) – np. $\neg x$
- operator wrostkowy lub infiksowy (ang. infix operator) – np. $a + b$
- operator przyrostkowy lub postfiksowy (ang. postfix operator) – np. `x++` (operator postinkrementacji)

7.2 Czym jest wyrażenie?

Wyrażenie (ang. expression) jest kombinacją operatorów i operandów. Każde wyrażenie w języku C ma wartość, obliczaną poprzez wykonanie wszystkich działań zgodnie z priorytetami i kierunkami wiązania operatorów (pojęcia te poznasz w rozdziałach poniżej). Najprostsze możliwe wyrażenie składa się z samego operandu, natomiast wyrażenia często składają się z mniejszych wyrażeń, zwanych **podwyrażeniami** (ang. subexpression). Kilka przykładów wyrażeń przedstawia tabela 7.1

Tablica 7.1. Przykłady wyrażeń w języku C

wyrażenie	wartość
`4`	`4`
`-6`	`-6`
`4 + 21`	`25`
`a * (b + c / d) / 20`	(zależy od wartości `a`, `b`, `c` i `d`; zawiera podwyrażenia)

[1] Operator ten poznasz w rozdziale 8.8.

7.3 Priorytet operatorów

Priorytet lub **pierwszeństwo** operatorów (ang. priority) określa, jak należy interpretować wyrażenie, w którym na jednym poziomie (struktury nawiasowej) występują dwa operatory. Priorytet odpowiada znanemu Ci z matematyki pojęciu kolejności działań – operator o wyższym priorytecie zostanie wykonany najpierw, a gdy obok siebie stoją, wówczas operacje są wykonywane od lewej do prawej (chyba, że sam operator działa inaczej, o czym za chwilę).

Przykładowo, w matematyce wyrażenie $(2 + 2 \times 3) \times 2$ zostanie obliczone w następujących krokach:

1. Obliczenie wartości w nawiasie:
   - Obliczenie $2 \times 3 = 6$
   - Obliczenie $2 + 6 = 8$
2. Obliczenie $8 \times 2 = 16$

Działania są wykonywane w akurat tej kolejności, gdyż nawiasy mają najwyższy priorytet, a mnożenie ma wyższy priorytet od dodawania.

Teoretycznie w zależności od priorytetów operatorów $+$ i $\times$ wyrażenie $a + b \times c$ mogłoby być interpretowane jako:

- $(a + b) \times c$ – gdy operator $+$ ma priorytet nie niższy niż operator $\times$,
- $a + (b \times c)$ – w przeciwnym przypadku

Projektanci języków programowania starają się tak dobierać priorytety, żeby odpowiadały intuicji w typowych konstrukcjach. Na przykład operatory arytmetyczne mają priorytety zgodne z kolejnością wykonywania działań, niższy priorytet mają operatory porównania a jeszcze niższy operatory logiczne. Podobnie jak w matematyce, w języku C możesz w wyrażeniach używać nawiasów w celu wymuszenia wykonywania obliczeń w odpowiedniej kolejności (nawiasy mają najwyższy priorytet).

Tabelę priorytetów operatorów w języku C znajdziesz tu.

7.4 Łączność operatorów

Łączność lub **kierunek wiązania** (ang. associativity) to cecha operatorów pozwalająca jednoznacznie interpretować wyrażenie w którym występują operatory o tym samym priorytecie (w szczególności kilka wystąpień tego samego operatora). Wyrażenie $a + b - c$ będzie zinterpretowane jako:

- $(a + b) - c$, jeśli operatory $+$ i $-$ są **lewostronnie łączne** (ang. left-to-right)
- $a + (b - c)$, jeśli operatory $+$ i $-$ są **prawostronnie łączne** (ang. right-to-left)

W języku C podstawowe operatory arytmetyczne są łączne lewostronnie, natomiast operator przypisania – prawostronnie. Operatory o tym samym priorytecie ale innej łączności nie mogą być mieszane ze sobą. Operatory nie posiadające łączności w ogóle nie mogą być mieszanie z jakimikolwiek innymi operatorami.

7.5 Operator przypisania: =

W języku C znak równości (=) nie oznacza matematycznego „równa się”, tylko oznacza **operator przypisania** (ang. assignment operator) – lewy operand operatora przypisania określa miejsce w pamięci, w które zostanie wpisana wartość stanowiąca prawy operand. Operator przypisania jest prawostronnie łączny (czyli najpierw obliczana jest wartość wyrażenia po prawej stronie operatora). Przykładowo, instrukcja

```
x = 2 * 3;
```

przypisuje zmiennej `x` wartość 6. Z kolei instrukcja w rodzaju

```
2 = y;
```

jest błędna, ponieważ 2 jest literałem (a więc z definicji ma stałą wartość).

Wartością wyrażenia zawierającego operator przypisania jest wartość prawego operandu, w związku z tym wyrażenie `q = 5 * 2` ma wartość 10, natomiast po wykonaniu instrukcji

```
int x;
int y = 2 + (x = 4);
```

zmienna `x` będzie mieć wartość 4, a `y` będzie mieć wartość 6.

7.6 Kategorie wartości wyrażeń

Każde wyrażenie w języku C (użycie operatora, wywołanie funkcji, stała, zmienna itp.) posiada dwie niezależne cechy: typ oraz kategorię wartości. Istnieją trzy kategorie wartości: l-wartość, r-wartość oraz desygnator funkcji.

7.6.1 l-wartość

L-wartością (ang. l-value) jest każde wyrażenie typu innego niż `void`, które potencjalnie określa obiekt danych – w szczególności wyrażenie składające się z nazwy zmiennej. Litera „l" w terminie *l-value* pochodzi od słowa *left* (lewy) i bierze się stąd, że termin ten został ukuty jeszcze w czasach, gdy każda l-wartość mogła pojawiać się jako lewy operand operatora przypisania[2]. Ponieważ później wprowadzono do języka modyfikator `const`, język C wyróżnia obecnie **modyfikowalną l-wartość** (ang. modifiable l-value) – czyli obiekt, który może zmieniać swoją wartość – oraz **niemodyfikowalną l-wartość** (ang. non-modifiable l-value), czyli l-wartość której nie można użyć do zmiany stanu obiektu, do którego się ona odwołuje. Stąd lewym operandem operatora przypisania, podobnie jak operandem operatorów inkrementacji i dekrementacji, może być wyłącznie modyfikowalna l-wartość.

7.6.2 r-wartość

R-wartość (ang. r-value) to potoczne określenie tzw. wyrażenia obiektowego nie będącego l-wartością (ang. non-lvalue object expression), czyli wyrażenia nie określającego konkretnego miejsca w pamięci, w efekcie czego nie można uzyskać adresu takiego wyrażenia. Litera „r" w terminie *r-value* pochodzi od słowa *right* (prawy) i bierze się stąd, że termin ten został ukuty jeszcze w czasach, gdy mianem r-wartości określano każde wyrażenie, która mogło pojawiać się jako prawy operand operatora przypisania[3].

Przykładami r-wartości są literały liczbowe i znakowe oraz wyniki działania operatorów nie zwracających l-wartości, m.in. wyniki:

- wywołań funkcji
- rzutowań typów
- operatorów arytmetycznych, relacyjnych, logicznych i bitowych
- operatorów inkrementacji i dekrementacji
- operatora przypisania

Ważne

W języku C++ operatory preinkrementacji, predekrementacji oraz przypisania zwracają l-wartości.

7.6.3 Desygnator funkcji

Desygnator funkcji (ang. function designator), czyli identyfikator wprowadzony poprzez deklarację funkcji, to wyrażenie posiadające typ funkcyjny.

7.7 Podstawowe operatory arytmetyczne

Język C udostępnia cztery podstawowe operatory matematyczne: dodawanie (`+`), odejmowanie (`-`), mnożenie (`*`) i dzielenie (`/`). Język C nie posiada operatora potęgowego, natomiast standardowa biblioteka C zawiera funkcję `pow()`, która jest jego odpowiednikiem.

Symbole `+` i `-` mogą występować w dwóch kontekstach:

- jako dwuargumentowe operatory odpowiednio sumy i różnicy (np. `a + b`, `a - b`)
- jako jednoargumentowe operatory znaku (np. `+a`, `-a`)

W tym drugim przypadku służą do zmiany algebraicznego znaku wartości.[4] Jednoargumentowe operatory znaku mają wyższy priorytet niż mnożenie i dzielenie.

Żaden ze wspomnianych operatorów nie modyfikuje żadnego z operandów, dlatego wykonanie poniższego ciągu instrukcji:

[2] Określenie to wywodzi się z języka programowania CPL.
[3] Określenie to wywodzi się z języka programowania CPL.
[4] W praktyce operator znaku `+` nie wpływa na działanie programu, lecz został dodany w standardzie C90 dla kompletności.

```
int x = 5;
printf("%d\n", x + 20);
printf("%d\n", 20 - x);
printf("%d\n", -x);
printf("%d\n", x);
printf("%d\n", 2 * 5);
```

da następujący wynik:

```
25
15
-5
5
10
```

Wynik operatora dzielenia `/` zależy od typu argumentów, co zostało omówione w rozdziale *3.14.3 Utrata cyfr znaczących*.

7.8 Wybrane inne operatory

Język C zawiera około 40 operatorów, ale niektóre z nich są używane znacznie częściej niż inne. Do najpowszechniejszych należą omówione przed chwilą operatory arytmetyczne i operator przypisania, a także cztery inne operatory, omówione poniżej.

7.8.1 Operator `sizeof`

Jednoargumentowy operator **`sizeof`** zwraca rozmiar operandu *w bajtach*. Operandem może być wyrażenie lub nazwa typu danych (w tym drugim przypadku operand *musi* być ujęty w nawiasy), przykładowo:

```
int s1 = sizeof(char);    // wartość `s1` to 1, bo tyle (z definicji) wynosi
                          //   rozmiar typu `char`
int x = 10;
int s2 = sizeof x;        // wartość `s2` wynosi tyle, ile rozmiar typu `int`
int s3 = sizeof(2 * 5);   // wartość `s3` wynosi tyle, ile rozmiar typu `int`
int s4 = sizeof 8.91;     // wartość `s4` wynosi tyle, ile rozmiar typu `double`
```

7.8.2 Operator `%`

Operator `%` jest wykorzystywany wyłącznie w arytmetyce liczb całkowitych. Zwraca on resztę powstałą w wyniku podzielenia liczby całkowitej po jego lewej stronie przez liczbę całkowitą po jego prawej stronie. Przykładowo, `13 % 5` (czytaj: „13 modulo 5”) wynosi 3, ponieważ 5 mieści się w 13 dwukrotnie i pozostawia resztę 3. Operator modulo przydaje się do sprawdzania parzystości liczby – jeśli reszta z dzielenia przez 2 wynosi 0, liczba jest parzysta, w przeciwnym razie (tj. gdy reszta wynosi 1) liczba liczba jest nieparzysta.

Jak działa operator modulo w przypadku, gdy jeden z operandów jest ujemny? Standard C99 wprowadził regułę, że „gdy wyrażenie `a / b` jest reprezentowalne, wyrażenie `(a/b) * b + a % b` musi wynosić `a`”, zatem przykładowo `-5 % 3` wynosi -2, a `5 % -3` wynosi 2.

7.8.3 Arytmetyczne operatory przypisania

Język C udostępnia tzw. **arytmetyczne operatory przypisania** (ang. arithmetic assignment operators) o ogólnej postaci `<op>=` (gdzie `<op>` oznacza operator arytmetyczny: `+`, `-`, `*`, `/` albo `%`), przy czym wyrażenie `x <op>= y` jest równoważne wyrażeniu `x = x <op> (y)`. Przykładowo:

```
1 int a = 5;
2 int b = 2;
3 a *= 3;       // nowa wartość `a` to: 5 * 3 = 15
4 b *= 3 + 1;   // nowa wartość `b` to: 2 * (3 + 1) = 8
```

Wszystkie operatory przypisania mają ten sam niski priorytet, co operator = (ilustruje to przykład w linii 4, w którym 1 zostaje dodane do 3 przed wykonaniem mnożenia przez `b`).

7.8.4 Operatory inkrementacji i dekrementacji: ++ i --

Operatory **inkrementacji** (ang. incrementation) `++` i **dekrementacji** (ang. decrementation) `--` powodują odpowiednio zwiększenie albo zmniejszenie wartości zmiennej o 1.

Są one dostępne w dwóch trybach:

- przedrostkowym (pre-incrementation, pre-decrementation) – najpierw następuje zmiana wartości zmiennej, potem dalsze operacje z wykorzystaniem zmienionej wartości; np. `++i`, `--i`
- przyrostkowym (post-incrementation, post-decrementation) – najpierw wykonywane są operacje z użyciem pierwotnej wartości zmiennej, a dopiero potem następuje zmiana wartości zmiennej; np. `i++`, `i--`

Przykładowo:

```
int a1 = 3;
int a2 = 3;
int b_pre = 2 * ++a1;  //po wykonaniu instrukcji: b_pre = 8, a1 = 4
int b_post = 2 * a2++; //po wykonaniu instrukcji: b_post = 6, a2 = 4

printf("a = %d, b_pre = %d, b_post = %d\n", b_pre, b_post);
```

Operatory inkrementacji i dekrementacji działają tylko dla modyfikowalnych l-wartości. Oznacza to, że instrukcja `int a = 6++;` jest błędna. Operatory inkrementacji i dekrementacji mają bardzo wysoki priorytet wiązania – pierwszeństwo przed nimi mają tylko nawiasy.

„Co za dużo to niezdrowo!”

Stosuj operatory inkrementacji i dekrementacji ostrożnie i z umiarem – zwiększają one zwięzłość kodu, lecz łatwo popełnić błąd w zapisie.

Jaka będzie wartość zmiannych `n` i `y` po wykonaniu poniższych instrukcji?

```
int n = 3;
int y = n++ + n++;
```

Zmienna `y` może mieć albo wartość `6` – jeśli kompilator dwukrotnie użyje starej wartości – albo wartość `7` (czyli `3 + 4`), jeśli zwiększenie nastąpi przed obliczeniem sumy. Standard języka C nie narzuca jednego słusznego sposobu obliczania takiego wyrażenia.

Na szczęście niektóre kompilatory poinformują nas o potencjalnym niebezpieczeństwie – przykładowo, kompilacja poniższego programu z użyciem kompilatora GCC (bez flag kompilacji):

```
#include <iostream>

int main() {
  int x = 1;
  // Jaki wynik -- 1*1 czy 1*2...?
  std::cout << (x++) * (x++) << std::endl;

  return 0;
}
```

spowoduje wyświetlenie ostrzeżenia

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

Generalnie jednak aby w ogóle uniknąć takich niejednoznaczności nie łącz zbyt wielu operatorów w jednym wyrażeniu i nie rób „optymalizacji” długości kodu kosztem czytelności. W szczególności:

- Nie stosuj operatora inkrementacji lub dekrementacji do zmiennej, która jest częścią więcej niż jednego argumentu funkcji.
- Nie stosuj operatora inkrementacji lub dekrementacji do zmiennej, która w wyrażeniu pojawia się więcej niż jeden raz.
- Jeśli obie wersje danego operatora są semantycznie dopuszczalne, użyj operatora „pre”.

7.9 Czym jest instrukcja?

Instrukcje (ang. statements) są głównymi elementami, z których zbudowane są programy. Instrukcja jest kompletnym poleceniem wydawanym komputerowi.

W tym rozdziale poznasz instrukcję wyrażeniową i blokową, natomiast inne rodzaje instrukcji (m.in. instrukcję wyrażeniową i pętle) poznasz w kolejnych rozdziałach.

7.9.1 Instrukcja wyrażeniowa

Najbardziej podstawowym typem instrukcji są tzw. **instrukcje wyrażeniowe** (ang. expression statements) – są to *wyrażenia* zakończone znakiem średnika. Oznacza to, że `a = 5` jest wyrażeniem, natomiast `a = 5;` to już instrukcja. (Podobnie instrukcją wyrażeniową jest `13;`, choć instrukcja ta nie wnosi wiele do programu. . .) Zauważ, że instrukcja deklaracji nie jest instrukcją wyrażeniową, gdyż pozbawiając ją średnika otrzymasz coś, co *nie jest* wyrażeniem.

7.9.2 Instrukcja blokowa

Instrukcja blokowa (ang. block statement), inaczej **instrukcja złożona** (ang. compound statement) to przynajmniej dwie instrukcje scalone ze sobą poprzez zawarcie ich w nawiasach klamrowych: `{ }`. Instrukcja blokowa jest traktowana przez kompilator jak pojedyncza instrukcja.

> **Ważne**
>
> W standardzie C90 deklaracje zmiennych lokalnych muszą znajdować się zawsze na początku bloku, czyli po otwierającej klamrze „`{`”. Standard C99 dopuścił możliwość deklarowania zmiennych lokalnych w dowolnym miejscu w obrębie bloku.

7.10 Skutki uboczne i punkty sekwencyjne

Skutkiem ubocznym (ang. side effect) nazywamy każdą modyfikację obiektu danych lub pliku. Na przykład skutkiem ubocznym instrukcji

```
x = 1;
```

jest przypisanie zmiennej `x` wartości 1. Mówimy, że przypisanie wartości to *skutek uboczny*, gdyż z punktu widzenia języka C głównym zamiarem jest zawsze obliczanie wyrażeń. Wartością uzyskaną po obliczeniu wyrażenia `x = 1` jest liczba 1, natomiast obliczenie tego wyrażenia daje skutek uboczny w postaci zmiany wartości zmiennej `x` na 1. Skutki uboczne są wywoływane również przez operatory inkrementacji i dekrementacji i w większości przypadków stosujemy je właśnie ze względu na te skutki (podobnie ma się rzecz z operatorem przypisania).

Punkt sekwencyjny (ang. sequence point) jest momentem w trakcie działania programu, w którym zrealizowane zostały wszystkie skutki uboczne. Przykładowo, średnik wymusza dokonanie wszelkich skutków ubocznych (spowodowanych przez operatory przypisania, inkrementacji, itp.) zanim program przejdzie do następnej instrukcji. Punkty sekwencyjne są częścią niektórych z operatorów, którymi zajmiemy się w dalszych rozdziałach. Punktem sekwencyjnym jest ponadto koniec każdego pełnego wyrażenia. Czym jest pełne wyrażenie? Jest to takie wyrażenie, które nie jest częścią większego wyrażenia. Przykładami pełnych wyrażeń są instrukcje wyrażeniowe.

Punkty sekwencyjne pomagają w zrozumieniu, kiedy ma miejsce inkrementacja przyrostkowa. Przykładowo, w poniższej instrukcji:

```
y = (4 + x++) + (6 + x++);
```

wyrażenie `4 + x++` nie jest pełnym wyrażeniem – nie mamy więc żadnej gwarancji co do tego, czy zmienna `x` zostanie zwiększona natychmiast po jego obliczeniu (pełnym wyrażeniem jest tutaj całe wyrażenie przypisania, zakończone średnikiem, który można utożsamiać z punktem sekwencyjnym). Jedynym, co do czego możemy mieć pewność, jest więc fakt, iż zmienna `x` zostanie dwukrotnie zwiększona zanim program przejdzie do wykonania kolejnej instrukcji. Definicja języka C nie określa, czy wartość `x` zostanie zmodyfikowana natychmiast po jej użyciu, czy też dopiero po obliczeniu całego wyrażenia – dlatego też instrukcji zawierających więcej niż jeden operator „post” (postinkrementacji i/lub postdekrementacji) należy bezwzględnie unikać.

7.11 Konwersje typów

Co do zasady, w obrębie każdej danej instrukcji i każdego danego wyrażenia powinny być wykorzystywane obiekty danych należące do tego samego typu. W pewnych sytuacjach język C dopuszcza jednak sytuacje, w których użyte mogą być obiekty różnych typów, przykładowo:

- W przypadku użycia operatora przypisania wartość prawego operandu jest konwertowana do niekwalifikowanego typu lewego operandu. Podobnie w przypadku inicjalizacji zmiennej skalarnej wartość użyta do inicjalizacji jest konwertowana do niekwalifikowanego typu[5] inicjalizowanego obiektu.
- W przypadku wywołań funkcji (posiadających prototyp) wartość każdego z wyrażeń-argumentów jest konwertowana do niekwalifikowanego typu odpowiadającego mu parametru, a wartość operandu instrukcji `return` – do typu zwracanego przez funkcję.
- W przypadku użycia dwuargumentowych operatorów arytmetycznych, relacyjnych, bitowych lub warunkowych operandy zostają poddane niejawnym konwersjom w celu uzyskania tzw. **wspólnego typu rzeczywistego** (ang. common real type), z użyciem którego zostanie obliczona wartość wyrażenia.

W sytuacjach tych zachodzą **niejawne konwersje** typów (ang. implicit conversion). Oto niektóre z reguł określających ten typ konwersji[6]:

1. W pewnych przypadkach typ całkowity o szerokości mniejszej niż typ `int` może zostać **awansowany** (ang. promoted) do typu `int` (jeśli umożliwia on reprezentację wszystkich wartości typu konwertowanego) lub do typu `unsigned int` (w przeciwnym przypadku).
2. Każda wartość typu skalarnego (liczba całkowita albo zmiennoprzecinkowa) może być skonwertowana do typu `_Bool`: wartości równe zeru są konwertowane do 0, a pozostałe – do 1.
3. Między dowolnymi dwoma typami całkowitymi mogą zachodzić niejawne konwersje. Jeśli nie zachodzi sytuacja 1 lub 2, reguły są następujące:
   - jeśli typ docelowy może reprezentować konwertowaną wartość, pozostaje ona niezmieniona;
   - w przeciwnym razie, jeśli docelowy typ nie posiada znaku, wartość 2^b (gdzie b to liczba bitów typu docelowego) jest odpowiednio odejmowana od lub dodawana do wartości źródłowej tyle razy, aż wynik zmieści się w typie docelowym;
   - w przeciwnym razie, jeśli typ docelowy posiada znak, zachowanie zależy od implementacji.

   W szczególności może to doprowadzić do **degradacji** (ang. demotion), przykładowo w instrukcji

```
char c = 'A';  // 'A' to literał typu `int`
```

4. W przypadku konwersji wartości zmiennoprzecinkowej do typu całkowitego zachodzi obcięcie. Jeśli obcięta wartość nie mieści się w typie docelowym, zachowanie programu jest niezdefiniowane. Przykładowo:

```
int n = 3.14;  // wartość `n` wynosi 3
int x = 1e10;  // niezdefiniowane zachowanie dla 32-bitowego typu `int`
```

5. W przypadku konwersji wartości całkowitej do typu zmiennoprzecinkowego wartość nie ulega zmianie, jeśli typ docelowy umożliwia jej dokładną reprezentację; jeśli wartość jest z zakresu typu docelowego, lecz nie może być reprezentowana dokładnie, zostaje zaokrąglona. Jeśli obcięta wartość nie mieści się w typie docelowym, zachowanie programu jest niezdefiniowane. Przykładowo:

```
double d = 10;        // wartość `d` wynosi 10
float f = 20000001;   // wartość  `f` wynosi 20000000.00
float x = 1 + (long long) FLT_MAX;  // niezdefiniowane zachowanie
```

Podobne zasady mają zastosowanie do konwersji pomiędzy typami zmiennoprzecinkowymi.

Choć automatyczne konwersje typów bywają wygodne, mogą także być niebezpieczne – gdyż mogą maskować niektóre rodzaje błędów programisty. Na szczęście istnieją narzędzia w rodzaju programu

[5] Typ niekwalifikowany to typ pozbawiony kwalifikatorów, m.in. kwalifikatora `const`.

[6] Szczegółowe zasady konwersji typów znajdziesz tu.

sprawdzającego *lint* (dostępnego na wielu systemach uniksowych) wykrywające „kolizje” typów (ang. clashes). Oprócz tego, wiele kompilatorów C dla systemów innych niż UNIX potrafi wyłapać możliwe problemy związane z niezgodnością typów po włączeniu odpowiednio wysokiej czułości na błędy. Do dobrej praktyki programistycznej należy unikanie automatycznych konwersji typów, a zwłaszcza degradacji – na przykład poprzez dokonanie niezbędnych konwersji w sposób jawny, z użyciem operatora rzutowania opisanego w rozdziale 7.12.

7.12 Operator rzutowania typów

Rzutowanie (ang. cast) polega na wymuszeniu interpretacji wartości typu `T1` jako wartości typu `T2`. Nazwa typu razem z nawiasem stanowi operator rzutowania. Jego ogólna postać to:

```
(typ) rzutowana_wartość
```

gdzie za `typ` należy podstawić żądaną nazwę typu. Przykładowo, `(int) 3.2` oznacza „potraktuj wartość stojącą po prawej stronie (czyli `3.2`) jako wartość typu **`int`**” (co powoduje m.in. pominięcie części ułamkowej).

Zastanów się nad następującymi dwoma instrukcjami:

```
int x = 1.6 + 1.7;               // zmienna `x` ma wartość 3
int y = (int) 1.6 + (int) 1.7;   // zmienna `y` ma wartość 2
```

W przypadku zmiennej `x` najpierw odbywa się sumowanie dwóch liczb zmiennoprzecinkowych, którego wynikiem jest wartość `3.3` typu **`double`**, a następnie wynik ten jest niejawnie konwertowany na typ **`int`** (aby dało się go przypisać), co powoduje obcięcie do `3`. Z kolei w przypadku zmiennej `y` najpierw każdy ze składników sumy jest rzutowany na typ **`int`**, co powoduje obcięcie każdego z nich do `1`, a następnie wynik sumowania (czyli wartość `2` typu **`int`**) jest przypisywany do zmiennej `y`.

Rozdział 8

Instrukcje warunkowe oraz operatory relacyjne i logiczne

W niniejszym rozdziale zapoznasz się z operatorami relacyjnymi i logicznymi, a także z instrukcją warunkową `if` oraz jej wariantami (`if-else`, `if-else if-else`).

8.1 Operatory relacyjne

Oprócz operatorów arytmetycznych istnieją również operatory, które stosuje się do porównywania danych (podobnie jak w matematyce) – tzw. **operatory relacyjne** (ang. relational operators). Operatory te zostały zebrane w tabeli 8.1 – zwróć szczególną uwagę na operatory *równy* oraz *różny*.

Tablica 8.1. Operatory relacyjne

Operator matematyczny	Operator w C	Opis	Przykład
$=$	`==`	równy	`a == b`
$>$	`>`	większy	`a > b`
$<$	`<`	mniejszy	`a < b`
$\geq$	`>=`	większy lub równy	`a >= b`
$\leq$	`<=`	mniejszy lub równy	`a <= b`
$\neq$	`!=`	różny	`a != b`

Wynik relacji fałszywej wynosi zawsze `0`, z kolei wynik relacji prawdziwej wynosi zawsze `1` (obie te wartości są typu `int`).

Ważne

Pierwotnie język C nie udostępniał specjalnego typu logicznego (o dedykowanych wartościach *prawda* i *fałsz*); zamiast tego reprezentował te wartości korzystając z typu `int`. Standard C99 wprowadził jednak typ `_Bool` (a także jego alias: `bool`) oraz stałe symboliczne `true` i `false` o wartościach odpowiednio 1 i 0 (funkcjonalnośc ta zadeklarowana jest w pliku nagłówkowym `stdbool.h`). Choć typ `_Bool` wewnętrznie reprezentuje wartość logiczną w sposób zbliżony do typu `int`, `_Bool` stanowi pełnoprawny typ danych.

Przykład użycia operatorów relacyjnych:

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

int main(void)
{
    int a = 12;
    int b = 34567;
    bool r1 = (a < b);
    bool r2 = (a == b);
```

```
    printf("a < b  = %d\n", r1);
    printf("a == b = %d\n", r2);

    return EXIT_SUCCESS;
}
```

Priorytet operatorów relacyjnych jest niższy niż operatorów arytmetycznych, ale wyższy niż operatorów przypisania. Operatory relacyjne dzielą się na dwie grupy o różnym priorytecie:

- grupa o wyższym priorytecie: `<`, `<=`, `>`, `>=`
- grupa o niższym priorytecie: `==`, `!=`

Podobnie jak większość operatorów arytmetycznych, operatory relacyjne są lewostronnie łączne, stąd `x != y == z` to tyle, co `(x != y) == z`.

W przypadku złożonych wyrażeń arytmetyczno-relacyjnych lepiej jednak stosować nawiasy w celu poprawy czytelności kodu.

8.2 Instrukcja warunkowa `if`

Instrukcja warunkowa (ang. conditinal statement) `if` podejmuje decyzję „którą instrukcję wykonać w następnej kolejności" na podstawie wartości dowolnego *wyrażenia* podanego w warunku: jeśli wyrażenie jest prawdziwe (niezerowe, tj. ma wartość różną od zera), wykonywana jest dokładnie jedna instrukcja bezpośrednio po instrukcji warunkowej (instrukcja ta może być instrukcją prostą lub złożoną – czyli blokiem kilku instrukcji objętych nawiasami klamrowymi). W przeciwnym przypadku (gdy wyrażenie ma wartość 0) jest ona pomijana.

Ogólna postać instrukcji `if`:

```
if (wyrażenie)
    instrukcja
```

Instrukcję warunkową nazywamy **instrukcją rozgałęzienia** (ang. branching statement), ponieważ jest ona swego rodzaju skrzyżowaniem, na którym program musi wybrać, którą z dwóch ścieżek dalej podążać, przykładowo „jeśli dzielnik wynosi 0, to wyświetl komunikat o błędzie, w przeciwnym razie oblicz wartość dzielenia". Instrukcję warunkowa należy do kategorii instrukcji zwanych czasem **instrukcjami strukturalnymi** (ang. structured statements), gdyż mają one budowę bardziej złożoną niż np. instrukcje przypisania[1].

Ważne

W języku C warunek uznawany jest za spełniony (prawdziwy), jeśli jego wartość jest *różna od 0* (a więc niekoniecznie wynosi dokładnie 1!).

Oto przykładowy program wykorzystujący instrukcję warunkową `if` do weryfikacji wieku osoby:

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

int main(void) {
    int wiek = 60;

    if(wiek > 50) {
        printf("Warunek spelniony - osoba ma ponad 50 lat.\n");
    }

    if(wiek == 50) {
        printf("Warunek spelniony - osoba ma rowno 50 lat.\n");
    }

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

[1] Inne instrukcje z tej kategorii, np. pętle, poznasz w kolejnych rozdziałach.

```
Warunek spelniony - osoba ma ponad 50 lat.
```

8.3 Pułapki przy korzystaniu z instrukcji `if`

Korzystając z instrukcji `if` łatwo o popełnienie pewnych błędów semantycznych, omówionych w poniższych rozdziałach.

8.3.1 Operator porównania (==) a przypisania (=)

Oto przykład demonstrujący problemy z użyciem operatora przypisania zamiast operatora porównania:

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

int main(void) {
    int wiek = 60;

    if (wiek > 50) {
        printf("Warunek spelniony - osoba ma ponad 50 lat.\n");
    }

    if (wiek = 50) { // czy na pewno chodzilo o operator przypisania '='?
        printf("Warunek spelniony - osoba ma rowno 50 lat.\n");
    }

    if (wiek = 0) { // czy na pewno chodzilo o operator przypisania '='?
        printf("Warunek spelniony - osoba dopiero sie urodzila.\n");
    }

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
Warunek spelniony - osoba ma ponad 50 lat.
Warunek spelniony - osoba ma rowno 50 lat.
```

Ważne

Programistom zdarza się przez nieuwagę zamiast operatora porównania == użyć operatora przypisania =. Ponieważ wartość wyrażenia przypisania wynosi tyle, co przypisywana wartość – czyli to, co stoi po prawej stronie – stąd wyrażenie to jest prawdziwe gdy tylko przypisujemy dowolną wartość różną od `0`. Przed takim przypadkowym błędem chroni flaga kompilacji `-Werror=parentheses`.

8.3.2 Średnik (;) po warunku

Uruchomienie poniższego programu:

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

int main(void) {
    int x = 3;

    // Oblicz wartość bezwzględną liczby `x`
    if (x < 0);  // UWAGA! Średnik na koncu if-a!
      x = -x;  // ta instrukcja nie nalezy do if-a...

    printf("%d\n", x);

    return EXIT_SUCCESS;
}
```

da następujący wynik:

```
-3
```

Sam średnik również oznacza instrukcję – instrukcję pustą – zatem powyższy program jest poprawny, choć niekoniecznie robi to, co zapewne miał robić w zamierzeniu jego twórcy.

Ważne

Jeśli program nie działa tak, jak oczekujesz, sprawdź czy po nawiasach obejmujących wyrażenie stanowiące warunek instrukcji `if` nie znalazł się przypadkiem średnik (oznacza instrukcję pustą).

8.3.3 „Instrukcja blokowa" bez nawiasów klamrowych

Porównaj działanie poniższych programów:

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

int main(void) {
    int wiek = 16;

    // nie ma klamer
    if (wiek > 18)
        printf("Osoba jest pelnoletnia.\n");
        printf("Osoba moze wyrobic dowod osobisty.\n");

    return EXIT_SUCCESS;
}
```

oraz

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

int main(void) {
    int wiek = 16;

    // są klamry
    if (wiek > 18) {
        printf("Osoba jest pelnoletnia.\n");
        printf("Osoba moze wyrobic dowod osobisty.\n");
    }

    return EXIT_SUCCESS;
}
```

W pierwszym przypadku wynik wykonania to:

```
Osoba moze wyrobic dowod osobisty.
```

a w drugim nie został wypisany żaden komunikat.

Instrukcja warunkowa `if` w przypadku spełnienia warunku wykonuje tylko dokładnie *jedną* instrukcję, znajdującą się bezpośrednio za instrukcją `if`. W pierwszym programie (mimo wcięć w kodzie) instrukcja `if` obejmuje tylko pierwsze wywołanie funkcji `printf()`.
W drugim programie zgrupowanie obu instrukcji `printf()` w instrukcję blokową sprawia, że i pierwsze i drugie wywołanie tej funkcji nastąpi jedynie w przypadku spełnienia warunku. Z tego powodu wielu programistów stosuje nawiasy klamrowe *zawsze* po instrukcji warunkowej (podobnie jak po pętlach, które poznasz w kolejnym rozdziale).

Dobre praktyki

Wcięcia nie wnoszą nic z punktu widzenia kompilatora – do zinterpretowania Twoich instrukcji wystarczy mu bowiem sam fakt umiejscowienia nawiasów klamrowych oraz znajomość budowy instrukcji warunkowej. Wcięcia stosuj natomiast w swoim własnym interesie tak, aby organizacja Twoich programów była widoczna na pierwszy rzut oka.

8.4 Instrukcja warunkowa `if-else`

Często chcemy, aby działanie programu zależało od pewnego warunku, na przykład „jeśli dzielnik wynosi 0, wówczas wyświetl komunikat o błędzie, *w przeciwnym razie* oblicz wartość dzielenia". Do realizacji takich zadań służy m.in. instrukcja warunkowa **`if-else`** (ang. else – w przeciwnym razie).

Ogólna postać instrukcji **`if-else`**:

```
if (wyrażenie)
    instrukcja1
else
    instrukcja2
```

Jeśli warunek jest prawdziwy (niezerowy), wykonywana jest instrukcja następująca po słowie kluczowym **`if`** (czyli `instrukcja1`), w przeciwnym razie wykonywana jest instrukcja po słowie kluczowym **`else`** (czyli `instrukcja2`). Cała konstrukcja **`if-else`** jest traktowana przez kompilator jako pojedyncza instrukcja.

Oto przykład zastosowania instrukcji **`if-else`**:

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

int main(void) {
    int wiek = 40;

    if (wiek > 50) {
        printf("Warunek spelniony - osoba ma ponad 50 lat.\n");
    } else {
        printf("Warunek nie byl spelniony - zatem osoba nie ma wiecej"
               " niz 50 lat.\n");
    }

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
Warunek nie byl spelniony - zatem osoba nie ma wiecej niz 50 lat.
```

Zwróć uwagę, że instrukcje warunkowe można zagnieżdżać. **Zagnieżdżanie** (ang. nesting) oznacza umieszczanie np. jednej instrukcji danego typu w innej instrukcji tego samego typu. Przykładowo:

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

int main(void) {
    int wiek = 40;

    if (wiek > 50) {
        if (wiek > 70) {
            printf("Osoba ma ponad 70 lat.\n");
        } else {
            printf("Osoba ma ponad 50, lecz mniej niz 70 lat.\n");
        }
    } else {
        printf("Osoba nie ma wiecej niz 50 lat.\n");
    }
```

```
    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
Osoba nie ma wiecej niz 50 lat.
```

W kolejnym rozdziale poznasz sposoby na zapisanie takiego złożonego algorytmu wyboru w bardziej czytelny sposób.

8.5 Konstrukcja `if-else if-else`

Zdarza się, że chcesz wybrać jedną spośród kilku możliwości – na przykład aby określić liczbę miejsc zerowych równania kwadratowego należy rozpatrzyć trzy przypadki: $\Delta > 0$, $\Delta = 0$ oraz $\Delta < 0$.

Do realizacji takiego zadania służy *konstrukcja* **`if-else if-else`** o ogólnej postaci:

```
if (wyrażenie1)
    instrukcja1
else if (wyrażenie2)
    instrukcja2
...
else if (wyrażenie_n-1)
    instrukcja_n-1
[else
    instrukcja_n]
```

Jeśli pierwszy warunek jest prawdą, wykonywana jest instrukcja `instrukcja1`. W przeciwnym razie, jeśli prawdziwe jest drugie wyrażenie, wykonaj instrukcję `instrukcja2` – i tak dalej. Jeśli żaden z warunków nie został spełniony, wykonaj instrukcję po ostatnim wystąpieniu słowa kluczowego **`else`** (czyli instrukcję `instrukcja_n`); ostatni człon **`else`** jest opcjonalny.

W rzeczywistości konstrukcja **`else if`** nie jest niczym nowym, a tylko zastosowaniem tego, co już znasz – członu **`else`** i kolejnej instrukcji warunkowej **`if`** – tyle że elementy te zostały zapisane z użyciem nieco innego *formatowania* kodu (jak pamiętasz, znaki niedrukowane są ignorowane przez kompilator). Aby się o tym przekonać, wystarczy że porównasz wykonanie poniższych fragmentów kodu:

```
int x = 7;
if (x < 3) {
    printf("x < 3\n");
} else if (x > 5) {
    printf("x > 5\n");
} else {
    printf("x in [3, 5]\n")
}
```

oraz

```
int x = 7;
if (x < 3) {
    printf("x < 3\n");
} else
    if (x > 5) {
        printf("x > 5\n");
    } else {
      printf("x in [3, 5]\n")
    }
```

Zatem konstrukcja **`if-else if-else`** to po prostu inny zapis zagnieżdżonej instrukcji warunkowej. Ponieważ cała konstrukcja **`if-else`** jest uznawana za pojedynczą instrukcję, wewnętrzna instrukcja **`if-else`** nie musi być otoczona klamrami – użycie klamer mogłoby jednak uczynić zamiar programisty czytelniejszym.

Aby uczynić program do weryfikacji wieku bardziej czytelnym możesz zatem napisać:

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

```c
int main(void) {
    int wiek = 40;

    if (wiek > 70) {
        printf("Osoba ma ponad 70 lat.\n");
    } else if (wiek > 50) {
        printf("Osoba ma ponad 50, lecz mniej niz 70 lat.\n");
    } else {
        printf("Osoba nie ma wiecej niz 50 lat.\n");
    }

    return EXIT_SUCCESS;
}
```

Formę wykorzystującą wcięcie zagnieżdżonej instrukcji **`if-else`** lepiej zachować dla sytuacji, w których jest ona logicznie uzasadniona – przykładowo, gdy sprawdzane są dwie różne zmienne.

8.5.1 Łączność `else` i `if`

Jeśli w programie występuje wiele instrukcji **`if`** i **`if-else`**, w jaki sposób kompilator decyduje, do której struktury **`if`** należy dany człon **`else`**? Zasady składni języka C mówią, że słowo **`else`** należy do najbliższej instrukcji **`if`**, chyba że klamry wskazują inaczej.

8.6 Operatory logiczne

Operatory logiczne (ang. logical operators) języka C pozwalają na łączenie ze sobą prostych wyrażeń (zwłaszcza wyrażeń zawierających operatory relacyjne) tak, aby konstruować efektywniejsze wyrażenia służące np. w jako wyrażenia testowe w instrukcjach warunkowych. Tabela 8.2 przedstawia operatory logiczne w języku C.

Tablica 8.2. Operatory logiczne w języku C (wg malejącego priorytetu)

Operator w algebrze Boole'a	Operator w języku C	Opis	Przykład
$\neg$	`!`	nie (NOT)	`!a`
$\wedge$	`&&`	i (AND)	`a && b`
$\vee$	`||`	lub (OR)	`a || b`

Priorytety operatorów logicznych w języku C są zgodne z priorytetami ich odpowiedników w algebrze Boole'a.

Ważne

Wynik wyrażenia logicznego wynosi zawsze `1` dla „prawdy", a `0` dla „fałszu", przy czym zwracana wartość jest typu **`int`**.

Oto przykład programu sprawdzającego wartość logiczną wyrażenia $\neg(a < b \wedge b > c)$:

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

int main(void) {
    int a = 1;
    int b = 12;
    int c = 100;

    printf("%d\n", !((a < b) && (b > c)));

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu to:

```
1
```

W pliku nagłówkowym `iso646.h` zdefiniowane są aliasy dla różnych operatorów, m.in. dla operatorów logicznych, przykładowo: `and` dla `&&`, `or` dla `||` oraz `not` dla `!`.

Sprawdzanie wartości logicznej wyrażenia

Ponieważ w języku C dowolna wartość niezerowa traktowana jest jako „prawda”[2], nie ma potrzeby jawnego przyrównywania wartości wyrażenia do wartości `0` lub `1` (ew. równoważnie do stałych symbolicznych `false` lub `true`), przykładowo:

```
if (x < 4) {  // OK
  ...
}

if (x < 4 == 1) {  // poprawne, ale ZŁY STYL
   ...
}

if (x < 4 == true) {  // poprawne, ale ZŁY STYL
   ...
}
```

Kolejność obliczeń

Operatory AND i OR są punktami sekwencyjnymi, zatem wszystkie skutki uboczne są realizowane, zanim program przejdzie do kolejnego operatora. Co więcej, wartości wyrażeń logicznych zawierających operatory AND i OR są *zawsze* obliczane od lewej do prawej[3], a proces zostaje przerwany gdy tylko można już określić wynik całego wyrażenia. Przykładowo, jeśli wyrażenia `a` i `c` mają wartość *prawda* a wyrażenie `b` wartość *fałsz*, w wyrażeniu a AND b AND c wartość wyrażenia c w ogóle nie zostanie obliczona – gdyż po obliczeniu a AND b wiadomo już, że wartością całego wyrażenia będzie *fałsz*). Po angielsku ten mechanizm nazywa się **short-circut evaluation**.

Jak sprawdzić czy wartość wyrażenia należy do zakresu?

Operator AND można wykorzystywać do sprawdzania przynależności zmiennej do zakresu wartości. Przykładowo, aby sprawdzić czy $x \in [90, 100]$, możesz użyć następującego warunku:

```
if (x >= 90 && x <= 100) ...
```

Nie należy naśladować powszechnego w matematyce zapisu:

```
if (90 <= x <= 100) ...  // błąd SEMANTYCZNY
```

Problem polega na tym, że powyższy kod nie jest błędny składniowo, a jedynie semantycznie, a więc kompilator nie wyświetli w tym przypadku komunikatu o błędzie (choć może wyświetlić ostrzeżenie). Dlaczego? Ponieważ operator <= jest lewostronnie łączny, wyrażenie testowe jest rozumiane następująco:

```
if ((90 <= wynik) <= 100) ...
```

Podwyrażenie `90 <= wynik` ma wartość 1 lub 0, w zależności od tego, czy jest prawdziwe, czy fałszywe. Każda z tych wartości jest mniejsza od 100, a więc całe wyrażenie jest *zawsze* prawdziwe, bez względu na wartość zmiennej wynik.

---

[2] zob. rozdz. 3.10 Typ logiczny

[3] W ogólnym przypadku, poza sytuacjami gdy dwa operatory mają wspólny operand, język C nie określa, które części wyrażenia złożonego są obliczane jako pierwsze. Tę niejednoznaczność pozostawiono po to, aby autorzy kompilatorów mogli wybrać najbardziej efektywną kolejność obliczeń na danym komputerze.

Dobre praktyki

Do sprawdzania przynależności do przedziału używaj operatora `&&`.

8.7 Instrukcja warunkowa `switch`

Konstrukcja **`if-else if-else`** jest łatwym sposobem realizacji wyboru jednej z kilku, jednak w pewnych przypadkach – gdy interesują nas konkretne wartości danego wyrażenia, a nie np. zakresy wartości – wygodniej jest skorzystać z instrukcji **`switch`**.

Ogólna postać instrukcji **`switch`**:

```
switch (wyrażenie_testowe) {
    case wartosc1:
        instrukcja1     <- opcjonalnie

    case wartosc2:      <- opcjonalnie
        instrukcja2     <- opcjonalnie

    ...

    case wartosc_n:     <- opcjonalnie
        instrukcja_n    <- opcjonalnie

    default:            <- klauzula opcjonalna
        instrukcja_domyslna
}
```

Wartość wyrażenia testowego musi być typu znakowego, całkowitego, lub wyliczeniowego [4]. Etykiety po słowie kluczowym **`case`** muszą być stałymi całkowitymi lub wyrażeniami zawierającymi wyłącznie stałe całkowite (etykieta nie może zawierać zmiennej). Etykiet, podobnie jak instrukcji, może być dowolnie dużo. Etykieta może nie posiadać następującej po niej instrukcji (o ile któraś z kolejnych etykiet lub człon **`default`** ją posiada). Obecność członu **`default`** nie jest wymagana.

Działanie instrukcji **`switch`** jest następujące:

- Obliczona zostaje wartość wyrażenia testowego.
- Wyszukiwana jest etykieta **`case`** o wartości odpowiadającej wartości wyrażenia testowego.
  - Jeśli taka etykieta została odnaleziona, wykonywane są wszystkie następujące po niej instrukcje – aż do instrukcji przerywającej (z reguły **`break`** lub **`return`**). Oznacza to, że w przypadku braku instrukcji przerywającej wykonane zostaną także instrukcje odpowiadające kolejnym przypadkom.
  - W przeciwnym przypadku (jeśli żadna z etykiet nie odpowiada wartości wyrażenia testowego), wykonany zostanie człon **`default`** – o ile występuje.

Instrukcja **`break`** powoduje natychmiastowe opuszczenie pętli albo instrukcji **`switch`** i przejście do kolejnego etapu programu (czyli do instrukcji bezpośrednio po instrukcji przerwanej).

Oto (anty)przykład zastosowania instrukcji **`switch`**:

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

int main(void) {
    int n = 1;

    switch (n) {
        case 1:
            printf("Jeden!\n");
            // zauwaz brak instrukcji `break`
        case 2:
            printf("Dwa!\n");
            break;
```

[4] Typu wyliczeniowe poznasz w rozdziale *18.2 Typy wyliczeniowe*.

```
        case 3:
            printf("Trzy!\n");
            break;

        default:
            printf("Cos innego...\n");
            break;
    }

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
Jeden!
Dwa!
```

Przed sytuacjami, gdy programista przez nieuwagę zapomni dodać instrukcji **break** chroni flaga kompilacji `-Werror=implicit-fallthrough=`.

8.8 Operator warunkowy: ? :

Język C udostępnia skrótowy sposób wyrażenia jednego z wariantów instrukcji **if-else** – sposób ten nosi nazwę **wyrażenia warunkowego** (ang. conditional expression), a jego głównym elementem jest operator warunkowy `? :`

Oto składnia operatora warunkowego (składa się on z dwóch symboli i wymaga trzech operandów):

```
(warunek) ? wyrazenie_gdy_prawda : wyrazenie_gdy_falsz
```

Operator warunkowy zwraca:

- wynik pierwszego wyrażenia, jeśli warunek jest prawdziwy (niezerowy),
- wynik drugiego wyrażenia w przeciwnym przypadku.

W poniższym przykładzie do zmiennej `x_abs` zostaje przypisana wartość bezwzględna z `x`:

```
x_abs = (x < 0) ? -x : x;
```

Zapis ten jest równoważny następującej instrukcji **if-else**:

```
if (x < 0) {
   x_abs = -x;
} else {
   x_abs = x;
}
```

Choć wyrażenie warunkowe można z reguły zastąpić instrukcją warunkową **if-else**, wersja z operatorem warunkowym `? :` jest bardziej zwięzła (a dla doświadczonych programistów – także bardziej czytelna).

Rozdział 9

Instrukcje sterujące: Pętle

Aby pisać programy efektywnie, musisz umieć dobrze kontrolować **przebieg programu** (ang. program flow), na który składają się poniższe elementy udostępniane przez większość języków wysokiego poziomu:

- Wykonanie ciągu instrukcji.
- Wybór jednego z kilku ciągów instrukcji za pomocą testu (rozgałęzienia).
- Powtarzanie ciągu instrukcji do momentu spełnienia jakiegoś warunku (pętle).

Pierwsze dwa elementy już znasz, w tym rozdziale zapoznasz się z pętlami: `for`, `while` i `do-while`.

9.1 Instrukcja `for`

Instrukcja `for` (ang. for – dla) pozwala na wykonywanie fragmentu programu określoną liczbę razy. Wykorzystuje ona trzy wyrażenia rozdzielone *średnikami*, są to kolejno:

- inicjalizacja licznika – wartość tego wyrażenia jest obliczana *tylko jeden raz*, zaraz przed pierwszą iteracją (tj. obiegiem pętli)
- warunek (test) – wartość tego wyrażenia jest obliczana *przed* każdym potencjalnym wykonaniem pętli (jeśli wyrażenie to jest fałszywe, pętla zostaje zakończona)
- zmiana (aktualizacja) – wyrażenie, którego wartość jest obliczana *na końcu* każdej iteracji, przy czym interesujące są wyłącznie jego skutki uboczne

Ogólna postać pętli `for`:

```
for (inicjalizacja; test; aktualizacja)
    instrukcja
```

Każde z tych trzech wyrażeń sterujących jest pełnym wyrażeniem, a więc wszelkie skutki uboczne wywołane przez jedno z nich są realizowane przed przejściem do kolejnego wyrażenia. Pętla `for` kończy się jedną instrukcją (prostą albo złożoną). Każdy cykl sprawdzania warunku, wykonania instrukcji i aktualizacji licznika nazywa się **iteracją** pętli `for` (ang. iteration).

Pętla `for` to tzw. pętla z **warunkiem wejścia** (ang. entry condition), co oznacza że warunek sprawdzany jest *przed* każdym jej obiegiem (także pierwszym) – zatem pętla może nie wykonać się ani razu, jeśli na samym początku warunek nie został spełniony.

Oto przykład demonstrujący wielokrotne wypisywanie na standardowe wyjście:

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

int main(void) {
    for (int i = 1; i <= 3; ++i) {
        printf("%d\n", i);
    }
    printf("Koniec.\n");

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
1
```

```
2
3
Koniec.
```

W pętli for masz pełną dowolność w kwestii wyboru każdego z trzech wyrażeń, m.in. możesz:

- użyć operatora dekrementacji lub `*=` do aktualizacji licznika
- aktualizować licznik inną wartością niż 1 (np. `i += 2`)
- sprawdzić warunek inny niż liczba powtórzeń (np. czy wartość pewnej zmiennej nie będącej licznikiem spełnia zadany warunek)
- pominąć każde z wyrażeń (pamiętając o pozostawieniu średników)
- ...

9.2 Instrukcja `while`

Instrukcja **`while`** wykonuje zadaną instrukcję dopóty, dopóki spełniony jest warunek testowy.

Oto ogólna postać instrukcji **`while`**:

```
while (wyrażenie_testowe)
    instrukcja
```

Każdy cykl składający się z testu i wykonania instrukcji nazywany jest iteracją pętli **`while`**.

Pętla **`while`** to pętla z warunkiem wejścia (podobnie jak pętla **`for`**), co oznacza, że warunek sprawdzany jest przed wykonywaniem instrukcji w ciele pętli.

9.3 Instrukcja `do-while`

Instrukcja **`do-while`**, podobnie jak pętla **`while`**, wykonuje instrukcję dopóki spełniony jest warunek testowy.

Oto ogólna postać instrukcji **`do-while`**:

```
do
    instrukcja
while (wyrażenie_testowe);
```

> **Ważne**
>
> Zwróć uwagę na średnik występujący po **`while`** `(...)`!

Różnica między dwoma typami pętli **`while`** polega na tym, że **`do-while`** to tzw. pętla z **warunkiem wyjścia** (ang. exit condition) – oznacza to, że warunek sprawdzany jest *po* wykonaniu instrukcji w ciele pętli, a więc pętla *zawsze* wykona się *co najmniej raz*.

9.4 Kiedy kończy się pętla?

Projektując pętlę (obojętnie jakiego typu) pamiętaj, że musi ona zawierać coś, co zmienia wartość testowanego wyrażenia tak, aby w pewnym momencie stało się ono fałszywe – w przeciwnym wypadku otrzymasz **pętlę nieskończoną** (ang. infinite loop), czyli pętlę, która nigdy się nie zakończy. (Ściśle rzecz biorąc, do wyjścia z pętli można także użyć np. instrukcji **`break`**, omówionej w dalszej części tego rozdziału.) Zastanów się nad następującym przykładem:

```
int index = 1;
while (index < 5) {
    printf("Dzień dobry!\n");
}
```

Powyższy fragment wyświetla komunikat bez końca, gdyż w pętli nie ma niczego, co powodowałoby zmianę początkowej wartości zmiennej `index`. Zbliżony problem wystąpi w poniższym przypadku:

```
int index = 1;
while (--index < 5) {
    printf("Dzień dobry!\n");
}
```

gdyż choć wartość zmiennej `index` ulega zmianie, zmienia się ona w złym kierunku! Jednak ta pętla w końcu zakończy swoje działanie – stanie się to wówczas, gdy wartość zmiennej `index` wyniesie `INT_MIN` (wówczas kolejna dekrementacja spowoduje „przekręcenie się" wartości na `INT_MAX`). Pamiętaj, że decyzja o zakończeniu bądź kontynuowaniu pętli zapada tylko w momencie obliczenia wyrażenia testowego!

Podobnie jak w przypadku instrukcji warunkowej, zwracaj szczególną uwagę na to, czy nie mylisz operatora porównania (==) z operatorem przypisania (=), czy poprawnie zgrupowałeś instrukcje z użyciem klamer, oraz gdzie umieszczasz średniki:

```
int index = 1;
while (index = 5);  // UWAGA: średnik na końcu; operator przypisania
    --index;
```

W powyższym przykładzie obecność średnika po członie wyrażenia testowego sprawia, że wartość zmiennej `index` nie ulegnie zmianie. Nawet jednak po usunięciu tego średnika pozostaje problem z warunkiem `index = 5`, który jest zawsze spełniony. Problemy te zostały dokładniej omówiony w rozdziale *8.3 Pułapki przy korzystaniu z instrukcji `if`*.

9.5 Instrukcje `continue` i `break`

Domyślnie po wejściu do pętli program wykonuje wszystkie należące do niej instrukcje aż do kolejnego sprawdzenia wyrażenia testowego. Instrukcje **`continue`** i **`break`** umożliwiają wyjście poza ten schemat – pozwalają one pominąć fragment pętli lub jej zakończenie, np. w oparciu o wynik testu wykonywanego wewnątrz pętli.

Oto przykład użycia instrukcji **`continue`** i **`break`** w prostym programie:

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

int main(void) {
    for (int i = 1; i <= 10; ++i) {
        if (i == 4) {
            printf("<ominiete> ");
            continue;
        }

        printf("%d ", i);

        if (i == 8) {
            printf("<przerwane> ");
            break;
        }
    }

    printf("\nKoniec.\n");

    return EXIT_SUCCESS;
}
```

Rezultat wykonania programu:

```
1 2 3 <ominiete> 5 6 7 8 <przerwane>
Koniec.
```

9.5.1 Instrukcja `continue`

Instrukcja **`continue`** powoduje natychmiastowe przejście do kolejnego obiegu pętli, z pominięciem ewentualnych dalszych instrukcji po **`continue`** (a więc w przypadku pętli **`while`** i **`do-while`** powoduje to przejście do sprawdzania warunku, a w przypadku pętli **`for`** – dokonanie aktualizacji, a następnie przejście do warunku).

Instrukcję **`continue`** można zastąpić choćby z użyciem instrukcji warunkowej, i to na dwa sposoby. Przykładowo, w poniższym fragmencie kodu:

```
while (test) {
    if (x < VAL_MIN || x > VAL_MAX) {
        continue;
    }

    /* instrukcje */
}
```

- Można usunąć instrukcję **continue** i umieścić resztę zawartości pętli w bloku **else**:

```
if (x < VAL_MIN || x > VAL_MAX) {
} else {
    /* instrukcje */
}
```

- Można też zanegować powyższy warunek, aby uniknąć konieczności stosowania bloku **else**:

```
if (x >= VAL_MIN && x <= VAL_MAX) {
    /* instrukcje */
}
```

Zaletą użycia instrukcji **continue** jest możliwość wyeliminowania jednego poziomu wcięcia w głównej grupie instrukcji należących do pętli. Jest to istotne z punktu widzenia czytelności kodu, zwłaszcza jeśli instrukcje są długie lub głęboko zagnieżdżone.

Instrukcja **continue** przydaje się również jako. . . „wypełniacz”. Przykładowo, poniższa pętla odczytuje i porzuca wszystkie znaki do końca wiersza włącznie[1]:

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

Problem polega na tym, że samotny średnik jest słabo widoczny wśród innych instrukcji. Użycie zamiast średnika instrukcji **continue** sprawi, że kod źródłowy stanie się znacznie czytelniejszy:

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

Nie używaj instrukcji **continue** wówczas, gdy komplikuje ona kod źródłowy zamiast go upraszczać. Zastanów się na przykład nad następującym fragmentem:

```
int ch;
while ((ch = getchar()) != '\n') {
    if (ch == '\t') {
        continue;
    }
    putchar(ch);
}
```

Powyższa pętla pomija tabulatory, a kończy działanie w przypadku napotkania znaku nowej linii. Jest ona równoważna następującej, bardziej zwięzłej pętli:

```
while ((ch = getchar()) != '\n') {
    if (ch != '\t') {
        putchar(ch);
    }
}
```

Często zdarza się, że (tak jak w powyższym przykładzie) negacja warunku testowego w instrukcji **if** eliminuje potrzebę obecności instrukcji **continue**.

[1]Ten idiom programistyczny został dokładniej omówiony w rozdziale *16.5 Czyszczenie bufora wejściowego*.

9.5.2 Instrukcja `break`

Instrukcja **`break`** powoduje natychmiastowe opuszczenie pętli (albo instrukcji **`switch`**) i przejście do kolejnego etapu programu (czyli do instrukcji bezpośrednio po instrukcji przerwanej).

Tak jak w przypadku instrukcji **`continue`**, unikaj stosowania instrukcji **`break`** wówczas, gdy komplikuje ona kod. Przykładowo, logika poniższej pętli:

```
int ch;
while ((ch = getchar()) != '\n') {
    if (ch == '\t') {
        break;
    }
    putchar(ch);
}
```

staje się dużo jaśniejsza, gdy oba testy znajdują się w jednym miejscu:

```
while ((ch = getchar()) != '\n' && ch != '\t') {
    putchar(ch);
}
```

Ważne

Używaj **`continue`** i **`break`** w pętlach tylko wtedy, gdy zwiększa to czytelność kodu!

9.6 Operator przecinkowy: ,

Operator przecinkowy (`,`) łączy dwa wyrażenia w jedno, gwarantując, że wyrażenie z lewej strony zostanie obliczone jako pierwsze. Wartością całego wyrażenia jest wartość jego prawostronnej części. Operator przecinkowy jest zwykle używany w pętlach **`for`**, gdzie umożliwia zmieszczenie większej ilości informacji w ramach wyrażenia inicjalizującego lub aktualizującego licznik(i) – umożliwia odpowiednio inicjalizację i aktualizację więcej niż jednej zmiennej.

Dobre praktyki

Choć operator przecinkowy jest lewostronnie łączny, do dobrej praktyki należy nie opieranie swojego kodu na tej zależności (tj. wyrażenia łączone za pomocą przecinka powinny być od siebie niezależne).

Ważne

W języku C przecinek jest również używany jako znak rozdzielający, a więc przecinki w instrukcjach

```
double x, y;
```

oraz

```
printf("%d\n", x);
```

są znakami rozdzielającymi, a *nie* operatorami.

9.7 Której pętli użyć?

Gdy już podejmiesz decyzję o użyciu pętli, który typ powinieneś wybrać? Najpierw zastanów się, czy potrzebujesz pętli z warunkiem wejścia, czy pętli z warunkiem wyjścia – właściwym wyborem powinna być najczęściej ta pierwsza możliwość. Istnieje kilka powodów, dla których programiści uważają pętlę z warunkiem wejścia za lepsze rozwiązanie. Pierwszym z nich jest ogólna zasada „pomyśl zanim zrobisz”. Drugim powodem jest fakt, iż umieszczenie testu na początku pętli poprawia czytelność programu. Po trzecie, w większości zastosowań istotne jest całkowite pominięcie pętli, jeśli warunek nie jest spełniony na początku (tj. przed pierwszą iteracją).

W praktyce programiści starają się unikać stosowania **`do-while`**, gdyż:

- umieszczanie warunku na końcu pętli zmniejsza czytelność kodu, natomiast
- bez trudu można tak zmienić kod, aby zastąpić pętlę **`do-while`** pętlą **`while`** lub pętlą **`for`**.

Przyjmijmy więc, że potrzebujesz pętli z warunkiem wejścia. Którą wybrać: `for` czy `while`? Jest to częściowo kwestia gustu, ponieważ możliwości obu pętli pokrywają się. Pętlę `for` można na przykład upodobnić do pętli `while` pomijając pierwsze i trzecie wyrażenie sterujące – innymi słowy

```
for ( ;test; ) ...
```

jest równoważne

```
while (test) ...
```

Z kolei pętlę `for` można łatwo zastąpić pętlą `while` wzbogaconą o inicjalizację i aktualizację zmiennych, przykładowo fragment:

```
inicjalizacja;
while (test) {
  zawartość;
  aktualizacja;
}
```

jest równoważny fragmentowi

```
for (inicjalizacja; test; aktualizacja) {
  zawartość;
}
```

Niektóre z przedstawionych przykładów pętli `while` były tzw. **pętlami nieokreślonymi** (ang. indefinite loop). Oznacza to, że liczby wykonanych powtórzeń nie można było przewidzieć przed zakończeniem pętli (np. gdy odczytujemy dane ze standardowego wejścia do momentu wystąpienia znaku niedrukowanego). Pozostałe przykłady były jednak tzw. **pętlami liczącymi** (ang. definite loop), w których liczba powtórzeń była z góry określona.

Dobre praktyki

Stosuj pętlę `for` tylko wtedy, gdy masz do czynienia z pętlą liczącą. *W przypadku* pętli niezdefiniowanej *lepiej (czytelniej) jest użyć pętli `while`.*

9.8 Pętle zagnieżdżone

Pętlą zagnieżdżoną (ang. nested loop) nazywamy pętlę znajdującą się w obrębie innej pętli[2]. Pętla wewnętrzna wykonuje swój pełny zakres powtórzeń w każdym cyklu pętli zewnętrznej.

Powszechnym zastosowaniem pętli zagnieżdżonych jest operowanie na lub wyświetlanie danych w rzędach i kolumnach. Jedna z pętli zajmuje się wówczas np. wszystkimi kolumnami w danym rzędzie, a druga – wszystkimi rzędami[3].

Oto prosty przykład pętli zagnieżdżonej.

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

int main(void) {
    const int N_ROWS = 3;
    const int N_CHARS = 10;

    for (int row_inx = 0; row_inx < N_ROWS; ++row_inx) {
        for (char ch = 'A'; ch < ('A' + N_CHARS); ++ch) {
            printf("%c", ch);
        }
        printf("\n");
    }

    return EXIT_SUCCESS;
}
```

[2]Podobnie, jak zagnieżdżać można np. instrukcje warunkowe (zob. rozdz. *8.4 Instrukcja warunkowa `if-else`*).

[3]Taka kolejność wynika z faktu, że standardowe wyjście operuje na wierszach – po wypisaniu wiersza ciężko do niego powrócić.

Wynik wykonania powyższego programu:

```
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
```

Zauważ, że pętla wewnętrzna wykonuje swój pełen zakres powtórzeń w każdym pojedynczym cyklu pętli zewnętrznej.

W powyższym przykładzie pętla wewnętrzna wykonywała dokładnie tę samą czynność w każdym cyklu pętli zewnętrznej. Można jednak sprawić, aby pętla wewnętrzna zachowywała się inaczej w każdej iteracji, uzależniając jeden z jej elementów od stanu pętli zewnętrznej. W poniższym przykładzie znak, od którego rozpoczyna wyświetlanie pętla wewnętrzna, został uzależniony od wartości licznika pętli zewnętrznej:

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

int main(void) {
    const int N_ROWS = 3;
    const int N_CHARS = 10;

    for (int row_inx = 0; row_inx < N_ROWS; ++row_inx) {
        for (char ch = (char) ('A' + row_inx); ch < ('A' + N_CHARS); ++ch) {
            printf("%c", ch);
        }
        printf("\n");
    }

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
ABCDEFGHIJ
BCDEFGHIJ
CDEFGHIJ
```

9.8.1 Instrukcje `continue` i `break` a pętle zagnieżdżone

W przypadku pętli zagnieżdżonych działanie tych instrukcji odnosi się wyłącznie do najbliższej pętli do której przynależy dana instrukcja **`continue`** lub **`break`** – a nie do którejś z pętli o niższym poziomie zagnieżdżenia (czyli pętli „płytszych”). Przykładowo, wykonanie poniższego programu:

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

int main(void) {
    for (int i = 0; i < 2; ++i) {
        printf("i = %d: ", i);
        for (int j = 0; j < 3; ++j) {
            if (j == 0) {
                continue;  // należy do `for (int j = 0; j < 3; ++j) ...`
            }
            printf("%d ", j);
        }
        printf("\n");
    }

    return EXIT_SUCCESS;
}
```

spowoduje wyświetlenie na konsoli poniższych komunikatów:

```
i = 0: 1 2
i = 1: 1 2
```

(Zwróć uwagę, że ani razu nie została wypisana wartość 0 dla zmiennej `j`).

9.8.2 Instrukcja `goto`

Instrukcja **`goto`** powoduje tzw. **skok** (ang. jump) do instrukcji oznaczonej podaną etykietą (tj. program kontynuuje wykonanie od instrukcji docelowej, oznaczonej etykietą). Pomiędzy etykietą a instrukcją docelową musi znajdować się dwukropek. Nazwy etykiet muszą być zgodne z zasadami nazewnictwa zmiennych; instrukcja docelowa może znajdować się przed instrukcją **`goto`** lub po niej.

Oto ogólna postać instrukcji skoku **`goto`**:

```
goto etykieta;

etykieta:
instrukcja
```

Dawniej instrukcja **`goto`** była fundamentem wielu programów[4], natomiast w języku C można znakomicie obyć się bez niej (stosując odpowiednio m.in. instrukcje warunkowe i pętle)[5]. W praktyce w języku C istnieje tylko jedno dopuszczalne zastosowanie instrukcji **`goto`**: wyjście z układu pętli zagnieżdżonych (instrukcja **`break`** powoduje bowiem zakończenie tylko jednej pętli – tej, w której się znajduje bezpośrednio znajduje). Takie dopuszczalne użycie instrukcji **`goto`** demonstruje schematycznie poniższy przykład:

```
for (size_t i = 0; i < VAL1; ++i) {
    for (size_t j = 0; j < VAL2; ++j) {
        /* instrukcje */
        if (test) {
            goto final;
        }
    }
}
final:
/* dalsze instrukcje */
```

[4] m.in. pisanych w starszych wersjach języków BASIC i FORTRAN

[5] Problemy z nadmiernym stosowaniem instrukcji **`goto`** dostrzegali już Kemighan i Ritchie, którzy określili ją mianem „pola nieograniczonych nadużyć” i zalecali, aby „stosować ją z umiarem, jeśli w ogóle”.

Rozdział 10

Funkcje

W tym rozdziale dowiesz się więcej o funkcjach: jak je definiować, jak korzystać z argumentów i wartości zwracanych, oraz jak przekazywać zmienne wskaźnikowe jako argumenty funkcji. Poznasz także rekurencję.

10.1 Programowanie strukturalne

Jak wspomniano w rozdziale 1, podejście strukturalne opiera się na podziale kodu źródłowego programu na procedury i hierarchicznie ułożone bloki z wykorzystaniem struktur kontrolnych w postaci instrukcji wyboru i pętli.

W miarę pisania programów wykonujących coraz bardziej złożone operacje umieszczanie całej jego logiki w jednym bloku (w przypadku programów konsolowych w język C – wewnątrz funkcji `main()`) staje się mało wygodne. Aby „rozbić" taki program na mniejsze, funkcjonalne elementy – w przypadku programowania strukturalnego takimi elementami są *funkcje*. Każda **funkcja** (ang. function) określa pewien z góry określony ciąg operacji (kroków algorytmu).

Funkcje umożliwiają między innymi:

- dekompozycję kodu – podział kodu na mniejsze, pogrupowane logicznie fragmenty;
- unikanie powtórzeń – jeśli określone zadanie ma zostać wykonane kilkakrotnie, wystarczy napisać jedną funkcję i wywoływać ją tam, gdzie jest to potrzebne;
- separację i enkapsulację – ukrycie „na zewnątrz" przebiegu pewnych procesów oraz uniemożliwienie dostępu do wewnętrznych danych funkcji; poprawia to bezpieczeństwo kodu;

a przez to pozwalają m.in. na wygodną integrację różnych fragmentów kodu dostarczanych przez różnych programistów.

Wielu programistów lubi wyobrażać sobie funkcje jako „czarne skrzynki", określone przez wchodzące do nich dane, wykonywane przez nie działania i zwracane wartości – to, co dzieje się wewnątrz takiej funkcji-czarnej skrzynki (tj. jej szczegóły implementacyjne), nie jest Twoim zmartwieniem, o ile nie jesteś osobą odpowiedzialną za implementację takiej funkcji. Na przykład aby skorzystać z funkcji `printf()` wystarczy, że zapoznasz się z jej dokumentacją – nie musisz analizować jej kodu źródłowego. Traktowanie funkcji w ten sposób pomaga skoncentrować się na ogólnej budowie programu, bez zagłębiania się w szczegóły. Zanim zaczniesz myśleć o tym, jak napisać funkcję, zastanów się dokładnie nad tym, jakie zadania ma ona realizować i jakie jest jej miejsce w programie jako całości.

Dana funkcja powinna grupować operacje o podobnym poziomie złożoności. Przykładowo, pisząc funkcję do rozpatrywania rejestracji użytkowników w systemie (rejestracja wymaga podania adresu e-mail i polskiego numeru telefonu komórkowego) można by zaimplementować ją w sposób zbliżony do poniższego pseudokodu:

```
rozpatrz_rejestrację(email, nr_tel):
    jeśli e-mail nie zawiera małpy - zakończ
    jeśli e-mail nie zawiera kropki w domenie - zakończ
    ...
    jeśli numer telefonu nie zawiera dokładnie 9 cyfr - zakończ
    ...
    otwórz połączenie z bazą danych
    wstaw do bazy dane użytkownika
    zamknij połączenie z bazą danych
```

```
    ...
    przygotuj treść wiadomości-potwierdzenia
    wyślij e-mail potwierdzający rejestrację
```

Takie podejście jest zdecydowanie mniej czytelne niż podejście poniższe, w których funkcja do rozpatrywania rejestracji korzysta z czterech wyspecjalizowanych funkcji:

```
rozpatrz_rejestrację(email, nr_tel):
    zweryfikuj_adres_email(email)
    zweryfikuj_numer_telefonu(nr_tel)
    jeśli dane poprawne:
        utwórz_użytkownika_w_bazie(email, nr_tel)
        wyślij_email_potwierdzający_rejestrację(email)
```

10.2 Tworzenie funkcji

W ogólnym ujęciu funkcja to wydzielony fragment kodu, spełniający określone zadanie (np. obliczenie wartości, wyświetlenie danych). Zazwyczaj do poprawnego wykonania wspomnianych operacji należy dostarczyć pewnych danych wejściowych; funkcja może również zwracać pewną wartość (np. wynik obliczeń).

Oto przykładowa definicja funkcji zwracającej sumę dwóch liczb całkowitych przekazanych jako argumenty funkcji:

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

Zwróć uwagę, że język C opiera się na ścisłej typizacji. Oznacza to, że musisz określać typ danych używanych w programie – koniecznym jest podanie typu parametrów i wartości zwracanej przez funkcję. Pierwszy `int` (przed nazwą funkcji) oznacza, że funkcja zwraca wartość typu całkowitego, potem następuje nazwa funkcji (`sumuj`), a na końcu – w nawiasach – podana została lista parametrów funkcji (wraz z typami).

Funkcji używa się w trzech kontekstach:

- **deklaracja** – wprowadzenie nazwy funkcji, określenie typów parametrów i typów wartości zwracanej
- **definicja** – określenie treści funkcji, czyli tzw. **ciała funkcji** (ang. function body)
- **wywołanie** (ang. call) – użycie funkcji z konkretnymi wartościami parametrów (czyli z argumentami) i ewentualne użycie wartości zwracanej

Ważne

W języku C funkcja musi być zadeklarowana przed pierwszym wywołaniem, ale nie musi być wtedy jeszcze zdefiniowana (innymi słowy, w kodzie programu deklaracja funkcji musi wystąpić przed pierwszym użyciem, natomiast definicję można umieścić w dowolnym dalszym miejscu w programie). Pozwala to zwiększyć czytelność kodu: zwyczajowo na początku pliku umieszcza się deklaracje, a na końcu – definicje.

10.2.1 Deklaracja (prototyp)

Deklaracja funkcji, zwana inaczej **prototypem funkcji** (ang. prototype), określa typ wartości zwracanych oraz liczbę i typy argumentów. Jej ogólna postać to:

```
typ_wartości_zwracanej nazwa_funkcji(typ_parametru_1 [nazwa_parametru_1,
  typ_parametru_2 nazwa_parametru_2, ..., typ_parametru_n nazwa_parametru_n]);
```

przy czym fragment umieszczony w nawiasach kwadratowych jest opcjonalny. W szczególności opcjonalne jest podawanie w deklaracji funkcji nazw parametrów (choć ich podawanie jest dobrą praktyką, poprawiającą czytelność kodu) – nazwy parametrów są potrzebne dopiero w momencie definiowania funkcji.

Deklaracja przytoczonej w poprzednim rozdziale funkcji `sumuj()` wyglądałaby zatem następująco:

```
int sumuj(int a, int b);
```

lub równoważnie, w skróconej postaci:

```
int sumuj(int, int);  // niezalecane
```

W przypadku deklaracji nie podajemy zatem „logiki” funkcji – zamiast tego umieszczamy średnik po liście parametrów.

W przypadku, gdy funkcja nie przyjmuje argumentów, zamiast listy parametrów należy podać specjalny typ **void** (ang. void – pustka), np.:

```
int foo(void) {
  return 1;
}
```

Podobnie funkcja, która nie zwraca żadnej wartości, powinna być zadeklarowana jako typu **void**:

```
void powitaj(void) {
    printf("Witaj!\n");
}
```

Dzięki zadeklarowaniu funkcji kompilator może sprawdzić, czy wywołanie funkcji pasuje do jej prototypu – i ostrzec nas o potencjalnych problemach z niezgodnością typów.

Zauważ, że opisowe nazwy funkcji pokazują w jasny sposób działanie i organizację programu. Dzięki podejściu modularnemu możesz pracować nad każdą funkcją z osobna, dopóki nie będzie wykonywała ona swojego zadania w należyty sposób, a jeśli Twoje funkcje będą wystarczająco wszechstronne, będziesz mógł wykorzystać je również w innych programach.

Nazwy zmiennych są prywatne dla funkcji. Oznacza to, że nazwa zadeklarowana w jednej funkcji nie wchodzi w konflikt z taką samą nazwą zadeklarowaną w innej funkcji (więcej o tym zagadnieniu przeczytasz w rozdziale *13 Klasy zmiennych*).

10.2.2 Definicja

Definicja funkcji to po prostu prototyp funkcji rozszerzony o jej logikę – zamiast średnika na końcu umieszczamy blok kodu, a w nim instrukcje do wykonania w ramach funkcji. Definicję możesz umieścić niemal w dowolnym miejscu pliku – język C nie pozwala jedynie na definiowanie funkcji zagnieżdżonych.

Do zwracania wartości przez funkcję służy instrukcja **return** – powoduje ona przerwanie dalszego wykonania funkcji i zwrócenie wartość wyrażenia stojącego na prawo od słowa kluczowego **return**.

Ogólna postać definicji funkcji to:

```
typ_wartości_zwracanej nazwa_funkcji(lista_parametrów) {
    instrukcje;  // mogą zawierać inne instrukcje `return`
    return wartość_zwracana;
}
```

Wewnątrz funkcji nie zwracającej żadnej wartości (tj. funkcji typu **void**) również można stosować instrukcję **return;** – powoduje ona wtedy jedynie przerwanie dalszego wykonywania funkcji i powrót sterowania do miejsca wywołania funkcji, bez zwracania jakiejkolwiek wartości.

Ważne

Jeśli wykonanie programu osiągnie koniec ciała funkcji (tj. symbol „}”) w przypadku funkcji będącej typu innego niż **void**, to próba odwołania się do wartości zwróconej skutkuje niezdefiniowanym zachowaniem:

```
int foo(void) {}

void bar(void) {
  int x = foo();  // BŁĄD: niezdefiniowane zachowanie
}
```

Funkcja może zawierać dowolną liczbę instrukcji **return** – ważne, żeby ich użycie poprawiało czytelność kodu. Przykładowo poniższy kod:

```
int foo(int x) {
  int result = 0;
    if (x > 0) {
      /* długie obliczenia */
        result = ...;
    }
    return result;
}
```

jest (prawdopodobnie) mniej czytelny niż:

```
int foo(int x) {
  if (x < 0) {
    return 0;
  }

  /* długie obliczenia */
    result = ...;
    return result;
}
```

W szczególności zastosowanie dwóch instrukcji `return` w powyższym przykładzie umożliwiło zmniejszenie liczby poziomów zagnieżdżenia kodu o jeden. Wielokrotne instrukcje `return` warto stosować w funkcjach o niewielkiej długości (i złożoności), w których od razu widać możliwe rezultaty działania.

Z użyciem instrukcji `return` można zwrócić z funkcji tylko pojedynczy obiekt. Chcąc zwrócić kilka wartości z funkcji musisz albo zwrócić obiekt będący strukturą (zob. rozdz. *18 Struktury i inne formy danych*), albo zmieniać wartości argumentów funkcji przekazanych z użyciem wskaźników (zob. rozdz. *11 Wskaźniki*).

10.2.3 Wywołanie funkcji

Aby wywołać funkcję (czyli wykonać instrukcje w ramach funkcji), należy w kodzie programu umieścić instrukcję składającą się z nazwy funkcji oraz nawiasów okrągłych, wewnątrz których podaje się wyrażenia oddzielone przecinkami – argumenty funkcji (czyli konkretne wartości parametrów, już bez podawania typów danych):

```
nazwa_funkcji(lista_argumentów);
```

na przykład

```
sumuj(3, 5);    // podczas wykonywania funkcji `a` przyjmie wartosc 3,
                //   natomiast `b` wyniesie 5
```

Aby wykorzystać wartość zwracaną przez funkcję `sumuj()`, możemy np. przypisać jej wartość do zmiennej:

```
int suma;
suma = sumuj(2, 3);
```

Jeśli funkcja nie wymaga przekazania żadnych wartości (np. `rand()`), po prostu zostawiamy nawiasy puste:

```
int r = rand();
```

Po zakończeniu wykonywania kodu wewnątrz wywoływanej funkcji komputer wraca do kolejnej instrukcji[1] albo do kolejnego fragmentu wyrażenia w funkcji wywołującej.

10.2.4 Parametr a argument

Parametr (ang. parameter) to zmienna, która pojawia się w nagłówku funkcji (np. przy okazji deklaracji albo definicji):

[1] Może być nią instrukcja wyrażeniowa.

```
void func(int a, float k);   // `a` i `k` to parametry
```

Parametry opisują typ danych przekazywanych do funkcji.

Argument (ang. argument) to wartość wyrażenia użytego podczas wywołania funkcji, która to wartość zostanie przypisana do danego parametru:

```
func(5, 0.03);  // 5 i 0.03 to argumenty
```

10.2.5 Przykład użycia funkcji

Przebieg wykonania programu konsolowego w języku C zostanie omówiony na poniższym przykładzie prostego programu:

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

// Deklaracja (inaczej ,,prototyp") funkcji.
int kwadrat(int n);

int main(void)
{
    int n = 3;

    // Wywolanie funkcji kwadrat(), zwrocona wartosc zostaje przekazana jako
    // argument funkcji printf()...
    printf("%d^2 = %d\n", n, kwadrat(n));

    return EXIT_SUCCESS;
}

// Definicja funkcji.
int kwadrat(int n) {
    // Podnies liczbe `n` do kwadratu i zwroc ta wartosc.
    return n * n;
}
```

- Wykonanie *każdego* programu konsolowego w języku C zaczyna się od pierwszej instrukcji funkcji `main()` – w powyższym przypadku zdefiniowana zostaje zmienna `n`.
- Następnie program natrafia na wywołanie funkcji `printf()` i aby ją wykonać, zaczyna obliczać wartości wyrażeń podanych jako *argumenty*. Trzecim argumentem jest wywołanie innej funkcji – `kwadrat()`.
- Sterowanie przechodzi więc do pierwszej instrukcji *definicji* funkcji `kwadrat()`. Wcześniej tworzona jest *nowa* zmienna `n`, która „przesłania" zmienną `n` utworzoną w funkcji `main()`. Oznacza to, że choć początkowa wartość zmiennej `n` utworzonej w funkcji `kwadrat()` wynosi tyle samo, co wartość zmiennej `n` utworzonej w funkcji `main()`, to jednak zmiana wartości `n` wewnątrz funkcji `kwadrat()` nie powoduje zmiany wartości zmiennej `n` z funkcji `main()`. (Przykład ilustrujący tą właściwość – poniżej.) Akurat tak się składa, że pierwszą (i jedyną) instrukcją funkcji `kwadrat()` jest instrukcja **`return`**. Wykonanie instrukcji **`return`** powoduje powrót sterowania do miejsca wywołania funkcji (a więc żadne dalsze instrukcje wewnątrz funkcji `kwadrat()` nie zostaną wykonane).
- Za wyrażenie `kwadrat(k)` zostanie podstawiona wartość zwrócona przez tę funkcję.
- Teraz wykonana zostanie funkcja `printf()` – w podobny sposób, co funkcja `kwadrat()`.
- Instrukcja **`return`** `EXIT_SUCCESS;` na końcu funkcji `main()` powoduje powrót sterowania do systemu operacyjnego. Zwraca wartość typu całkowitego oznaczającą status zakończenia programu.

Oto przykład ilustrujący fakt, że zmienne zadeklarowane wewnątrz jednej funkcji **przesłaniają** zmienne z zadeklarowane w miejscu wywołania funkcji:

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

```c
void zmien(int n) {
    int i = 4;
    n = 1;

    printf("ZMIEN:    i = %d, n = %d\n", i, n);
}

int main(void) {
    int n = 7;
    int i = 0;

    printf("MAIN (1): i = %d, n = %d\n", i, n);
    zmien(n);
    printf("MAIN (2): i = %d, n = %d\n", i, n);

    return EXIT_SUCCESS;
}
```

Wynik wykonania programu:

```
MAIN (1): i = 0, n = 7
ZMIEN:    i = 4, n = 1
MAIN (2): i = 0, n = 7
```

To między innymi ta własność funkcji sprawia, że można je traktować jak „czarne skrzynki”. Więcej o przesłanianiu przeczytasz w rozdziale *13 Klasy zmiennych*.

10.3 Deklarowanie i definiowanie funkcji a błędy budowania programu

Ponieważ kompilator analizuje kod jednostki translacji tylko raz (w kolejności od pierwszego do ostatniego wiersza) funkcja musi zostać zadeklarowana przed pierwszym użyciem jej identyfikatora, a dodatkowo musi zostać zdefiniowana w którymś miejscu w programie poza definicjami innych funkcji – próba zbudowania poniższego programu:

```c
#include <stdlib.h>

int main(void) {
    foo();  // PROBLEM: niejawna deklaracja funkcji `foo`

    return EXIT_SUCCESS;
}
```

spowoduje pojawienie się dwóch komunikatów o problemach:

- Na etapie *kompilacji* pojawi się *ostrzeżenie* „`implicit declaration of function 'foo'`”, gdyż (ze względu na kompatybilność wsteczną) standard wciąż dopuszcza pominięcie typu zwracanego przez funkcję w jej deklaracji[2] – zatem instrukcja

  ```c
  foo();  // brak deklaracji symbolu `foo`
  ```

  jest traktowana jako deklaracja funkcji `foo()` o niejawnym typie `int`.
- Na etapie *konsolidacji* pojawi się *błąd* „`undefined reference to 'foo'`”, gdyż owszem, funkcja `foo()` została *zadeklarowana* (niejawnie), jednak nie została nigdzie *zdefiniowana* – zatem konsolidator nie jest w stanie powiązać odniesień do tego symbolu (w tym przypadku: nie wie przykładowo, jaki kod wykonać w momencie potencjalnego wywołania funkcji).

Podobnie próba zbudowania poniższego programu:

```c
#include <stdlib.h>

void foo();  // BŁĄD: funkcja `foo` zadeklarowana, ale nie posiada definicji
```

[2] zob. rozdz. *4.5 Definicja i inicjalizacja zmiennej*

```
int main(void) {
    foo();

    return EXIT_SUCCESS;
}
```

spowoduje pojawienie się błędu konsolidacji „`undefined reference to 'foo'`”, gdyż funkcja `foo()` została (jawnie) zadeklarowana, ale nie została nigdzie w kodzie zdefiniowana.

Wreszcie, funkcja może zostać *zadeklarowana* w kodzie dowolną liczbę razy, ale musi zostać zdefiniowana *dokładnie raz* – próba zbudowania poniższego programu:

```
#include <stdlib.h>

void foo() {}
void foo() {}  // BŁĄD: redefinicja funkcji `foo()`

void bar();
void bar();
void bar() {}  // OK, powyżej tylko deklaracje

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

zakończy się błędem konsolidacji „`error: redefinition of 'foo'`”.

10.4 Rekurencja

Rekurencja, zwana także **rekursją** (ang. recursion), oznacza odwoływanie się funkcji (albo definicji) do samej siebie.

Oto przykład prostego programu, który wykorzystuje rekurencję:

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

void gora_i_dol(int n) {
    printf("Poziom %d\n", n);

    if (n < 4) {
        gora_i_dol(n + 1);
    }

    printf("POZIOM %d\n", n);
}

int main(void) {
    gora_i_dol(1);
    return EXIT_SUCCESS;
}
```

Efekt wykonania powyższego programu (zwróć uwagę na kolejność komunikatów):

```
Poziom 1
Poziom 2
Poziom 3
Poziom 4
POZIOM 4
POZIOM 3
POZIOM 2
POZIOM 1
```

Prześledźmy program, aby zobaczyć, jak działa rekurencja:

- Na początku funkcja `main()` wywołuje funkcję `gora_i_dol()`, przekazując jej jako argument wartość 1. W wyniku tego parametr `n` otrzymuje wartość 1, co powoduje wyświetlenie tekstu `Poziom 1` przez pierwszą instrukcję `printf()`.
- Następnie, ponieważ warunek `n < 4` jest spełniony, funkcja `gora_i_dol(poziom 1)` wywołuje funkcję `gora_i_dol(poziom 2)` z argumentem `n + 1`, czyli 2. Powoduje to przypisanie zmiennej `n` na drugim poziomie wartości 2 oraz wyświetlenie przez pierwszą instrukcję `printf()` komunikatu `Poziom 2`.
- W podobny sposób kolejne dwa wywołania prowadzą do wyświetlenia tekstów `Poziom 3` i `Poziom 4`.
- Gdy osiągnięty zostaje poziom 4, warunek `n < 4` przestaje być spełniony, przez co funkcja `gora_i_dol()` nie zostaje ponownie wywołana. Zamiast tego, funkcja poziomu 4 przechodzi do drugiej instrukcji pisania, która wyświetla tekst `POZIOM 4` (ponieważ `n` wynosi 4).
- Następnie program wykonuje instrukcję **`return`**. W tym momencie funkcja na poziomie 4 ulega zakończeniu, a program wraca do funkcji, która ją wywołała, czyli do funkcji `gora_i_dol()` poziomu 3.
- Ostatnią instrukcją wykonaną na poziomie 3 było wywołanie poziomu 4 w ramach instrukcji **`if`**. Poziom 3 wznawia więc działanie od kolejnej instrukcji, którą jest druga instrukcja `printf()`. Powoduje to wyświetlenie tekstu `POZIOM 3`.
- Następnie zakończeniu ulega poziom 3, program wraca do poziomu 2, który wyświetla tekst `POZIOM 2`, i tak dalej.

Oto zasady rekurencji (a zarazem każdego innego, nierekurencyjnego, wywołania funkcji):

- Każdy poziom rekurencji posiada swoje własne (niezależne od innych poziomów) zmienne – bo wywoływana jest nowa funkcja.
- Każdemu wywołaniu funkcji odpowiada jeden powrót. Po wykonaniu instrukcji **`return`** na końcu ostatniego (najgłębszego) poziomu rekurencji, program przechodzi do poprzedniego poziomu rekurencji (w miejsce tuż po wywołaniu funkcji) – a nie do funkcji `main()`.
- Po powrocie do miejsca wywołania funkcji wykonywane są dalsze instrukcje znajdujące się w funkcji.
- Chociaż każdy poziom rekurencji posiada swój własny zestaw zmiennych, kod funkcji nie jest zwielokrotniany (tj. w pamięci operacyjnej znajduje się tylko jeden „egzemplarz" kodu funkcji). Po prostu podczas kolejnego wywołania tej samej funkcji program ponownie przechodzi na jej początek, tyle że z nowym zestawem zmiennych.

Powyższe zasady są *cechą języka C* (i większości stosowanych obecnie języków programowania).

Ważne

Jeśli do wyboru mamy napisać pętlę albo użyć rekurencji, to ze względów wydajności lepiej użyć pętli. Rekurencja wymaga o wiele więcej pamięci na dodatkowe zestawy zmiennych! Natomiast w niektórych przypadkach użycie rekurencji jest dużo prostsze i bardziej intuicyjne – m.in. gdy sam problem jest sformułowany w sposób rekurencyjny.

Przykład: obliczanie potęgi liczby

Znanym z matematyki przykładem definicji rekurencyjnej jest definicja potęgi liczby naturalnej o wykładniku naturalnym:

$$a^n := \begin{cases} 1 & \text{dla } n = 0; \\ a \cdot a^{n-1} & \text{dla } n > 0. \end{cases} \qquad a, n \in \mathbb{N}$$

W poniższym programie demonstrującym działanie funkcji obliczającej potęgę w sposób rekurencyjny umieszczono dodatkowe „diagnostyczne" fragmenty kodu umożliwiające śledzenie zmian wartości zmiennych.
Zmienna `poziom` oznacza poziom rekurencji, czyli który raz z kolei funkcja wywołała samą siebie.

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

```
int potega(int a, int n, int poziom);

int main(void) {
    int a = 3;
    int n = 2;

    printf("\n\n %d^%d = %d\n", a, n, potega(a, n, 1));

    return EXIT_SUCCESS;
}

int potega(int a, int n, int poziom) {
    int p;

    printf("%*c poziom %d: n = %d\n", poziom + 1, ' ', poziom, n);

    if (n == 0) {
        p = 1;
    } else {
        p = a * potega(a, n - 1, poziom + 1);
    }

    printf("%*c poziom %d: p = %d\n", poziom + 1, ' ', poziom, p);

    return p;
}
```

Wynik wykonania powyższego programu:

```
   poziom 1: n = 2
    poziom 2: n = 1
     poziom 3: n = 0
     poziom 3: p = 1
    poziom 2: p = 3
   poziom 1: p = 9


 3^2 = 9
```

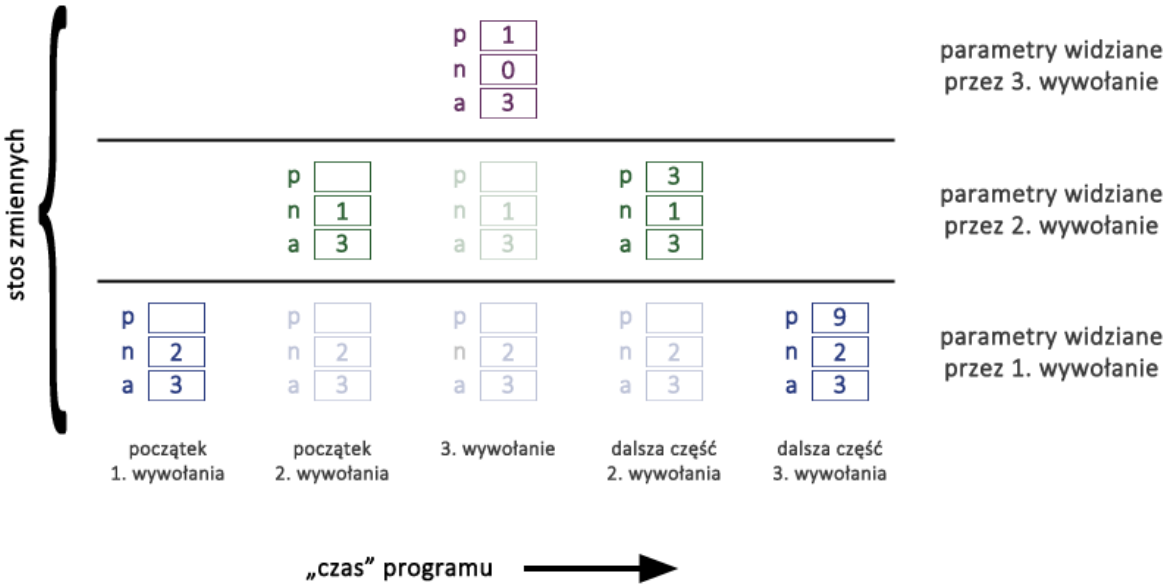
Rysunek 10.1 ilustruje zmianę zmiennych podczas wykonania powyższego programu (szare ramki oznaczają zmienne z wyższego poziomu *przesłonięte* przez zmienne z niższego poziomu wywołań funkcji).

Używając notacji matematycznej rekurencyjne obliczanie potęgi można rozpisać następująco (cyfry rzymskie oznaczają poziom rekurencji):

$$\text{potega}(a=3, n=2): \quad 3^2 \overset{I}{=} 3 \cdot a^1 \overset{II}{=} 3 \cdot (3 \cdot a^0) \overset{III/II}{=} 3 \cdot (3 \cdot \underbrace{1}_{a^0}) \overset{I}{=} 3 \cdot \underbrace{3}_{a^1} \overset{I}{=} 9 \rightarrow \text{main()}$$

Jak projektować funkcję rekurencyjną?

Rekurencja oznacza po prostu to, że funkcja wywołuje samą siebie. Musi jednak istnieć przypadek, w którym funkcja ta *nie* wywoła samej siebie – w przeciwnym razie program wpadnie w nieskończony cykl wywołań, wyczerpie dostępny limit pamięci na wywołania funkcji i... zostanie „ubity” przez system operacyjny. Przypadek, w którym funkcja nie wywołuje samej siebie, nazywamy **przypadkiem podstawowym** (w przykładzie z potęgowaniem był to przypadek dla $n = 0$). Projektowanie funkcji rekurencyjnej sprowadza się więc do wyboru rozsądnego przypadku podstawowego, a następnie upewnieniu się, że każdy ciąg wywołań funkcji w końcu osiągnie ten przypadek.



Rysunek 10.1. Zawartość stosu podczas wywołania rekurencyjnego funkcji

Rozdział 11

Wskaźniki

11.1 Czym jest wskaźnik?

Wskaźnik (ang. pointer) to obiekt danych, którego wartość oznacza *adres* miejsca w pamięci komputera (a nie przechowywaną pod tym adresem wartość) – przy czym adres to liczba całkowita dodatnia, jednoznacznie definiująca położenie pewnego obszaru w pamięci komputera[1].

Obrazowo możemy wyobrazić sobie pamięć komputera jako bibliotekę a zmienne jako książki. Zamiast brać książkę z półki samemu (analogicznie do korzystania wprost ze zwykłych zmiennych), możemy podać bibliotekarzowi wypisany rewers z numerem katalogowym książki a on znajdzie ją za nas. Analogia ta nie jest doskonała, ale pozwala wyobrazić sobie niektóre cechy wskaźników:

- numer na rewersie identyfikuje pewną książkę,
- kilka rewersów może dotyczyć tej samej książki,
- możemy skreślić numer na rewersie i podać napisać numer w celu zamówienia innej książki,
- jeśli wpiszemy nieprawidłowy numer, to możemy dostać nie tę książkę, którą chcemy, albo też nie dostać nic.

11.2 Definiowanie wskaźników

Definicja wskaźnika ma ogólną postać:

```
<typ_wskazywanego_obiektu_danych>* nazwa_wskaznika;
```

W tym przypadku znak `*` informuje kompilator, że chodzi o zmienną wskaźnikową, a nie zwykłą zmienną. Oto przykładowa definicja „wskaźnika do zmiennej typu **`char`**” (albo w skrócie: „wskaźnik do **`char`**”):

```
char* pc;
```

Nie wystarczy stwierdzić, że zmienna jest wskaźnikiem – należy również określić, na jaki typ zmiennej będzie ona wskazywała. Powodem tego jest fakt, iż zmienne różnych typów zajmują różne ilości pamięci, a niektóre operacje na wskaźnikach wymagają wiedzy o rozmiarze wskazywanej zmiennej. Ponadto, program powinien wiedzieć, jaki rodzaj danych jest przechowywany pod określonym adresem – przykładowo, typy **`long`** i **`float`** zajmują na niektórych systemach tyle samo pamięci, ale wyrażają liczby w zupełnie różny sposób.

Zwróć uwagę, że jeśli napiszesz

```
int* a, b, c;
```

to otrzymasz nie trzy wskaźniki na **`int`**, tylko jeden wskaźnik na **`int`** (zmienna `a`) oraz dwie zmienne typu **`int`** (o identyfikatorach `b` i `c`). Aby faktycznie zdefiniować trzy wskaźniki na **`int`** należy napisać

```
int *a, *b, *c;
```

Jak widzisz, aby uniknąć pomyłek, lepiej definiować zmienne pojedynczo.

[1] To stwierdzenie jest prawdziwe m.in. dla komputerów z mikroprocesorem w architekturze x86, z którymi stykasz się na co dzień, natomiast w przypadku innych architektur model pamięci może się różnić.

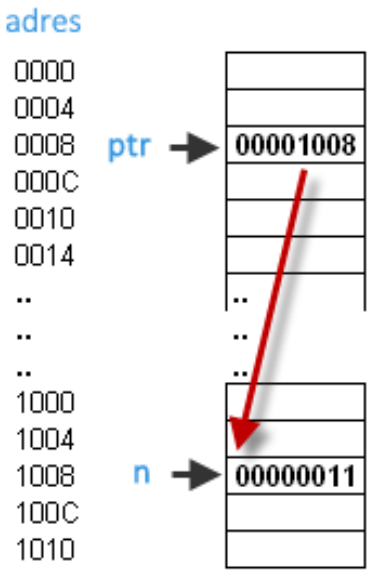
11.3 Uzyskiwanie adresów: operator &

Adres, pod którym w pamięci znajduje się zmienna, można uzyskać używając jednoargumentowego operatora `&` (jeśli `x` jest nazwą zmiennej, to `&x` jest jej adresem).

Zatem przypisanie wskaźnikowi adresu odbywa się w następujący sposób:

```
int n = 11;
int* ptr = &n;
```

Mówimy, że wskaźnik `ptr` „*wskazuje na*” zmienną `n`. Rysunek 11.1 pokazuje, jak wygląda pamięć komputera po takiej operacji[2].



Rysunek 11.1. Wartość zmiennej wskaźnikowej

11.4 Zmiana wartości wskazywanych: operator dereferencji *

Aby uzyskać wartość przechowywaną pod danym adresem, korzystamy z jednoargumentowego operatora **dereferencji** `*` (ang. dereference), zwanego w żargonie **operatorem wyłuskiwania**:

```
int n = 3;
int* p_int = &n;
printf("adres = %p , wartosc = %d\n", (void*) p_int, *p_int);
```

Ważne

Nie pomyl jednoargumentowego operatora dereferencji z dwuargumentowym operatorem arytmetycznym mnożenia – oba oznaczane są tym samym symbolem `*`.

Operator dereferencji `*` jest **operatorem odwrotnym** dla operatora adresu `&`, zatem `p = *&q;` jest równoważne `p = q;`

Wartość wskazywaną można zmienić w następujący sposób:

```
int n = 1;
int* ptr = &n;
*ptr = 3;
// wartosc `n` wynosi teraz 3
```

Oto jeszcze jeden przykład na zmianę przeprowadzoną w identyczny sposób:

[2]Przy założeniu, że adres zmiennej `n` to 0x1008, a adres zmiennej `ptr` to 0x008.

```
int i = 1;
int j = 2;
int* p = &i;
int* q = p;  // obie zmienne `p` i `q` wskazuja na `i`

q = &j; // teraz `p` i `q` wskazuja na rozne zmienne

*q = *p; // przypisz wartosc wskazywana przez `p` w miejsce wskazywane
         //   przez `q` - wartosc `j` wynosi teraz 1
```

Ważne

Przypisanie `q = p` nie jest tym samym, co `*q = *p`! Pierwsze przypisanie sprawi, że oba wskaźniki będą wskazywać na to samo miejsce w pamięci, natomiast drugie – że w miejscu wskazywanym przez `q` znajdzie się wartość wskazywana przez `p`.

11.5 Wyświetlanie wskaźników

Do wyświetlenia wartości wskaźnika – a więc adresu w pamięci – najlepiej użyć specyfikatora konwersji `%p`, który wyświetla wartość w kodzie szesnastkowym (taka zwykło się podawać adresy)[3]:

```
int n = 5;
int* ptr = &n;

printf("n:   wartosc = %d , adres = %p\n", n, (void*) &n);
printf("ptr: wartosc = %p , adres = %p\n", (void*) ptr, (void*) &ptr);
```

Specyfikator `%p` wymaga wartości typu `void*`. Dlaczego? Otóż „wskaźnik na typ `void`" to specjalny „uniwersalny" typ wskaźnika – może wskazywać na dane dowolnego typu, jednak z tego samego powodu nie ma możliwości jego dereferencji (należy najpierw zrzutować go do pożądanego typu)[4].

11.6 Wskaźnik pusty

Wskaźnik pusty albo inaczej **wskaźnik zerowy** (ang. null pointer) to wskaźnik, którego wartość wynosi 0. Ponieważ *poprawne* adresy pamięci zaczynają się od 1, wskaźnik pusty służy do sygnalizowania pewnych szczególnych sytuacji (np. podczas dynamicznej alokacji pamięci[5]) oraz jako rozsądna domyślna wartość deklarowanych wskaźników.

W celu poprawy czytelności kodu zamiast przypisywać do wskaźnika wartość 0 wprost (jako liczbę magiczną[6]), lepiej zastosować stałą symboliczną `NULL` z biblioteki `stdlib.h`.

11.7 Pułapki wskaźników

Ważne jest, aby przy posługiwaniu się wskaźnikami nigdy nie próbować odwoływać się do komórki wskazywanej przez wskaźnik o wartości `NULL` oraz aby nie używać niezainicjowanego wskaźnika:

```
int *wsk;
printf ("zawartosc komorki: %d\n", *(wsk));    /* BŁĄD */
wsk = NULL;
printf ("zawartosc komorki: %d\n", *(wsk));    /* BŁĄD */
```

Pamiętaj też, że operator `sizeof` użyty na zmiennej wskaźnikowej (tj. `sizeof(ptr)`) zwraca rozmiar *adresu*, a nie rozmiar typu obiektu wskazywanego przez wskaźnik. Wielkość ta będzie zawsze miała taki sam rozmiar dla każdego wskaźnika i będzie zależeć od kompilatora i docelowej platformy. Jeśli chcesz uzyskać rozmiar typu wskazywanego, użyj `sizeof(*ptr)`:

[3] Choć adres to na większości komputerów po prostu liczba całkowita dodatnia, użycie specyfikatora `%d` lub `%u` może nie być poprawne ze względu na różnice w długościach typów – dlatego lepiej stosować standardowy specyfikator `%p`.

[4] Wskaźnik na `void` będzie wykorzystywany w przypadku dynamicznej alokacji pamięci opisanej w rozdziale *15 Zarządzanie pamięcią*.

[5] zob. rozdz. *15 Zarządzanie pamięcią*

[6] zob. rozdz. *4.7 Stałe*

```
char *zmienna;
int z = sizeof(zmienna);  // z może być równe 4 (rozmiar adresu na maszynie
                          //   32-bitowej)
z = sizeof(char*);        // robimy to samo, co wyżej
z = sizeof(*zmienna);     // tym razem z to rozmiar znaku (tj. 1)
z = sizeof(char);         // robimy to samo, co wyżej
```

11.8 Wykorzystanie wskaźników do komunikacji pomiędzy funkcjami

Przekazywanie wskaźników do zmiennych jako parametrów funkcji umożliwia zmianę wartości tych zmiennych podczas wykonywania funkcji. Ten sposób przekazywania argumentów do funkcji jest nazywany **przekazywaniem przez wskaźnik** (ang. pass-by-pointer) w przeciwieństwie do domyślnego **przekazywania przez wartość** (ang. pass-by-value).

Ważne

Argumenty do funkcji przekazujemy przez wartość, chyba że występuje konieczność zmiany jej wartości wewnątrz funkcji – wówczas przekazujemy przez wskaźnik.
W przypadku tablic *musimy* przekazać wskaźnik – to wymóg języka C[7].

Oto przykład funkcji rozkładającej liczbę rzeczywistą na część całkowitą i ułamkową:

```
void rozloz(double x, int* czesc_calkowita, double* czesc_ulamkowa)
{
    *czesc_calkowita = (int) x;
    *czesc_ulamkowa = x - *czesc_calkowita;
}
```

Jej wywołanie wygląda następująco:

```
double x = 3.14;
int i = 0;
double f;

rozloz(x, &i, &f);  // zwróć uwagę, że przekazywane są adresy zmiennych
                    // a nie ich wartości

printf("%.2f = %d + %.2f\n", x, i, f);  // 3.14 = 3 + 0.14
```

Ważne

Zwróć uwagę, że ponieważ wartość wskaźnika jest liczbą całkowitą, wywołanie `rozloz(x, i, &f);` też będzie poprawne – jednak wówczas funkcja użyłaby liczby 0 jako adresu (jako wartości parametru `czesc_calkowita`) i starałaby się zmienić komórkę pamięci o numerze 0, co z pewnością doprowadziłoby do błędów wykonania programu. Kompilator powinien ostrzec w takim przypadku o konwersji z typu `int` do wskaźnika, ale jeśli zignorujesz ostrzeżenie, Twój program prawdopodobnie zakończy się z komunikatem o błędzie.

Zauważ, że chcąc wewnątrz funkcji zmienić wartość *wskaźnika* (a nie wartość wskazywaną) w taki sposób, aby zmiana była widoczna po powrocie z takiej funkcji, musisz jako argument przekazać adres tego wskaźnika – czyli parametrem funkcji musi być *wskaźnik do wskaźnika*:

```
void increment_wrong(int* ptr) {
    ++ptr;  // modyfikacja adresu wskazywanego przez KOPIĘ wskaźnika
            //   przekazanego jako argument
}
void increment_coorrect(int** ptr) {
```

[7] Przyczyną jest wydajność. Gdyby funkcja przekazywała wartość tablicy, musiałaby zarezerwować wystarczająco dużo miejsca, aby zmieścić duplikat tablicy wyjściowej, a następnie skopiować wszystkie dane z oryginału do kopii. O wiele szybszą metodą jest przekazanie adresu tablicy i wykonywanie operacji na tablicy pierwotnej.

```
    ++(*ptr);  // modyfikacja adresu wskazywanego przez wskaźnik przekazany
               //   jako argument
}

int main(void) {
    int arr[] = {1, 2};
    int* ptr1 = arr;
    int* ptr2 = arr;

    increment_wrong(ptr1);
    increment_coorrect(&ptr2);

    printf("tab  = %p\n", (void*) arr);
    printf("ptr1 = %p\n", (void*) ptr1);
    printf("ptr2 = %p\n", (void*) ptr2);

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
tab  = 000000000022fe40
ptr1 = 000000000022fe40
ptr2 = 000000000022fe44
```

Zwróć uwagę, że w przeciwieństwie do wartości wskaźnika `ptr2` wartość wskaźnika `ptr1` (czyli wskazywany adres) nie uległa zmianie – wskaźnik ten wciąż wskazuje na początek tablicy `arr`.

11.9 Wskaźniki a `const`

W przypadku wskaźników mamy do czynienia z dwoma elementami, które mogą być stałe (są to: wskazywany obiekt oraz sam wskaźnik), a zatem istnieją cztery możliwe kombinacje użycia **`const`**:

```
T* p;              // "zwykły" wskaźnik:
                   //   można zarówno zmienić stan wskazywanego obiektu,
                   //   jak i przepiąć wskaźnik na inny obiekt

const T* p;        // wskaźnik na stałą wartość:
                   //   nie można zmienić stanu wskazywanego obiektu,
                   //   natomiast można przepiąć wskaźnik na inny obiekt

T* const p;        // stały wskaźnik:
                   //   można zmienić stan wskazywanego obiektu,
                   //   lecz nie można przepiąć wskaźnika na inny obiekt

const T* const p;  // stały wskaźnik na stałą wartość:
                   //   nie można zmienić stanu wskazywanego obiektu,
                   //   oraz nie można przepiąć wskaźnika na inny obiekt
```

Pamiętaj, że deklaracje wskaźników czytamy od prawej do lewej: **`const`** `T*` `p` oznacza, że „`p` to wskaźnik na stały typ `T`”.

Szczególnie często wykorzystywane są wskaźniki na stałą wartość, gdyż pozwalają wydajnie przekazywać duże obiekty (np. złożone struktury danych) do funkcji nawet, jeśli dana funkcja nie modyfikuje argumentów – rozmiar wskaźnika jest stały (nie zależy od rozmiaru wskazywanego typu), a użycie kwalifikatora **`const`** uniemożliwia modyfikację wskazywanego obiektu:

```
// `T_large` to hipotetyczny typ o dużym rozmiarze
//   (np. złożona struktura danych)
void funkcja(const T_large* ptr) {
  // wartość wskazywana przez `ptr` jest TYLKO DO ODCZYTU
}
```

To szczególnie istotne w przypadku systemów wbudowanych z małą ilością pamięci operacyjnej – przekazywanie przez wskaźnik pozwala uniknąć niepotrzebnego kopiowania obiektów i tym samym pozwala oszczędzić miejsce w pamięci (oraz czas potrzebny na kopiowanie).

11.10 Zasady ścisłego aliasingu

Zasady **ścisłego aliasingu** (ang. strict aliasing) określają, kiedy w przypadku obiektu o efektywnym typie `T1` użycie go w l-wyrażeniu (zwykle: dereferencji wskaźnika) o innym typie `T2` spowoduje niezdefiniowane zachowanie (na razie przyjmij, że efektywny typ odpowiada typowi użytemu w deklaracji; dokładne zasady określania typu efektywnego zostaną omówione w rozdz. 15.7). Dzieje się tak wówczas, gdy *nie zachodzi* żadna z poniższych sytuacji[8]:

- `T2` i `T1` są typami odpowiadającymi sobie[9], przy czym
  - `T2` może posiadać kwalifikator(y) **`const`**
  - nie ma znaczenia, czy są to typy ze znakiem, czy bez znaku
- `T2` jest typem znakowym.

Przykładowo:

```
int i = 7;
float* pf = (float*)(&i);
float d = *pf;  // NIEZDEFINIOWANE ZACHOWANIE: l-wyrażenie `*p` typu `float`
                //   nie może zostać użyte do uzyskania dostępu do wartości
                //   typu `int`
```

[8] W rzeczywistości istnieje jeszcze jedna sytuacja związana ze strukturami i uniami, jednak jest ona dość specjalistyczna i dlatego została pominięta w tym zestawieniu.

[9] Dla uproszczenia przyjmij, że `T2` i `T1` są dokładnie tym samym typem. O typach odpowiadających poczytasz tu.

Rozdział 12

Tablice

W tym rozdziale nauczysz się tworzyć i inicjalizować tablice – ciągi wartości tego samego typu (np. liczb) przechowywanych obok siebie – poszerzając przy okazji swoją wiedzę o wskaźnikach i ich związkach z tablicami. Dowiesz się, w jaki sposób pisać funkcje przetwarzające tablice, oraz przyjrzysz się tablicom dwuwymiarowym.

12.1 Tablice jednowymiarowe

Tablice jednowymiarowe służą do reprezentowania danych tego samego typu, które możemy przedstawić w postaci jednego wiersza (lub równoważnie: jednej kolumny), np. wyniki pomiaru prędkości samochodu w równych odstępach w czasie.

12.1.1 Deklarowanie

Ogólna forma definicji tablicy ma postać:

```
typ_danych nazwa_tablicy[liczba_elementow];
```

przy czym dostępne typy elementów są identyczne, jak dla „zwykłych” zmiennych (`int`, `float` itd.)

Przykład:

```
int tab[8]; // tablica osmiu wartosci typu calkowitego
```

12.1.2 Inicjalizowanie

Inicjalizacja tablicy polega na przypisaniu do tablicy podanej w nawiasach klamrowych listy wartości kolejnych elementów (rozdzielonych przecinkami).

Przykład:

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

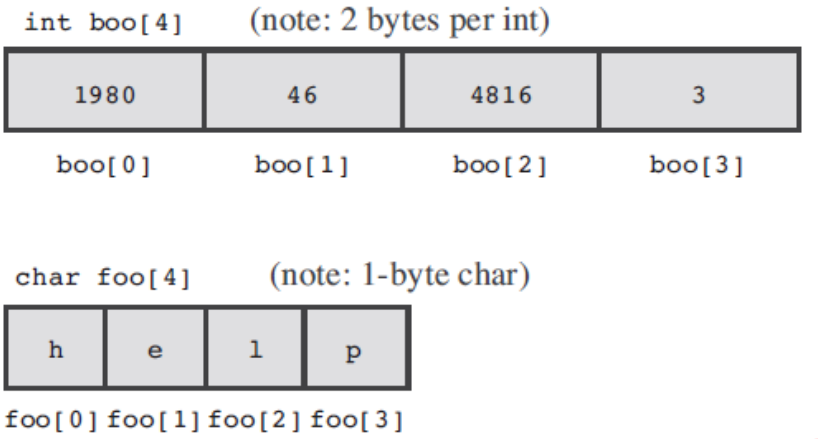
W przypadku podania zbyt małej liczby wartości do inicjalizacji, pozostałe elementy tablicy otrzymują wartość 0:

```
int tab[4] = {3, 2, 5};  // tab[3] ma wartość 0
```

12.1.3 Dostęp do elementów

Liczby identyfikujące elementy tablicy nazywamy **indeksami** (ang. index), przesunięciami lub offsetami. Indeksy muszą być liczbami naturalnymi, przy czym pierwszemu elementowi odpowiada zawsze indeks 0. Elementy tablicy znajdują się obok siebie w pamięci komputera, co ilustruje rys. 12.1.

Dostęp do poszczególnych elementów tablicy można uzyskać za pomocą indeksu elementu, będącego zawsze liczbą naturalną, umieszczonego w nawiasach kwadratowych.



Rysunek 12.1. Tablice typu `int` i `char` w pamięci

Ważne

W języku C indeksowanie zaczyna się od zera – czyli wyrażenie `tab[0]` odwołuje się do wartości pierwszego elementu tablicy!

Oto przykład uzyskiwania dostępu do elementu tablicy:

```
int tab[] = {1, 2, 3, 4};
int elem = tab[2];  // przypisz zmiennej `elem` wartość TRZECIEGO
                    //   elementu tablicy `tab`
```

Język C nie pozwala ani na przypisywanie tablic w całości:

```
int tab1[] = {1, 2, 3, 4};
int tab2[] = tab1;  // BŁĄD
```

ani na odwoływanie się naraz do więcej niż jednego elementu tablicy (np. w celu zmiany wartości takiego zakresu tablicy).

12.2 Dlaczego nie należy wychodzić poza zakres tablicy?

W poniższym przykładzie mimo, że tablica składa się z trzech elementów (a więc poprawne indeksy to 0, 1 i 2), możemy próbować odwołać się do elementów spoza zakresu – ani kompilator nie zgłosi błędów, ani (w tym przypadku) nie wystąpi błąd podczas wykonania programu:

```
#include <stdio.h>

int main()
{
    int tab[3] = {1, 2, 3};

    printf("%d\n", tab[-1]);
    printf("%d\n", tab[3]);

    return 0;
}
```

Tyle, że. . . wartości `tab[-1]` i `tab[3]` zawierają „śmieci". W dodatku ta pamięć nie należy już do zmiennej `tab`.

Ostrzeżenie

W ogólnym przypadku odwoływanie się do nieswojej pamięci skutkuje niezdefiniowanym zachowaniem programu.

12.3 Współpraca tablicy i pętli `for`

Niestety, język C nie pozwala na wyświetlanie tablicy z użyciem pojedynczej instrukcji – tablicę trzeba wyświetlać element po elemencie, najlepiej z użyciem pętli **`for`**:

```
#define __USE_MINGW_ANSI_STDIO 1

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

#define ARRAY_SIZE 3

int main(void) {
    int arr[ARRAY_SIZE] = {1, 2, 3};
    for (size_t i = 0; i < ARRAY_SIZE; ++i) {
        printf("arr[%zu] = %d\n", i, arr[i]);
    }
    return EXIT_SUCCESS;
}
```

Powyższy przykład ilustruje kilka zagadnień dotyczących stylu:

- Po pierwsze, warto użyć dyrektywy **`#define`**, aby utworzyć stałą symboliczną (w tym przypadku `ARRAY_SIZE`), określającą rozmiar tablicy. Stała ta przydaje się zarówno w deklaracji tablicy, jak i do określenia granic pętli. Jeśli zaistnieje potrzeba rozszerzenia programu tak, aby obsługiwał 20 wyników, wystarczy zmienić definicję stałej symbolicznej `ARRAY_SIZE` na 20 – nie ma natomiast konieczności dokonania zmian we wszystkich miejscach programu, w których użyty został rozmiar tablicy.
- Po drugie, następujący idiom programistyczny:

  ```
  for (size_t i = 0; i < n; i++)
  ```

  jest bardzo przydatny do przetwarzania tablicy o rozmiarze n elementów. Ważne jest przy tym, aby ustalić właściwy zakres wartości licznika pętli. Pierwszy element tablicy ma indeks 0 – tyle też wynosi początkowa wartość zmiennej `i`. Ponieważ numerowanie rozpoczyna się od zera, indeks ostatniego elementu jest równy $n - 1$ – odzwierciedla to warunek testowy `i < n`, zgodnie z którym ostatnią wartością zmiennej `i` wykorzystywaną wewnątrz pętli jest właśnie $n - 1$.

12.4 Jak „elastycznie” określać rozmiar tablicy: tablice bezwymiarowe

Jeśli definiujemy tablicę z użyciem listy, możemy nie podawać jej rozmiaru, tylko po prostu pozostawić puste nawiasy kwadratowe – kompilator sam automatycznie dobierze rozmiar tablicy tak, aby zmieściły się w niej wszystkie elementy z listy:

```
int tab1[] = {1, 2, 3};      // tablica tab1 ma rozmiar 3 elementów
int tab2[] = {1, 2, 3, 4};   // tablica tab2 ma rozmiar 4 elementów
```

12.5 Jak „elastycznie” określać rozmiar tablicy: operator `sizeof`

Standard języka C *nie pozwala* na używanie zmiennych albo stałych do określania rozmiaru tablicy podczas deklaracji:

```
int n = 3;
int tab[n];   // BŁĄD
```

```
const int N = 3;
int tab[N];   // BŁĄD
```

(Powyższy mechanizm określania rozmiaru z użyciem zmiennych i „zwykłych” stałych został wprowadzony w standardzie C99, jednak w standardzie C11 został on oznaczony jako opcjonalny – dlatego w tym przypadku lepiej trzymać się mechanizmów opisanych w standardzie C90)[1].

[1] Szczegóły znajdziesz tu.

Mimo to wygodnie jest mieć w programie zdefiniowaną w jednym miejscu w programie wartość określającą liczbę elementów tablicy – aby później korzystać z niej np. w pętlach (i uczynić program elastycznym). Dobrze, aby wartość ta była określana automatycznie – zmniejszymy w ten sposób ryzyko błędów. Z pomocą przychodzi operator **sizeof**:

```
int tab[5];
int liczba_elementow = sizeof(tab) / sizeof(tab[0]);
// liczba elementow = rozmiar całej tablicy (w bajtach) / rozmiar
//   pojedynczego elementu (w bajtach)
```

12.6 Wskaźniki a tablice

Ważne

Nazwa tablicy jest równocześnie adresem jej pierwszego elementu – stąd jeśli `tab` jest tablicą, prawdziwe jest wyrażenie `tab == &tab[0]` (dla przypomnienia: `&` jest operatorem adresu).

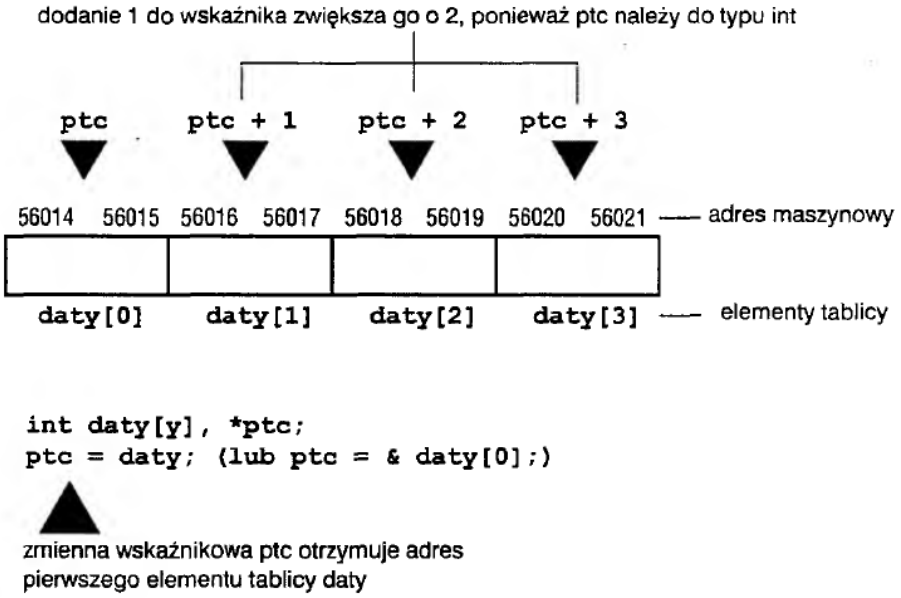
Zwróć uwagę na to, co dzieje się z wartością wskaźnika, gdy zostaje do niego dodana liczba:

```
/* wsk_dod.c - dodawanie do wskaźników */
#include <stdlib.h>
#include <stdio.h>

#define N 3

int main(void) {
    short arr_short[N];
    short* ptr_short = arr_short;
    double arr_double[N];
    double* ptr_double = arr_double;

    printf("%25s %16s\n", "short", "double");
    for (int index = 0; index < N; index++)
        printf("ptr + %d: %10p %10p\n", index,
               (void*) (ptr_short + index),
               (void*) (ptr_double + index));

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
                    short            double
ptr + 0: 000000000022FE32 000000000022FE10
ptr + 1: 000000000022FE34 000000000022FE18
ptr + 2: 000000000022FE36 000000000022FE20
```

Adresy odnoszą się do pojedynczych bajtów, ale trzeba pamiętać, że typ **short** zajmuje co najmniej 2 bajty, a typ **double** – co najmniej 4 bajty (w powyższym przykładzie – 8 bajtów). W wyniku wydania polecenia „dodaj 1 do wskaźnika” język C dodaje do adresu liczbę bajtów zajmowanych przez pojedynczy obiekt wskazywanego typu. W przypadku tablicy oznacza to, że bieżąca wartość wskaźnika (adres) zostanie zmieniona na adres następnego *elementu*, a nie następnego *bajtu* (zob. rys. 12.2). To właśnie dlatego w deklaracji wskaźnika należy określić typ wskazywanego obiektu: typ daje bowiem informację o rozmiarze, a rozmiar – o liczbie bajtów, o jaką należy zwiększyć adres, aby otrzymać adres następnego elementu tablicy. (Konieczność określenia typu dotyczy także zmiennych skalarnych, w przeciwnym wypadku niemożliwe byłoby poprawne pobranie wartości za pomocą wyrażenia w rodzaju `*ptr` – komputer nie wiedziałby, ile bajtów odczytać.)



Rysunek 12.2. Dodawanie do wskaźnika

Istnieje kilka rzeczy, o których należy pamiętać przy zwiększaniu lub zmniejszaniu wskaźnika:

- Komputer nie sprawdza, czy po wykonaniu operacji arytmetycznej na wskaźniku wciąż znajdujemy się w obszarze pamięci tablicy! Programista musi o to zadbać sam.
- Możliwe jest znalezienie różnicy między dwoma wskaźnikami. Działanie to jest wykonywane na wskaźnikach do elementów tej samej tablicy (lub na miejsce w pamięci następujące bezpośrednio po tej tablicy) w celu określenia, jak daleko od siebie się znajdują (ile elementów znajduje się pomiędzy tymi wskaźnikami, a *nie* ile bajtów dzieli te dwa adresy). Odejmowanie jest prawidłową operacją, o ile oba wskaźniki wskazują na elementy tej samej tablicy – zastosowanie go do wskaźników do dwóch różnych tablic może spowodować błąd programu.

Nazwa tablicy jest równocześnie adresem jej pierwszego elementu, stąd:

- `tab + 2 == &tab[2]` – ten sam adres
- `*(tab + 2) == tab[2]` – ta sama wartość

Ważne

Ze względu na kolejność operatorów wyrażenie „`*(tab + i)`” różni się od „`*tab + i`”:

- `*(tab + i)` – wartość $(i+1)$-tego elementu
- `*tab + i` – wartość pierwszego elementu zwiększona o i

Powyższe zależności podsumowują ścisły związek pomiędzy tablicami a wskaźnikami. Oznaczają one, że wskazania pojedynczego elementu tablicy i pobrania jego wartości można dokonać za pomocą zarówno indeksu, jak i wskaźnika. W gruncie rzeczy mamy bowiem do czynienia z dwoma różnymi zapisami o identycznym działaniu. Ponieważ standard języka C w opisie notacji tablicowej korzysta z pojęcia wskaźnika, podejście wskaźnikowe można uznać za bardziej elementarne. Z punktu widzenia C notacja tablicowa i wskaźnikowa są równoważne, ale użycie notacji tablicowej pokazuje *intencję* programisty.

Ważne

Standard języka C nie definiuje zachowania programu w sytuacji, gdy skonstruujesz wskaźnik wskazujący na miejsce w pamięci gdzieś poza zadeklarowaną tablicę. Wyjątkiem jest wskaźnik na miejsce tuż za ostatnim elementem tablicy (ang. one past last).

12.7 Tablice wielowymiarowe

Tablice wielowymiarowe są naturalnym rozszerzeniem tablic jednowymiarowych. Przydają się do reprezentowania macierzy oraz danych tabelarycznych (takich, w których np. kolumny oznaczają serie danych) w których elementy są tego samego typu.

12.7.1 Definiowanie

Uogólniona definicja tablicy wielowymiarowej ma postać:

```
typ_danych nazwa_tablicy[rozmiar_1][rozmiar_2]...[rozmiar_n];
```

na przykładzie dwuwymiarowym:

```
typ_danych nazwa_tablicy[ilosc_wierszy][ilosc_kolumn];
```

Przykład:

```
int tab[8][3]; // tablica o ośmiu wierszach i trzech kolumnach,
               //   wartości typu całkowitego ze znakiem
```

Komputer przechowuje w pamięci taką tablicę dwuwymiarową liniowo, jako kilka następujących po sobie tablic jednowymiarowych.

12.7.2 Inicjalizacja

Tablicę wielowymiarową można zainicjalizować na dwa sposoby:

- używając „zagnieżdżonych” klamer (pierwszy zestaw – pierwszy wiersz, drugi zestaw – drugi wiersz itd.)
- stosując „płaską” inicjalizację – wszystkie wartości znajdują się w jednych klamrach, przypisywane są do kolejnych elementów wierszami (najpierw całkowicie zapełniany jest pierwszy wiersz, potem drugi itd.)

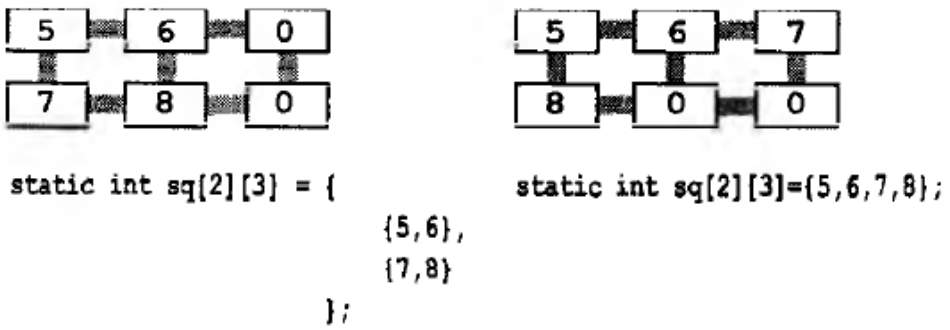
Oto przykład sposobu z „zagnieżdżaniem” klamer:

```
int tab[2][3] = {
        {3, 2, 6},
        {5, 1, 9}
    };
```

Oto przykład sposobu „płaskiego":

```
int tab[2][3] = {3, 2, 6, 5, 1, 9};  // zdecydowanie mniej czytelne,
                                      // w miare mozliwosci unikac!
```

Rysunek 12.3 przestawia co się dzieje, gdy liczba elementów w liście inicjalizacyjnej tablicy wielowymiarowej jest mniejsza od liczby elementów tablicy.



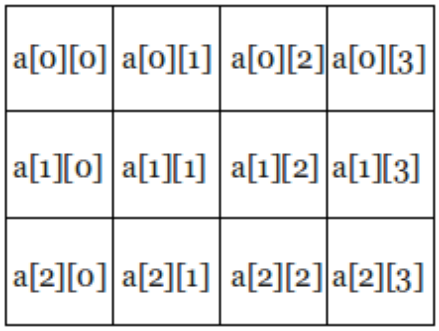
Rysunek 12.3. Dwie metody inicializacli tablicy wielowymiarowej

12.7.3 Dostęp do elementów

Podobnie jak w przypadku tablicy jednowymiarowej, dostęp uzyskuje się przez podanie w nawiasach kwadratowych numerów pozycji dla kolejnych wymiarów – dla przypadku dwuwymiarowego podaje się najpierw numer wiersza, a potem numer kolumny (zob. rys. 12.4).

Oto przykład odwołania się do konkretnego elementu w tablicy dwuwymiarowej:

```
int tab[3][2];
tab[2][0] = 5;  // przypisz wartosc elementowi w 3. wierszu w 1. kolumnie
```



Rysunek 12.4. Indeksowanie elementów w tablicy dwuwymiarowej

Aby wykonać operację na wszystkich elementach tablicy wielowymiarowej najwygodniej jest użyć zagnieżdżonych pętli **for**. Przykładowo, dla tablicy dwuwymiarowej o rozmiarze $N \times M$:

```
int tab[N][M];
for (size_t row = 0; row < N; ++row) {
    for (size_t col = 0; col < M; ++col) {
        tab[row][col] = row * col;
    }
}
```

12.8 Jak „elastycznie” określać rozmiar tablicy: dyrektywa `#define`

Ponieważ obowiązujący standard języka C (tj. C18) *nie zaleca* używania zmiennych albo stałych do określania rozmiaru tablicy podczas deklaracji, chcąc utworzyć tablicę o rozmiarze zadanym przez użytkownika *powinno się* skorzystać z dynamicznej alokacji pamięci (zob. rozdz. 15).

Problem elastycznego określania rozmiarów tablicy wielowymiarowej można jednak częściowo rozwiązać za pomocą stałych symbolicznych – możesz zdefiniować po jednej stałej symbolicznej dla każdego z wymiarów tablicy:

```
#define N_ROWS    2
#define N_COLS    3
```

a następnie stosować je konsekwentnie w programie we wszystkich miejscach, w których potrzebujesz odwołać się do danego wymiaru tablicy, przykładowo:

```
int tab2d[N_ROWS][N_COLS];
```

Ważne

Stosuj stałe symboliczne (albo liczbę elementów obliczoną z użyciem **sizeof**) wszędzie tam, gdzie normalnie powinien występować rozmiar tablicy (tj. dany jej wymiar) – w deklaracjach, w pętlach **for** itd. Dzięki temu z łatwością zmienisz program w sytuacji, gdy dana tablica będzie przechowywała inną liczbę elementów.

12.8.1 Tablice bezwymiarowe

Różnica w tworzeniu tablic „bezwymiarowych” w przypadku tablicy o liczbie wymiarów większej niż 1 jest następująca:

Ważne

Kompilator jest w stanie obliczyć tylko jeden wymiar tablicy – pierwszy od lewej. Pozostałe wymiary muszą zostać określone przez programistę:

```
int tab[][2] = {{1,2}, {3,4}, {5,6}};   // OK
int tab[][] = {{1,2}, {3,4}};           // błąd
```

12.9 Funkcje, tablice i wskaźniki

Listing 12.1 zawiera przykład funkcji zwracającej sumę elementów tablicy. Aby zwrócić uwagę na pewien interesujący fakt związany z argumentami tablicowymi, program wyświetla również rozmiary dwóch występujących w nim tablic.

Listing 12.1. Sumowanie elementów tablicy (wersja 1)

```
/* suma_tab.c - sumuje elementy tablicy */
#define __USE_MINGW_ANSI_STDIO 1

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

#define N 5

int sum_array(int arr[], int n);

int main(void) {
    int balls[N] = {20, 10, 5, 39, 4};
    int n_balls_total = sum_array(balls, N);
    printf("Calkowita liczba kulek: %d\n", n_balls_total);
    printf("Rozmiar tablicy `balls`: %zu bajtow\n", sizeof(balls));
    return EXIT_SUCCESS;
}

int sum_array(int arr[], int n) {
    int sum = 0;
    for (size_t i = 0; i < n; i++) {
        sum += arr[i];
    }
    printf("Rozmiar tablicy `arr`: %zu bajtow\n", sizeof(arr));
    return sum;
}
```

Rezultat uruchomienia powyższego programu:

```
Rozmiar tablicy `arr`: 8 bajtow
Calkowita liczba kulek: 78
Rozmiar tablicy `balls`: 20 bajtow
```

Co wyraża parametr `int arr[]` w funkcji `sum_array()`? Kierując się typami parametrów funkcji, możesz oczekiwać, że `arr` jest nową tablicą, do której kopiowane są wartości z tablicy `balls` (której nazwa została przekazana jako argument wywołania funkcji `sum_array()`). Jednak zmienna `arr` zajmuje mniej miejsca w pamięci niż zmienna `balls` – nie może zatem stanowić jej kopii. W rzeczywistości `arr` jest zwykłym wskaźnikiem, a ponieważ identyfikator tablicy jest jednocześnie adresem jest pierwszego elementu, wywołanie

```
sum_array(balls, N);
```

powoduje przekazanie do funkcji adresu początku tablicy `balls`, który jest przechowywany (w momencie rozpoczęcia wykonywania funkcji `sum_array()`) jako wartość parametru `arr`.

Język C nie pozwala przekazywać tablic jako argumentów funkcji, natomiast pozwala na zaznaczenie faktu przekazywania „wskaźnika do (elementu) tablicy” za pomocą pustych nawiasów kwadratowych:

```
int sum_array(int* arr, int n);
int sum_array(int arr[], int n);  // równoważne powyższej deklaracji, ale
                                  //   czytelniej pokazuje intencję:
                                  //   `arr` to wskaźnik do elementu tablicy
```

Zauważ, że nie wystarczy przekazać do funkcji samego adresu początku tablicy, gdyż nie ma wówczas możliwości określenia rozmiaru takiej tablicy (operator `sizeof` zwróci rozmiar wskaźnika). Są dwa możliwe rozwiązania tego problemu:

- przekazanie dodatkowo liczby elementów tablicy (rozwiązanie zastosowane w przykładzie 12.1)
- przekazanie dodatkowo wskaźnika na koniec tablicy (rozwiązanie zastosowane w przykładzie 12.2)

Listing 12.2. Sumowanie elementów tablicy (wersja 2)

```
/* suma_tab_v2.c - sumuje elementy tablicy */

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

#define N 5

int sum_array(int* arr_begin, int* arr_end);

int main(void) {
    int balls[N] = {20, 10, 5, 39, 4};
    int n_balls_total = sum_array(balls, balls + N);
    printf("Calkowita liczba kulek: %d\n", n_balls_total);

    return EXIT_SUCCESS;
}

int sum_array(int* arr_begin, int* arr_end) {
    int sum = 0;
    for (int* arr_current = arr_begin; arr_current < arr_end; ++arr_current) {
        sum += *arr_current;
    }
    return sum;
}
```

Ponieważ test w pętli `for` zawiera znak ostrej nierówności, ostatnią przetwarzaną wartością jest element znajdujący się tuż przed elementem, na który wskazuje `arr_end`. Choć `arr_end` wskazuje na miejsce w pamięci następujące po ostatnim elemencie tablicy (a więc znajdujące się poza tablicą), język C rezerwuje miejsce dla tablicy w taki sposób, aby wskaźnik do pierwszego bajtu poza jej końcem był wskaźnikiem prawidłowym, zatem wyrażenie `balls + N` jest poprawne (natomiast nie daje on żadnych gwarancji co do wartości przechowywanej pod tym adresem).

12.10 Wskaźniki a tablice wielowymiarowe

Jaki związek istnieje pomiędzy wskaźnikami a tablicami wielowymiarowymi? Aby znaleźć odpowiedź na to pytanie, przyjrzyjmy się kilku przykładom. Dla uproszczenia użyjemy poniższej tablicy:

```
int tab[4][2];  // tablica tablic typu `int`
```

Wówczas `tab`, będąc nazwą tablicy, jest adresem jej pierwszego elementu. W tym przypadku pierwszy element tablicy `tab` jest sam tablicą składającą się z dwóch wartości typu `int`, a zatem `tab` jest adresem tablicy zawierającej dwie liczby całkowite. Poprowadźmy nasze rozumowanie dalej, koncentrując się na własnościach wskaźników:

- Ponieważ `tab` jest adresem pierwszego elementu tablicy, `tab` jest równoważne `&tab[0]`. Z kolei `tab[0]` jest tablicą składającą się z dwóch liczb całkowitych, a więc `tab[0]` jest równoważne `&tab[0][0]`, adresowi jej pierwszego elementu (liczby typu `int`). Mówiąc w skrócie, `tab[0]` jest adresem obiektu o rozmiarze `int`, a `tab` – adresem obiektu o rozmiarze dwóch wartości typu `int`. Ponieważ zarówno liczba całkowita, jak i tablica dwóch liczb całkowitych rozpoczynają się w tym samym miejscu, oba wskaźniki – `tab` i `tab[0]` – mają tę samą wartość liczbową.
- Dodanie 1 do wskaźnika lub adresu zwiększa jego wartość o rozmiar wskazywanego obiektu. Tu `tab` i `tab[0]` różnią się, ponieważ `tab` wskazuje obiekt o rozmiarze jednej wartości typu `int`, a `tab[0]` – obiekt o rozmiarze dwóch takich wartości. Wyrażenia `tab + 1` i `tab[0] + 1` nie mają więc tej samej wartości.
- Dereferencja wskaźnika lub adresu (zastosowanie operatora `*` lub nawiasów kwadratowych z indeksem) daje w wyniku wartość wskazywanego obiektu. Ponieważ `tab[0]` jest adresem elementu `tab[0][0]`, wyrażenie `*(tab[0])` oznacza wartość przechowywaną w tym elemencie. Podobnie

`tab` przechowuje adres podtablicy `tab[0]`, która jest adresem elementu `tab[0][0]`, a zatem `*tab` to tyle, co `&tab[0][0]`. Zastosowanie operatora dereferencji do obu wyrażeń prowadzi do wniosku, że `**tab` jest równoważne `*&tab[0][0]`, czyli tym samym jest jest równoważne `tab[0][0]` – a zatem oznacza wartość typu `int`. Innymi słowy, `tab` jest *adresem adresu* i aby otrzymać „zwykłą" wartość (a nie adres) musimy dokonać *dwukrotnej* dereferencji. Adres adresu lub wskaźnik wskaźnika to przykłady podwójnej pośredniości.

Jak widać, zwiększenie liczby wymiarów tablicy zwiększa równocześnie poziom skomplikowania z punktu widzenia wskaźników. Powyższe własności ilustruje listing 12.3.

Listing 12.3. Tablica dwuwymiarowa a adresy

```
#include <stdio.h>

int main(void) {
    int tab[4][2];
    printf("tab         = %p , tab[0] = %p\n",
           (void*) tab, (void*) tab[0]);
    printf("&tab[0][0] = %p , &tab    = %p\n",
           (void*) &tab[0][0], (void*) *tab);
    printf("\n");
    printf("tab + 1         = %p , tab[0] + 1 = %p\n",
           (void*) (tab + 1), (void*) (tab[0] + 1));
    printf("&tab[0][0] + 1 = %p , *tab + 1    = %p\n",
           (void*) (&tab[0][0] + 1), (void*) (*tab + 1));
    printf("*(tab + 1)      = %p\n", (void*) *(tab + 1));

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
tab         = 000000000022FE30 , tab[0] = 000000000022FE30
&tab[0][0] = 000000000022FE30 , &tab    = 000000000022FE30

tab + 1         = 000000000022FE38 , tab[0] + 1 = 000000000022FE34
&tab[0][0] + 1 = 000000000022FE34 , *tab + 1    = 000000000022FE34
*(tab + 1)      = 000000000022FE38
```

Każdy element tablicy `tab` jest tablicą, a tym samym adresem pierwszego elementu – zachodzą więc następujące zależności:

```
tab[0] == &tab[0][0] == *tab
tab[1] == &tab[1][0] == *(tab + 1)
tab[2] == &tab[2][0] == *(tab + 2)
tab[3] == &tab[3][0] == *(tab + 3)
```

Zastosowanie operatora `*` do wszystkich wyrażeń daje następujący wynik:

```
*tab[0] == tab[0][0] == **tab
*tab[1] == tab[1][0] == *(*(tab + 1))
*tab[2] == tab[2][0] == *(*(tab + 2))
*tab[3] == tab[3][0] == *(*(tab + 3))
```

Mówiąc bardziej ogólnie, notacja tablicowa przekłada się na zapis wskaźnikowy zgodnie z poniższym wzorem:

`tab[m][n] == *(*(tab + m) + n)`

Wartość `m`, jako indeks związany z tablicą `tab`, zostaje dodana do `tab`. Wartość `n`, jako indeks związany z podtablicą `tab[m]`, zostaje dodana do wyrażenia `*(tab + m)`, które jest równoważne `tab[m]`. Tym samym `*(tab + m) + n` jest adresem elementu `tab[m][n]`, a użycie operatora `*` daje w wyniku wartość zapisaną pod tym adresem.

Załóżmy, że chcesz zadeklarować zmienną wskaźnikową `ptr`, która byłaby zgodna z tablicą `tab`. Czy wystarczy typ „wskaźnik do `int`"? Nie – ten typ jest bowiem zgodny z podtablicą `tab[0]`, która wskazuje

pojedynczą wartość `int`, a `ptr` ma przecież wskazywać na całą tablicę takich wartości. Oto, co możesz zrobić:

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

Powyższa instrukcja stwierdza, że `ptr` jest wskaźnikiem do tablicy składającej się z dwóch wartości typu `int`. Po co nawiasy? Są one niezbędne, ponieważ operator `[]` ma wyższy priorytet niż operator dereferencji.

12.11 Funkcje a tablice wielowymiarowe

Wiedza o wskaźnikach do wskaźników jest niezbędna m.in. do pisania funkcji przetwarzających tablice dwuwymiarowe – przede wszystkim musisz rozumieć wskaźniki na tyle dobrze, aby móc sformułować prawidłowe deklaracje dla argumentów funkcji, natomiast w samej treści funkcji zazwyczaj wystarcza znajomość notacji tablicowej.

Załóżmy więc, że chcesz przetwarzać tablice dwuwymiarowe. Masz kilka możliwości. Możesz użyć funkcji dostosowanej do tablic jednowymiarowych, przekazując jej po jednej podtablicy. Możesz również przekazać takiej funkcji całą tablicę, traktując ją jako tablicę jednowymiarową. Możesz wreszcie napisać funkcję przeznaczoną specjalnie do obsługi tablic dwuwymiarowych. Aby zilustrować te trzy podejścia, stwórzmy tablicę dwuwymiarową o niewielkim rozmiarze i spróbujmy podwoić każdy z jej elementów.

Standard języka C stwierdza, że „elementy tablicy (jednowymiarowej) są przechowywane w sposób ciągły", zatem stosując indukcję możemy zauważyć, że skoro tablica dwuwymiarowa to tablica tablic – jej elementy (tablice jednowymiarowe) również są przechowywane w sposób ciągły itd. Zatem tablicę o rozmiarze (liczbie elementów) $N_1 \times N_2 \times \ldots N_n$ możesz potraktować jako jednowymiarową tablicę o liczbie elementów $N_1 \cdot N_2 \cdot \ldots \cdot N_n$ i tym samym możesz wywołać funkcję oczekującą tablicy jednowymiarowej przekazując jako argument wskaźnik do tablicy wielowymiarowej – w wywołaniu musisz jedynie podać odpowiedni rozmiar tablicy oraz dokonać stosownego rzutowania wskaźnika na początek tablicy. Przykładowo:

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

#define N 2
#define M 3

int sum_array(int arr[], int n);

int main(void) {
    int balls[N][M] = {{1, 2, 3},
                       {4, 5, 6}};
    int n_balls_total = sum_array((int*) balls, N * M);
    printf("Calkowita liczba kulek: %d\n", n_balls_total);

    return EXIT_SUCCESS;
}

int sum_array(int arr[], int n) {
    int sum = 0;
    for (size_t i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

W powyższym przykładzie `balls` jest typu `int**`, który nie jest zgodnym z typem parametru `arr` (czyli `int*`), stąd niezbędne jest zastosowanie rzutowania.

W przypadku, gdy dane w tablicy wielowymiarowej powinny być rozpatrywane z uwzględnieniem wymiaru (np. gdy wiersze oznaczają lata, a kolumny – miesiące), funkcja powinna dysponować informacją o liczbie rzędów i kolumn. Można to osiągnąć przez zadeklarowanie odpowiedniego rodzaju parametru tak, aby możliwe było poprawne przekazanie tablicy wielowymiarowej. W powyższym przykładzie `balls` jest tablicą dwóch tablic, z których każda zawiera trzy wartości typu `int`. Jak sugerowało to wcześniejsze omó-

wienie, oznacza to, że `balls` jest wskaźnikiem do tablicy złożonej z trzech liczb całkowitych, a definicja odpowiadającego mu parametru funkcji powinna wyglądać następująco:

```
int (* ptr)[3];
```

lub równoważnie

```
int ptr[][3];
```

Pierwszy nawias kwadratowy jest pusty co sygnalizuje, że `ptr` jest wskaźnikiem. Tak zdefiniowana zmienna umożliwia dostęp do elementów w identyczny sposób, co w przypadku oryginalnej tablicy. Jest to możliwe dzięki temu, że `balls` i `ptr` należą do tego samego typu – oba są wskaźnikami do tablic złożonych z trzech wartości typu **`int`**.
Miej świadomość, że poniższa definicja nie zadziała:

```
int tab[][];  // BŁĄD
```

Kompilator przekształca zapis tablicowy na zapis wskaźnikowy – oznacza to na przykład, że `tab[1]` zostaje zamienione na `tab + 1`, natomiast aby obliczyć to wyrażenie, kompilator musi znać rozmiar obiektu wskazywanego przez `tab`. W przypadku pustych nawiasów kompilator nie wiedziałby, o ile zwiększyć adres. Ogólnie rzecz biorąc, deklarując wskaźnik odpowiadający tablicy n-wymiarowej, musisz umieścić wartości we wszystkich nawiasach oprócz pierwszego (pierwszy rozmiar zostanie zignorowany przez kompilator).

Rozdział 13

Klasy zmiennych

W tym rozdziale dowiesz się, w jaki sposób język C pozwala określić zasięg zmiennej (czyli jej dostępność w różnych modułach programu) oraz jej okres trwania (czyli okres pozostawania w pamięci).

Każda zmienna należy nie tylko do określonego typu (np. `int` albo `double`), ale także do określonej klasy określającej:

- *widoczność* zmiennej – czyli to, który fragment kodu może korzystać z określonej zmiennej (widoczność zmiennej określona jest przez jej zasięg),
- *łączność* zmiennej – czyli to w ilu różnych miejscach może zostać zadeklarowana ta sama zmienna, oraz
- *okres trwania* zmiennej – czyli to jak długo zmienna jest obecna w pamięci.

Dodatkowo zmienna może posiadać kwalifikatory określające jej *stałość* i *ulotność*.

13.1 Odnajdywanie nazw i przestrzenie nazw

Gdy kompilator natrafia w kodzie programu na identyfikator, rozpoczyna procedurę poszukiwania deklaracji tego identyfikatora, która obowiązuje w danym zasięgu[1]. Język C dopuszcza występowanie kilku identycznych identyfikatorów w tym samym zasięgu, o ile należą one do różnych **przestrzeni nazw** (ang. namespaces); oto przestrzenie nazw występujące w języku C:

- nazwy etykiet – wszystkie identyfikatory oznaczające etykiety
- nazwy tagów (ang. tags) – wszystkie identyfikatory oznaczające nazwy (szablonów) struktur, unii i typów wyliczeniowych; wszystkie te trzy rodzaje tagów współdzielą tą samą przestrzeń nazw
- nazwy składowych – wszystkie identyfikatory oznaczające składowe danej struktury albo unii; każda struktura i każda unia wprowadza własną przestrzeń nazw
- inne, „zwykłe” identyfikatory (ang. ordinary identifiers) – nazwy funkcji i obiektów, aliasy, stałe wyliczeniowe

13.2 Zasięg

Każdy identyfikator pojawiający się w programie napisanym w języku C ma swoją określoną **widoczność** (ang. visibility), czyli może być użyty jedynie w określonym fragmencie programu nazywanym jego **zasięgiem** (ang. scope), przy czym fragment ten nie musi być ciągły. Przykładowo, w poniższym programie.

```c
#include <stdlib.h>

int main(void) {
    for (int i = 0; i < 3; ++i) {
        continue;
    }
    i = 2;  // BŁĄD: zmienna `i` istnieje tylko w obrębie pętli for

    return EXIT_SUCCESS;
}
```

[1] Procedura ta nazywa się po angielsku *lookup*

próba odwołania się do zmiennej `i` w linii 8 – a więc poza instrukcją **`for`**, wewnątrz której zmienna `i` została zadeklarowana – spowoduje błąd kompilacji.

Język C wyróżnia cztery rodzaje zasięgu: blokowy, plikowy, funkcyjny i prototypowy.

W obrębie danego zasięgu dany identyfikator może oznaczać więcej niż dwa byty (funkcje, zmienne, typy itd.) o ile znajdują się one w różnych przestrzeniach nazw, przykładowo:

```
#include <stdlib.h>

int main(void) {
    int finish = 1;

    while (1) {
        while (1) {
            goto finish;
        }
    }
    finish:  // etykieta `finish` należy do innej przestrzeni nazw niż
             //   nazwa zmiennej `finish`

    return EXIT_SUCCESS;
}
```

13.2.1 Zasięg blokowy

Zasięg blokowy (ang. block scope) odnosi się do każdego identyfikatora zadeklarowanego w obrębie

- instrukcji złożonej (w tym ciała funkcji),
- wyrażenia, deklaracji lub instrukcji pojawiającej się w instrukcjach **`if`**, **`switch`**, **`for`**, **`while`** lub **`do-while`**[2], lub
- listy parametrów funkcji w ramach jej *definicji*

i obejmuje fragment programu od miejsca deklaracji takiego identyfikatora do końca bloku lub instrukcji, w ramach której identyfikator został zadeklarowany. Zmienne o zasięgu blokowym nazywane są również **zmiennymi lokalnymi** (ang. local variables). Sytuację tę ilustruje przykład 13.1.

Listing 13.1. Zasięg blokowy

```
void f(int n)  // początek zasięgu parametru `n` funkcji `f`
{  // początek bloku - ciała funkcji
    ++n;   // `n` jest w zasięgu i odnosi się do parametru funkcji `f`
// int n = 2; // BŁĄD: nie można ponownie zadeklarować identyfikator
              // w tym samym zasięgu
    for (int n = 0; n < 10; ++n) // początek zasięgu lokalnej zmiennej `n`
                                 //   (widocznej tylko w pętli) - przesłania
                                 //   ona parametr funkcji
    {  // początek bloku - ciała pętli
        printf("%d\n", n); // wypisze: 0 1 2 3 4 5 6 7 8 9
    }  // początek bloku - ciała pętli = koniec zasięgu lokalnej zmiennej `n`
       //   od tego miejsca parametr `n` znów w zasięgu
    printf("%d\n", n);  // wypisze wartość parametru `n`
}  // koniec zasięgu parametru `n`
//int a = n; // BŁĄD: nazwa `n` nie znajduje się w zasięgu
```

(W powyższym przykładzie występuje zjawisko „przesłaniania", omówione dokładniej w rozdziale *13.2.5 Zasięg zagnieżdżony*.)

Zmienne o zasięgu blokowym mają domyślnie brak łączności i automatyczny okres trwania. Choć trwanie zmiennej lokalnej[3] zaczyna się w momencie wejścia do bloku, dopóki program nie dotrze do deklaracji zmiennej zmienna nie znajduje się w zasięgu (czyli nie można korzystać z takiej zmiennej).

[2] Począwszy od standardu C99 – wcześniej instrukcje wyboru i iteracji nie wprowadzały własnego zakresu.

[3] nie będącej tablicą o zmiennej długości

13.2.2 Zasięg plikowy

Zasięg plikowy (ang. file scope) odnosi się do każdego identyfikatora zadeklarowanego poza jakimkolwiek blokiem kodu lub listą parametrów – zasięg ten obejmuje fragment programu od miejsca deklaracji identyfikatora do końca danej jednostki translacji. Zmienne o zasięgu plikowym są również nazywane **zmiennymi globalnymi** (ang. global variables).

```
int i;  // początek zasięgu `i`
int main(void) {
    i = 1;
    /* ... */
}
```

Wszystkie identyfikatory o zasięgu plikowym mają domyślnie łączność zewnętrzną i statyczny okres trwania.

13.2.3 Zasięg funkcyjny

Zasięg funkcyjny (ang. function scope) odnosi się tylko i wyłącznie do etykiet skoku zadeklarowanych w obrębie funkcji – etykieta ta jest widoczna w całym zakresie tej funkcji, we wszystkich blokach zagnieżdżonych, zarówno przed, jak i po deklaracji tej etykiety:

```
void f() {
    {
        goto label; // etykieta w zasięgu, choć zadeklarowana później
        label:;
    }
    goto label;
}

void g() {
    goto label; // BŁĄD: etykieta nie jest w zasięgu funkcji `g()`
}
```

13.2.4 Zasięg prototypowy

Zasięg prototypowy (ang. function prototype scope) odnosi się do każdego identyfikatora wprowadzonego w ramach listy parametrów deklaracji funkcji (która nie jest jednocześnie jej definicją) – zasięg ten obejmuje fragment programu od momentu wprowadzenia identyfikatora do końca deklaracji funkcji.

W praktyce oznacza to tyle, że kompilator interesują wyłącznie typy zmiennych zadeklarowanych w prototypie, a nie ich ewentualnie użyte nazwy.

13.2.5 Zasięg zagnieżdżony

W przypadku, gdy dwa różne byty nazwane z użyciem tego samego identyfikatora znajdują się w tym samym zakresie a jednocześnie należą do tej samej przestrzeni nazw, mamy do czynienia z tzw. **zasięgami zagnieżdżonymi** (ang. nested scopes) – to jedyna dopuszczalna forma nakładania się zasięgów. Deklaracja, która pojawia się w zasięgu wewnętrznym **przesłania** (ang. hide) deklarację występującą w zasięgu zewnętrznym. Taka sytuacja występowała w listingu 13.1 – poniżej zamieszczono ponownie jego kluczowy fragment:

```
label={lst:storage_classes__block_scope}]
void f(int n) { // początek zasięgu parametru `n` funkcji `f`
    n = 0;
    for (int n = 0; n < 10; ++n) {  // początek zasięgu lokalnej zmiennej `n`
                                    //   (widocznej tylko w pętli) -
                                    //   przesłania ona parametr funkcji
        printf("%d\n", n); // wypisze: 0 1 2 3 4 5 6 7 8 9
    }  // początek bloku - ciała pętli = koniec zasięgu lokalnej zmiennej `n`
       //   od tego miejsca parametr `n` znów w zasięgu
```

```
    printf("%d\n", n);  // wypisze wartość parametru `n`, czyli 0
}  // koniec zasięgu parametru `n`
```

13.3 Łączność

Łączność (ang. linkage) to cecha identyfikatora (zmiennej lub funkcji) określająca to, czy i w jakich innych zasięgach można się do niego odnosić. Jeśli zmienna albo funkcja o tym samym identyfikatorze została zadeklarowana w kilku zasięgach, lecz (zgodnie z regułami języka C) nie można się do niej odwoływać z każdego z tych zasięgów, wówczas wygenerowanych zostanie kilka instancji takiej zmiennej albo funkcji.

Język C wyróżnia następujące typy łączności:

- **brak łączności** (ang. no linkage) – zmienna pozbawiona łączności może być użyta tylko w tym bloku, w którym została zadeklarowana (przykładami takich zmiennych są parametry funkcji)
- **łączność wewnętrzna** (ang. internal linkage) – zmienna o łączności wewnętrznej może być użyta wyłącznie w zasięgach w obrębie tej jednostki translacji, w której została zadeklarowana
- **łączność zewnętrzna** (ang. external linkage) – zmienna o łączności zewnętrznej może zostać użyta w każdej z jednostek translacji całego programu

Zmienne o łączności wewnętrznej lub zewnętrznej mogą występować w więcej niż jednej deklaracji, natomiast zmienne pozbawione łączności mogą być deklarowane tylko raz.

Pojęcia łączności i zasięgu są ze sobą związane:

- zmienne o zasięgu blokowym lub prototypowym nie posiadają łączności
- zmienne o zasięgu plikowym może cechować łączność zewnętrzna lub wewnętrzna

13.4 Okres trwania

Każdy obiekt w języku C istnieje, posiada stały adres i zachowuje ostatnią nadaną mu wartość (o ile nie jest to wartość nieokreślona) w obrębie pewnego fragmentu wykonania programu nazywanego **cyklem życia** obiektu (ang. object's lifetime). Każdy obiekt w języku C charakteryzowany jest przez **okres trwania** (ang. storage duration), ograniczający cykl życia takiego obiektu. Język C wyróżnia cztery typy okresu trwania:

- **automatyczny** (ang. automatic) – Obiekt o automatycznym okresie trwania jest alokowany w momencie wejścia do bloku, w którym obiekt ten został zdefiniowany, a usuwany z pamięci w momencie wyjścia z tego bloku w dowolny sposób (osiągnięcie końca bloku, instrukcja `return`, instrukcja `goto`). Jeśli dany blok jest odwiedzany rekursywnie, na każdym poziomie rekursji dokonywana jest nowa alokacja.
- **statyczny** (ang. static) – Obiekt o statycznym okresie trwania istnieje przez cały okres działania programu; obiekt ten jest inicjalizowany tylko raz, przed rozpoczęciem wykonywania funkcji `main()`.
- **alokowany** (ang. allocated) – Obiekt jest alokowany i usuwany na żądanie, z użyciem funkcji do zarządzania pamięcią[4].
- **wątkowy** (ang. thread) – Występuje w przypadku programów wielowątkowych, co wykracza poza ramy niniejszego skryptu.

13.5 Specyfikatory klas przechowywania

Język C udostępnia pięć specyfikatorów określających tzw. klasę przechowywania" (ang. storage-class specifiers) obiektów i funkcji – czyli określających jednocześnie zarówno ich okres trwania, jak i łączność:

- `auto` – automatyczny okres przechowywania, brak łączności
- `register` – automatyczny okres przechowywania, brak łączności; nie można uzyskać adresu takiej zmiennej
- `static` – statyczny okres przechowywania, łączność wewnętrzna (o ile nie w zasięgu blokowym)
- `extern` – statyczny okres przechowywania, łączność zewnętrzna (o ile nie został już zadeklarowany z łącznością wewnętrzną)
- `_Thread_local` – występuje w przypadku programów wielowątkowych, co wykracza poza ramy niniejszego skryptu

Specyfikatory te występują w deklaracjach, przy czym w jednej deklaracji może występować co najwyżej jeden specyfikator.

[4] zob. rozdz. 15 Zarządzanie pamięcią

Jeśli deklaracja nie zawiera jawnie podanego specyfikatora klasy przechowywania, kompilator przyjmuje domyślny specyfikator zgodnie z poniższymi zasadami:

- Wszystkie funkcje otrzymują specyfikator **extern**.
- Wszystkie obiekty o zasięgu plikowym otrzymują specyfikator **extern**.
- Wszystkie obiekty o zasięgu blokowym otrzymują specyfikator **auto**.

13.5.1 Specyfikator auto

Specyfikator **auto** jest dozwolony jedynie dla obiektów o zasięgu blokowym (z wyjątkiem list parametrów funkcji); wskazuje on automatyczny okres przechowywania i brak łączności. Zmienne zadeklarowane w obrębie funkcji należą do klasy automatycznej standardowo, zatem specyfikator **auto** można pominąć (np. zamiast pisać **auto int** x; wystarczy **int** x;).

Domyślna początkowa wartość zmiennej automatycznej jest niezdefiniowana – wartością tą jest cokolwiek, co znajdowało się wcześniej w zajmowanych przez nią komórkach pamięci. Dlatego do dobrych praktyk należy każdorazowe jawne określanie początkowej wartości zmiennej automatycznej – czyli jej inicjalizacja:

```
int x;  // `x` ma wartość niezdefiniowaną ("śmieci" z pamięci)
int y = 1;  // `y` ma wartość 1
```

13.5.2 Specyfikator register

Miejscem przechowywania zmiennych jest zwykle pamięć operacyjna. Zmienne zadeklarowane z użyciem specyfikatora **register** (tzw. zmienne rejestrowe) są wyjątkiem – mogą być one przechowywane w rejestrach procesora, co pozwala odczytywać je i przetwarzać ze znacznie większą prędkością niż zmienne zwykłego typu. Wszystkie inne własności zmiennych rejestrowych są takie same, jak w przypadku zmiennych automatycznych (tj. automatyczny okres przechowywania oraz brak łączności). Specyfikator **register** jest dopuszczalny jedynie dla obiektów o zasięgu blokowym (w tym list parametrów funkcji).

Użycie specyfikatora **register** nie tyle *gwarantuje*, że zmienna zostanie umieszczona w rejestrze procesora, co *dopuszcza taką możliwość*. Ostateczną decyzję podejmuje kompilator na etapie optymalizacji kodu, uwzględniając m.in. liczbę dostępnych rejestrów. Niezależnie od tego, czy ostatecznie zmienna rejestrowa zostanie umieszczona w rejestrze, czy w pamięci operacyjnej, nie może być ona użyta jako argument operatora adresu.

13.5.3 Specyfikator static

Specyfikator **static** określa statyczny okres przechowywania oraz łączność wewnętrzną (o ile nie zostanie użyty do zadeklarowania obiektu o zasięgu blokowym); może być użyty w odniesieniu do funkcji o zasięgu plikowym oraz zmiennych zarówno o zasięgu plikowym, jak i blokowym (ale nie do list parametrów funkcji).

> **Ważne**
>
> Przymiotnik „statyczny” (ang. static) może oznaczać – w zależności od kontekstu – albo statyczny okres trwania, albo specyfikator **static**. W związku z tym do dobrej praktyki należy każdorazowe precyzowanie, o który rodzaj „statyczności” chodzi.

Zmienne statyczne otrzymują domyślną wartość początkową 0 (podobnie tablice statyczne są domyślnie wypełniane zerami).

W przypadku zmiennej o zasięgu blokowym użycie specyfikatora **static** sprawia, że jej wartość będzie przechowywana pomiędzy kolejnymi wywołaniami funkcji:

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

void f(void);

int main(void) {
    for (int i = 1; i <= 3; i++) {
        printf("Iteration #%d\n", i);
```

```
        f();
    }

    return EXIT_SUCCESS;
}

void f(void) {
    int auto_var = 1;
    static int static_var = 1;
    printf("auto = %d, static = %d\n", auto_var++, static_var++);
}
```

Wynik wykonania powyższego programu to:

```
Iteration #1
auto = 1, static = 1
Iteration #2
auto = 1, static = 2
Iteration #3
auto = 1, static = 3
```

Zwróć uwagę, że zmienna `static_var` jest inicjalizowana tylko raz, przed rozpoczęciem wykonywania funkcji `main()`.

13.5.4 Specyfikator `extern`

Specyfikator **`extern`** określa statyczny okres przechowywania oraz łączność zewnętrzną; może być użyty w odniesieniu do deklaracji funkcji i obiektów zarówno o zasięgu plikowym, jak i blokowym (ale nie do list parametrów funkcji).

Specyfikator **`extern`** może posłużyć do powtórnej *deklaracji* identyfikatora:

- jeśli identyfikator ten został wcześniej zadeklarowany z łącznością wewnętrzną, wówczas łączność pozostaje bez zmian;
- w przeciwnym razie (jeśli identyfikator ten został wcześniej zadeklarowany z łącznością zewnętrzną, bez łączności lub poza zasięgiem) łączność jest zewnętrzna.

Zmienne zewnętrzne otrzymują domyślną wartość początkową 0 (podobnie tablice zewnętrzne są domyślnie wypełniane zerami).

Aby poprawić czytelności kodu zmienna zewnętrzna może zostać ponownie zadeklarowana w obrębie wykorzystującej ją funkcji (jeśli zmienna jest zdefiniowana w innym pliku, zadeklarowanie jej przy pomocy specyfikatora **`extern`** jest obowiązkowe):

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

int global_var__same_file;
double global_arr__same_file[10];

extern char global_var__other_file;  // deklaracja obowiązkowa -
                                     //   zmienna zadeklarowana w globals.h
                                     //   zdefiniowana w innym pliku

int main(void) {
    extern int global_var__same_file;  // opcjonalna deklaracja
    extern double global_arr__same_file[];  // opcjonalna deklaracja

    global_var__same_file = 1;
    global_arr__same_file[0] = 1;

    return EXIT_SUCCESS;
}
```

Zauważ, że opcjonalna deklaracja zmiennej `global_arr__same_file` nie musi zawierać rozmiaru tablicy, ponieważ informacja ta znajduje się już w deklaracji pierwotnej.

Pamiętaj, że użycie specyfikatora **`extern`** zawsze oznacza *deklarację* a nie *definicję* – deklaracja nie powoduje przydzielenia miejsca w pamięci na przechowywanie obiektu, a jedynie informuje kompilator o występowaniu danego identyfikatora:

```
int x;  // definicja

int main (void) {
  extern int x;  // deklaracja zmiennej `x` zdefiniowanej gdzie indziej
  /* ... */
}
```

W powyższym kodzie zmienna `x` jest deklarowana dwukrotnie: pierwsza deklaracja jest równocześnie definicją; druga wskazuje jedynie, że kompilator powinien skorzystać z utworzonej wcześniej zmiennej. Pierwszą deklarację nazywamy **deklaracją definiującą** (ang. defining declaration), a drugą – **deklaracją nawiązującą** (ang. referencing declaration). Tak więc wielokrotna *deklaracja* symbolu nie jest błędem – błędem jest dopiero jego wielokrotna *definicja*. To szczególnie istotne podczas pracy ze zmiennymi zewnętrznymi – zmienna zewnętrzna może zostać zainicjalizowana tylko jeden raz, a jej inicjalizacja musi zostać dokonana razem z definicją. W związku z tym użycie specyfikatora **`extern`** w instrukcji w rodzaju **`extern int`** `q = 1;` jest błędne, ponieważ obecność specyfikatora **`extern`** sygnalizuje deklarację nawiązującą, a nie deklarację definiującą.

Do dobrych praktyk należy umieszczanie specyfikatora **`extern`** przy prototypie funkcji w sytuacji, gdy jej definicja znajduje się w innym pliku – poprawia to czytelność kodu (nie jest jednak konieczne, gdyż każda funkcja domyślnie należy do klasy **`extern`**).

Pamiętaj, że niezdefiniowanie zadeklarowanego symbolu spowoduje, że na etapie *konsolidacji* pojawi się błąd „`undefined reference to '...'`”, przykładowo:

```
#include <stdlib.h>

int main(void) {
    extern int x;  // deklaracja
    x = 1;  // odwołanie się do niezdefiniowanej zmiennej

    return EXIT_SUCCESS;
}
```

13.5.5 Który specyfikator wybrać?

Jedną ze złotych reguł bezpiecznego programowania jest reguła „minimalnej wiedzy”, która mówi: *zawsze stosuj rozwiązania dające minimum swobody – akurat tyle, by rozwiązać problem*. W szczególności, wewnętrzne mechanizmy funkcji powinny być tak bardzo ukryte, jak to tylko możliwe; „na zewnątrz” widoczne powinny być tylko te zmienne, które koniecznie muszą być udostępnione. W związku z tym:

- Domyślnie stosuj zmienne automatyczne (przed użyciem klasy nieautomatycznej zastanów się, czy jest to niezbędne).
- Unikaj wprowadzania zmiennych zewnętrznych – ponieważ są one widoczne w wielu funkcjach, łatwo o ich przypadkową modyfikację (możesz natomiast stosować stałe zewnętrzne).

13.6 Łączność a biblioteki

W przypadku programu złożonego z wielu plików pojawia się pytanie: gdzie należy zdefiniować (i równocześnie zainicjalizować) funkcje i zmienne zadeklarowane w pliku nagłówkowym?

Przykładowo, zdefiniowanie funkcji w pliku nagłówkowym spowoduje problemy w przypadku dołączania takiego pliku nagłówkowego w kilku plikach źródłowych:

```
/* header.h */
#ifndef HEADER_H
#define HEADER_H
```

```
void foo(void) {}

#endif //HEADER_H
```

```
/* source1.c */
#include "header.h"

// ...
```

```
/* source2.c */
#include "header.h"

// ...
```

gdyż preprocesor skopiuje definicję z pliku nagłówkowego do obu plików źródłowych, przez co na etapie konsolidacji programu wystąpi błąd wielokrotnej definicji tego samego obiektu: `multiple definition of 'foo'`.

Umieszczenie w definicji specyfikatora **`static`** rozwiąże problem wielokrotnych definicji, gdyż oznacza on obiekt o łączności wewnętrznej – zatem każda z jednostek translacji otrzyma własną definicję funkcji `foo()` widoczną tylko w tej jednostce translacji (odnosi się to do każdego obiektu o łączności wewnętrznej). Pamiętaj jednak, że w tym przypadku program potrzebuje więcej pamięci operacyjnej na przechowanie duplikatów kodu (implementacji) takiej funkcji. Analogicznie użycie specyfikatora **`static`** w definicji zmiennej spowoduje utworzenie *niezależnej kopii* w każdej jednostce translacji zawierającej dołączony ten plik nagłówkowy.

Inne rozwiązanie polega na umieszczeniu w pliku nagłówkowym jedynie *deklaracji* identyfikatora (z użyciem specyfikatora **`extern`**), a jego definicji – w osobnym pliku źródłowym. Wówczas wszystkie jednostki translacji zawierające ten plik nagłówkowy będą współdzielić identyfikator odnoszący się do tego samego miejsca w pamięci komputera.

Oto przykład biblioteki, która korzysta z drugiego rozwiązania:

```
// flib.h - interfejs biblioteki
#ifndef FLIB_H
#define FLIB_H

extern void f(void);  // deklaracja funkcji z łącznością zewnętrzną

#endif // FLIB_H
```

```
// flib.c - implementacja biblioteki
#include "flib.h"

void f(void) {}  // definicja funkcji o łączności zewnętrznej
```

```
// main.c - główny kod programu
#include <stdlib.h>
#include "flib.h"

int main(void) {
    f();                  // wywołanie funkcji `f` z pliku flib.c

    return EXIT_SUCCESS;
}
```

13.7 Kwalifikatory typów

Oprócz zasięgu, łączności i okresu trwania zmienną opisują jeszcze dwie cechy: **stałość** (ang. constancy) i **ulotność** (ang. volatility). Własności te deklaruje się za pomocą słów kluczowych **`const`** i **`volatile`**; słowa te tworzą tzw. **typy kwalifikowane** (ang. qualified types)[5].

[5] Analogicznie, **typ niekwalifikowany** (ang. unqualified type) to typ pozbawiony kwalifikatorów.

13.7.1 Kwalifikator `const`

Kwalifikator **`const`** służy do deklarowania obiektu, którego wartość nie ulega zmianie po inicjalizacji (czyli do deklarowania stałych) w czasie jego cyklu życia:

```
const double PI = 3.14;  // definicja stałej `PI`
PI = 3;  // BŁĄD (kompilacji)
```

Stałe lokalne

W przypadku stałych lokalnych zwykle lepiej zadeklarować je jako stałe statyczne (chyba, że wartość stałej zmienia się przy każdym wywołaniu funkcji)[6], gdyż zmienne statyczne są inicjalizowane tylko raz.

W przypadku inicjalizacji stałej wartością trywialnego wyrażenia (np. `3.14`) nie ma to dużego znaczenia. Jednak w przypadku inicjalizacji stałej wartością pewnego nietrywialnego wyrażenia (np. `fibonacci(100)` – czyli obliczanie 100. wyrazu ciągu Fibonacciego) możesz zauważyć istotną różnicę w wydajności programu:

```
// brak istotnych różnic w wydajności
void foo1(void) {
    static const double PI = 3.14;
}
void foo2(void) {
    const double PI = 3.14;
}

// bar2() wolniejsze niż bar1()
void bar1(void) {
    static const unsigned long long fib100 = fibonacci(100);
}
void bar2(void) {
  // Wartość `fib100` obliczana przy każdym wywołaniu funkcji.
    const unsigned long long fib100 = fibonacci(100);
}
```

Stałe globalne

W przeciwieństwie do „zwykłych” zmiennych zewnętrznych korzystanie ze „stałych” zmiennych zewnętrznych jest bezpieczne – ponieważ ich wartość nie ulega zmianie po inicjalizacji, mimo współdzielenia takiej zmiennej przez różne fragmenty programu nie ma ryzyka przypadkowej modyfikacji wartości takich obiektów.

Korzystając ze stałych globalnych w połączeniu z plikami nagłówkowymi zwróć uwagę na potencjalne problemy omówione w rozdziale 13.6 Łączność a biblioteki:

- Umieszczenie w definicji specyfikatora **`static`** rozwiąże problem wielokrotnych definicji, gdyż oznacza on obiekt o łączności wewnętrznej – zatem każda z jednostek translacji otrzyma własną definicję stałej `PI` widoczną tylko w tej jednostce translacji. Pamiętaj, że w tej sytuacji wartość stałej będzie obliczana osobno w każdej jednostce translacji oraz że program potrzebuje więcej pamięci operacyjnej na przechowanie duplikatów stałej.
- Inne rozwiązanie polega na umieszczeniu w pliku nagłówkowym jedynie *deklaracji* stałej (z użyciem specyfikatora **`extern`**), a jej definicji – w osobnym pliku źródłowym. Wówczas wszystkie jednostki translacji dołączające z tego pliku nagłówkowego będą współdzielić jedną i tą samą stałą w pamięci komputera (której wartość zostanie obliczona tylko raz).

13.7.2 Kwalifikator `volatile`

Kwalifikator **`volatile`** informuje kompilator, że zmienna może zostać zmodyfikowana przez czynniki inne niż sam program. Zwykle jest on wykorzystywany w odniesieniu do adresów sprzętowych lub da-

[6] W praktyce nawet jeśli nie zadeklarujesz stałej jako statycznej, ale jej wartość nie zmienia się pomiędzy wywołaniami funkcji, kompilator zwykle dokona stosownej optymalizacji za Ciebie.

nych użytkowanych wspólnie z innymi, działającymi równolegle programami. Na przykład, wskaźnik może przechowywać adres aktualnego czasu systemowego. Wartość pod tym adresem ulega zmianie w czasie, niezależnie od tego, co robi Twój program.

Informacja o ulotności jest istotna dla każdego kompilatora dokonującego optymalizacji kodu. Przykładowo kompilując poniższy fragment

```
wart1 = x;
/* kod nie zmieniający wartości `x` */
wart2 = x;
```

inteligentny (optymalizujący) kompilator mógłby zorientować się, że zmienna `x` jest używana dwukrotnie bez zmiany wartości, i tymczasowo przechować ją w rejestrze procesora. Dzięki temu druga instrukcja wykorzystująca `x` mogłaby zostać wykonana szybciej – program nie musiałby bowiem odczytywać wartości zmiennej `x` z pamięci. Procedura taka nosi nazwę **buforowania** (ang. caching). Buforowanie jest dobrą metodą optymalizacji tylko wówczas, gdy wiadomo, że zmienna `x` nie ulegnie zmianie wskutek działania jakiegoś czynnika zewnętrznego. Gdyby nie kwalifikator **`volatile`**, kompilator nie wiedziałby, czy może się to zdarzyć – w imię bezpieczeństwa musiałby więc zupełnie zrezygnować z buforowania.

Kwalifikator **`volatile`** może występować w połączeniu z kwalifikatorem **`const`** (przy czym ich kolejność nie ma znaczenia). Przykładowo, zegar sprzętowy komputera nie powinien być modyfikowany przez program (czyli jest stały), ale równocześnie jest modyfikowany przez czynniki inne niż program (czyli jest ulotny) – zatem deklaracja identyfikatora pozwalającego na odwoływanie się do zegara systemowe może wyglądać następująco:

```
const volatile int* clock = ...;
```

Rozdział 14

Organizacja programu

W tym rozdziale zapoznasz się ze sposobem podziału programu na pliki nagłówkowe i źródłowe tak, aby poprawić jego czytelność i umożliwić wielokrotne wykorzystywanie kodu.

14.1 Przykład tworzenia biblioteki programistycznej

Załóżmy, że zajmujesz się obliczeniami z zakresu dynamiki Newtona i często musisz korzystać z takich wartości jak stała grawitacji czy masa Ziemi oraz np. z funkcji obliczającej siłę przyciągania się dwóch ciał. Na krótką metę najprostszym rozwiązaniem byłoby skopiowanie definicji tych stałych i funkcji do każdego z plików źródłowych, które z nich korzystają – i tak dla każdego programu z osobna. Jest to jednak rozwiązanie czasochłonne oraz bardzo podatne na błędy – jeśli kiedyś zmienisz definicję choć jednej ze stałych czy funkcji, musisz pamiętać o wprowadzeniu stosownych zmian we *wszystkich* wszystkich plikach źródłowych korzystających z tej definicji. Znacznie lepszym wyjściem jest umieszczenie takiej „bibliotecznej" funkcjonalności do osobnej pary plików – do **pliku nagłówkowego** (ang. header file) zawierającego *deklaracje* stałych i funkcji oraz do **pliku źródłowego** (ang. source file) zawierającego *definicje* tych symboli – a następnie dołączaniu takiego pliku nagłówkowego (za pomocą dyrektywy `#include`) w każdym z plików źródłowych chcącym korzystać z funkcjonalności tej biblioteki (na etapie konsolidacji programu konsolidator odnajdzie niezbędne definicje z „bibliotecznego" pliku źródłowego).

Oto przykład wspomnianego wydzielenia biblioteki, składającej się z pary plików `physics.h` i `physics.c`, oraz wykorzystania tej biblioteki w programie. Aby nie definiować stosowanych stałych i funkcji w każdym pliku źródłowym możesz utworzyć plik nagłówkowy np. o nazwie `physics.h` zawierający następujące *deklaracje*:

```
/* physics.h - deklaracje podstawowych stałych i funkcji fizycznych */
#ifndef PHYSICS_H
#define PHYSICS_H

extern const double G;
extern const double EARTH_MASS;

extern double compute_attractive_force(double m1, double m2, double r);

#endif //PHYSICS_HPP
```

oraz powiązany z nim plik źródłowy `physics.c` zawierający *definicje* symboli wprowadzonych w pliku `physics.h`:

```
/* physics.c - definicje stalych i funkcji z pliku physics.h */
#include "physics.h"

const double G = 6.6743015e-11;  // [m^3 * kg^-1 * s^-2]
const double EARTH_MASS = 5.9722e24;  // [kg]

double compute_attractive_force(double m1, double m2, double r) {
    return G * m1 * m2 / (r * r);
}
```

Pliku nagłówkowego `physics.h` możesz następnie użyć w poniższym przykładowym programie:

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

int main(void) {
    double object_mass = 1;  // [kg]
    double distance = 6373.14 * 1000;  // [m]
    double f = compute_attractive_force(EARTH_MASS, object_mass, distance);
    printf("F = %.2f N\n", f);

    return EXIT_SUCCESS;
}
```

którego efekt uruchomienia wygląda tak:

```
F = 9.81 N
```

Zwróć uwagę, że niestandardowe pliki nagłówkowe dołącza się za pomocą poniższej wersji dyrektywy **#include**:

```
#include "sample_header.h"  // Zwróć uwagę na cudzysłowy zamiast
                            //   nawiasów trójkątnych!
```

Cudzysłowy informują preprocesor, żeby zaczął poszukiwanie żądanego pliku (tu: `sample_header.h`) począwszy od lokalizacji wskazanych w opcjach kompilacji (domyślnie: od bieżącego katalogu, w którym został wywołany kompilator), a dopiero później w lokalizacjach standardowych (zawierających m.in. pliki nagłówkowe biblioteki standardowej).

Użycie słowa kluczowego **extern** np. w instrukcji **extern int** `x;` mówi kompilatorowi tyle, że „(gdzieś) istnieje obiekt typu **int** o identyfikatorze `x`". Rolą kompilatora nie jest sprawdzanie, gdzie dokładnie wspomniany obiekt istnieje – kompilatorowi potrzebna jest jedynie informacja o typie i nazwie tego obiektu, aby wiedzieć jak go użyć. Gdy wszystkie pliki źródłowe zostaną skompilowane, wówczas konsolidator uzgodni wszystkie odniesienia do `x` w programie z użyciem jednej definicji, którą odnajdzie w jednym ze skompilowanych plików źródłowych. Aby proces ten przebiegł pomyślnie, definicja zmiennej `x` musi posiadać łączność zewnętrzną (czyli w praktyce `x` musi być zdefiniowane jako obiekt globalny – o zasięgu plikowym – oraz bez użycia specyfikatora **static**).

14.2 Strażnik nagłówka (*header guard*)

Aby zabezpieczyć się przed kilkukrotnym dołączaniem elementów tego samego pliku nagłówkowego w ramach tego samego pliku źródłowego (co wydłuża proces kompilacji, a w przypadku definicji powoduje błędy wielokrotnej definicji), *zawsze* korzystaj w tworzonych przez siebie plikach nagłówkowych z tzw. **strażnika nagłówka** (ang. #include guard, header guard):

```
/* sample_header.h */
#ifndef UNIKALNE_ID_PLIKU_
#define UNIKALNE_ID_PLIKU_
// Powyższy identyfikator (UNIKALNE_ID_PLIKU_) można wybrać dowolnie,
//   zwykle IDE samo umieszcza stosowny "unikalny" identyfikator.

// główna zawartość pliku nagłówkowego
// ...

#endif
```

Załóżmy, że powyższy plik z jakiegoś powodu został dołączony kilka razy. Przy pierwszym przetworzeniu pliku `sample_header.h` stała `UNIKALNE_ID_PLIKU_` jest niezdefiniowana, zatem preprocesor definiuje ją, a następnie wykonuje pozostałą część pliku. Jednak za kolejnym razem stała `UNIKALNE_ID_PLIKU_` jest już zdefiniowana, a więc preprocesor pomija cały kod pomiędzy dyrektywami **#ifndef** i **#endif**.

Identyfikator `UNIKALNE_ID_PLIKU_` musi być unikalny w obrębie całego programu, gdyż preprocesor nie respektuje zasad widoczności języka C; sporą gwarancję niepowtarzalności daje użycie nazwy pliku nagłówkowego pisanej wielkimi literami[1], zawierającej znaki podkreślenia zamiast kropek oraz rozpoczynającej się i kończącej znakiem podkreślenia (lub dwoma takimi znakami).

Ważne

Dyrektywa **`#ifndef`** powinna znaleźć się w *pierwszym* wierszu pliku, a dyrektywa **`#endif`** – w *ostatnim* (tak, aby *cała* zawartość pliku nagłówkowego była objęta „strażnikiem").

Próba wielokrotnego dołączania tego samego pliku źródłowego wynika zwykle z faktu, że wiele plików nagłówkowych dołącza inne pliki (przykładowo: plik `crtdefs.h` jest dołączany w plikach `stdlib.h` i `stdio.h`, zatem większość z pisanych przez Ciebie programów dołącza dwukrotnie plik `crtdefs.h`).

Dla zainteresowanych...

Krótkie omówienie mechanizmu strażnika nagłówka znajdziesz tu.

14.3 Pliki źródłowe a pliki nagłówkowe

Do dobrych praktyk (m.in. ułatwiających nawigację w projekcie) należy rozdzielenie deklaracji od implementacji poprzez umieszczenie ich w osobnych plikach:

- Pliki nagłówkowe (`.h`[2]) służą do udostępniania informacji o:
  - stałych symbolicznych (np. plik `stdlib.h` definiuje stałą `EXIT_SUCCESS`)
  - makrach (np. plik `ctype.h` definiuje niemal wszystkie funkcje jako makra)
  - *prototypach* funkcji
  - *deklaracjach* obiektów
  - definicjach niestandardowych typów: aliasów, typów wyliczeniowych, szablonów struktur i szablonów unii (np. alias `size_t`)
- Pliki źródłowe (`.c`) służą do udostępniania informacji o *definicjach* funkcji i obiektów.

Do weryfikacji poprawności kodu pliku źródłowego korzystającego z „zewnętrznej" funkcji kompilator nie potrzebuje znać jej pełnej implementacji, a jedynie jej prototyp – dzięki prototypowi jest w stanie sprawdzić zgodność typów argumentów z typami parametrów oraz typ wartości zwracanej z typem wyrażenia, w którym zostało użyte wywołanie tej funkcji.

Oto powody, dla których w pliku nagłówkowym *nie należy* umieszczać definicji funkcji i obiektów:

- Nawet zastosowanie strażnika nagłówka nie zapobiega problemowi redefinicji symbolu w sytuacji, gdy dwa różne pliki źródłowe dołączają ten sam plik nagłówkowy zawierający definicję funkcji lub symbolu – wówczas każdy z plików źródłowych będzie posiadał własną „kopię" definicji takiego globalnego symbolu, co nie jest problemem na etapie kompilacji (gdyż każdy plik źródłowy jest kompilowany oddzielnie), lecz spowoduje błąd na etapie konsolidacji programu.
- Jakakolwiek zmiana w pliku nagłówkowym wymusza ponowną kompilację programu u wszystkich użytkowników tego pliku (w przeciwnym razie wprowadzone zmiany nie będą w nich widoczne) – a znacznie częściej dochodzi do zmiany ciała funkcji niż jej nagłówka...
- Z implementowaniem funkcjonalności zazwyczaj wiąże się konieczność dołączenia wielu dodatkowych plików nagłówkowych – umieszczenie ich w „naszym" pliku nagłówkowym powodowałoby niepotrzebne uzależnianie jego użytkowników od posiadania tych dodatkowych bibliotek.
- Dołączanie pliku nagłówkowego polega na skopiowaniu całej zawartości takiego pliku tekstowego i wklejenie jej w miejsce dyrektywy **`#include`** – plik nagłówkowy powinien zawierać możliwie mało elementów, aby usprawnić proces preprocessingu.

Dla zainteresowanych...

Dobre praktyki związane z tworzeniem plików nagłówkowych znajdziesz na stronie C++ Header File Guidelines (David Kieras, Univ. of Michigan).

[1] Zgodnie z konwencją nazwy stałych są pisane wielkimi literami.

[2] Rozszerzenie `.h` pochodzi od angielskiego terminu *header file* oznaczającego plik nagłówkowy.

14.4 Błąd umieszczenia definicji funkcji lub obiektu w pliku nagłówkowym

Aby przekonać się jakie problemy powoduje umieszczenie definicji funkcji lub obiektu w pliku nagłówkowym spróbuj zbudować program składający się z pliku nagłówkowego `myheader.h`:

```
/* myheader.h */
#ifndef MYHEADER_H_
#define MYHEADER_H_

void foo(void) {}

#endif //MYHEADER_H_
```

oraz dwóch plików źródłowych korzystających z tego pliku nagłówkowego – pliku `src1.c`:

```
/* src1.c */
#include "myheader.h"
```

oraz pliku `main.c`:

```
/* main.c */
#include <stdlib.h>
#include "myheader.h"

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

Proces kompilacji przebiega poprawnie, natomiast na etapie konsolidacji pojawia się błąd:

```
(...)/objects.a(src1.c.obj): In function `foo':
(...)/myheader.h:5: multiple definition of `foo'
(...)/objects.a(main.c.obj):(...)/myheader.h:5: first defined here
```

gdyż konsolidator natrafia na (potencjalnie różne) definicje tego samego symbolu `foo` w różnych plikach obiektowych i nie jest w stanie wybrać odpowiedniego powiązania.

Błędy postaci „`multiple definition of `...'`" bardzo często wynikają właśnie z umieszczenia definicji funkcji lub obiektu w pliku nagłówkowym.

14.5 Kolejność deklaracji

Ponieważ kompilator analizuje kod jednostki translacji tylko raz, w kolejności od pierwszego do ostatniego wiersza, deklaracje i definicje muszą występować w odpowiednim porządku – deklaracje zmiennych i funkcji oraz definicje typów muszą pojawiać się przed pierwszym użyciem identyfikatora zmiennej, funkcji lub typu. Oto *błędna* kolejność:

```
#include <stdlib.h>

int main(void) {
    foo();  // użycie identyfikatora `foo`, ALE analizując ten wiersz
            //   kompilator nie wie jeszcze, czym jest `foo`...

    return EXIT_SUCCESS;
}

void foo(void) {}  // definicja funkcji o identyfikatorze `foo`
```

Natomiast to kolejność poprawna:

```
#include <stdlib.h>

void foo(void);  // deklaracja poprzedzająca funkcji o identyfikatorze `foo`

int main(void) {
```

```
    foo();  // użycie identyfikatora `foo`

    return EXIT_SUCCESS;
}

void foo(void) {}  // definicja funkcji o identyfikatorze `foo`
```

Rozdział 15

Zarządzanie pamięcią

W języku C można wyróżnić dwie podstawowe metody alokacji (przydzielania) pamięci:

- **alokacja statyczna** – ilość pamięci zostaje z góry określona na etapie pisania programu (poprzez odpowiednie deklaracje zmiennych); pamięć jest zwalniana automatycznie – po zakończeniu bloku programu, w którym była deklarowana (chyba, że obiekt został zadeklarowany jako `static`), lub po zakończeniu programu

```
int a;
char str[] = "Hello!";
float tab[5];
```

- **alokacja dynamiczna** – program przydziela dowolne ilości pamięci w trakcie pracy, poprzez wywołanie odpowiednich funkcji; pamięć musi być jawnie zwolniona (z użyciem odpowiedniej funkcji)

```
int* tab = malloc(5 * sizeof(*tab));
free(tab);
```

Dynamiczna alokacja pamięci pozwala przede wszystkim na:

- utworzenie tablicy o rozmiarze obliczanym dopiero podczas działania programu
- dostęp do zmiennych utworzonych wewnątrz funkcji po wyjściu z nich

15.1 Przydzielanie pamięci – funkcja `malloc()`

Funkcja `malloc()` (od ang. memory allocation) o poniższym prototypie

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

rezerwuje blok wolnej pamięci o zadanym rozmiarze (w bajtach) i zwraca adres jego pierwszego bajtu. Ponieważ rezerwowany blok może być zmienną dowolnego typu, funkcja ta zwraca uniwersalny „wskaźnik na `void`". Przypisanie takiego wskaźnika do wskaźnika dowolnego innego typu jest operacją dozwoloną przez standard języka, lecz ze względu na czytelność zalecane jest użycie rzutowania.

Aby utworzyć tablicę typu `double` o zadanym rozmiarze z użyciem funkcji `malloc()` można napisać następujący kod:

```
// dynamiczna alokacja tablicy 10 elementów typu `double`
int rozmiar = 10;
double* tab = (double*) malloc(rozmiar * sizeof(*tab));
```

Powyższy kod zarezerwuje obszar pamięci dla 10 wartości typu wskazywanego przez `tab` (czyli `double`) i przypisze adres wskaźnikowi `tab`. Zauważ, że zmienna `tab` została zadeklarowana jako wskaźnik do pojedynczej wartości `double`, a nie do bloku 10 takich wartości. Dlaczego? Ponieważ nazwa tablicy jest zarazem adresem jej pierwszego elementu, powyższe przypisanie umożliwia korzystanie z `tab` jak ze zwykłej nazwy tablicy (np. `tab[0]` to wartość jej pierwszego elementu). Zauważ, że powyższe wywołanie funkcji `malloc()` jest poprawne, gdyż w momencie obliczania wartości jej argumentów zmienna `tab` już została zdefiniowana.

W przypadku nieznalezienia wolnego obszaru pamięci o zadanym rozmiarze funkcja `malloc()` zwraca wskaźnik zerowy (czyli – dla przypomnienia – wskaźnik o wartości `0`) – należy taką sytuację obsłużyć w programie, np.:

```
if (tab == NULL) {
    printf("Blad przydziału pamięci.\n");
    exit(EXIT_FAILURE);
}
```

15.2 Zwalnianie pamięci – funkcja `free()`

Funkcja `free()` o poniższym prototypie

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

zwalnia dynamicznie przydzieloną pamięć (i tylko taką – próba zwalniania pamięci przydzielonej statycznie to błąd). Argumentem funkcji `free()` jest adres obszaru pamięci do zwolnienia. Na przykład:

```
double* tab = malloc(8 * sizeof(*tab));
free(tab);
tab = NULL;  // nieobowiązkowe, ALE dobra praktyka
```

Do dobrej praktyki należy „wyzerowanie” wskaźnika po zwolnieniu miejsca wskazywanego przez niego miejsca w pamięci (aby mieć gwarancję, że nie odwołamy się przypadkiem ponownie do tego miejsca).

15.3 Zmiana rozmiaru bloku pamięci – funkcja `realloc()`

Funkcja `realloc()` o poniższm prototypie

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

służy do zmiany rozmiaru zaalokowanego dynamicznie bloku pamięci. Jej pierwszym argumentem jest adres bloku pamięci, a drugim – nowy rozmiar (w bajtach).

Do zwalniania pamięci przydzielonej przez `realloc()` służy również funkcja `free()`.

Oto przykład wykorzystania funkcji `realloc()`:

```
double* tab;
int rozmiar_1 = 10;
int rozmiar_2 = 20;

tab = malloc(rozmiar_1 * sizeof(*tab)); // pierwotnie `tab` ma rozmiar 10 bajtow
tab = realloc(tab, rozmiar_2 * sizeof(*tab)); // teraz `tab` ma rozmiar 20 bajtow
free(tab);
```

15.4 Alokacja pamięci dla tablicy dwuwymiarowej

Aby utworzyć dynamiczną tablicę dwuwymiarową możemy postąpić w następujący sposób:
Najpierw alokujemy pamięć dla „wierszy", a następnie dla „kolumn". Podczas zwalniania pamięci postępujemy na odwrót – najpierw zwalniamy pamięć przydzieloną dla „kolumn", a potem dla „wierszy".

Oto przykład dynamicznej alokacji tablicy dwuwymiarowej:

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

int main ()
{
    const int w = 6;  //liczba wierszy
    const int k = 3;  // liczba kolumn

    double** tab2d = malloc(w * sizeof(double*));  // alokacja "wierszy"

    for(int i = 0; i < w; i++) {
        tab2d[i] = malloc(k * sizeof(double));  // alokacja "kolumn"
```

```
    }

    for(int i = 0; i < w; i++) {
        free(tab2d[i]); // zwolnienie pamieci kolumn"
    }

    free(tab2d); //uwolnienie pamieci ,,wierszy"
    tab2d = NULL;

    return EXIT_SUCCESS;
}
```

[OPT] Inne sposoby tworzenia dynamicznych tablic dwuwymiarowych przestawione są w artykule How to dynamically allocate a 2D array in C?.

15.5 Funkcja `calloc()`

Kolejną możliwością przydzielania pamięci jest skorzystanie z funkcji `calloc()` o poniższym prototypie:

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

Funkcja `calloc()` działa bardzo podobnie do funkcji `malloc()`, z tym że:

- zamiast podawać rozmiar całego bloku pamięci (w bajtach) podaje się rozmiar pojedynczego elementu i liczbę elementów, oraz
- wszystkie *bity* zarezerwowanego bloku są zerowane.

15.6 Klasy zmiennych a dynamiczne przydzielanie pamięci

W wyidealizowanym modelu zarządzania pamięcią program dzieli dostępną pamięć na trzy oddzielne obszary:

- jeden dla zmiennych zewnętrznych i statycznych,
- jeden dla zmiennych automatycznych i
- jeden dla bloków przydzielanych dynamicznie.

Ilość pamięci potrzebna do przechowania zmiennych zewnętrznych i statycznych jest znana w momencie kompilacji, a dane przechowywane w tym obszarze są dostępne przez cały czas działania programu. Pamięć niezbędna do przechowania każdej zmiennej należącej do jednej z tych klas zostaje przydzielona w momencie uruchomienia programu, a zwolniona przy jego zakończeniu.

Z kolei pamięć dla zmiennej automatycznej zostaje przydzielona, gdy program wejdzie do bloku kodu zawierającego definicję zmiennej, a zwolniona w chwili opuszczenia tego bloku – tym samym ilość pamięci wykorzystywana przez zmienne automatyczne zmienia się w czasie pracy programu. Obszar pamięci przeznaczony dla danych automatycznych przyjmuje zazwyczaj postać stosu. Oznacza to, że zmienne są usuwane z pamięci w odwrotnej kolejności niż były do niej dodawane.

Pamięć przydzielana dynamicznie pojawia się w momencie wywołania funkcji z rodziny `*alloc()`[1], a znika w wyniku wywołania funkcji `free()` – zatem to programista kontroluje okres trwania zmiennej zaalokowanej dynamicznie. W szczególności możliwe jest dynamiczne przydzielenie bloku pamięci w jednej funkcji i usunięcie go w innej (z tego powodu obszar pamięci przeznaczony do alokacji dynamicznej może ulec fragmentacji: aktywne bloki pamięci mogą przeplatać się z blokami nieużywanymi). Warto zwrócić uwagę, że korzystanie z pamięci dynamicznej jest z reguły wolniejsze niż korzystanie z pamięci zorganizowanej w formie stosu.

15.7 Typ efektywny obiektu

Każdy obiekt posiada tzw. **typ efektywny** (ang. effective type) określający rodzaj które przypisania do niego są poprawne, a które naruszają zasady ścisłego aliasingu[2]:

1. Jeśli obiekt został utworzony z użyciem deklaracji, jego deklarowany typ odpowiada typowi efektywnemu. Jeśli jednak obiekt został utworzony z użyciem jednej z funkcji alokujących (w tym funkcji

[1] czyli: `malloc()`, `calloc()` i `realloc()`
[2] Zasady te zostały omówione w rozdz. 11.10.

`realloc()`), nie posiada on żadnego zadeklarowanego typu (funkcje te zwracają wartość typu `void*`) i wówczas jego typ efektywny jest wyznaczany w sposób opisany w kolejnych punktach.

2. Pierwsze przypisanie do obiektu nie posiadającego typu efektywnego l-wartości typu innego niż typ znakowy sprawia, że typ tej l-wartości staje się efektywnym typem obiektu (dla tego przypisania oraz dla wszystkich dalszych odczytów tego obiektu).
3. Użycie funkcji `memcpy()` i `memmove()` sprawia, że efektywny typ obiektu docelowego (tj. do którego odbywa się kopiowanie), który obecnie nie posiada zadeklarowanego typu, odpowiada efektywnemu typowi obiektu źródłowego (dla tego przypisania oraz dla wszystkich dalszych odczytów obiektu docelowego) – oczywiście o ile obiekt źródłowy posiadał taki typ.
4. W przypadku każdego innego dostępu do obiektu bez zadeklarowanego typu, efektywnym typem obiektu jest typ l-wartości użytej do uzyskania dostępu.

Poniższe dwa przykłady pokazują, jak reguły wyznaczania typu efektywnego przekładają się na praktykę („n/d” użyte w omówieniach kodów przykładów oznacza, że dana reguła „nie dotyczy”).

Przykład 1

```
unsigned int* p = malloc(sizeof *p);
memset(p, 0x55, sizeof *p);
unsigned int u = *p;
```

- `memset(...);`
  - 1: n/d (brak zadeklarowanego typu);
  - 2: n/d (`memset()` zapisuje z użyciem semantyki typu znakowego);
  - 3: n/d (to ani nie funkcja `memcpy()` ani `memmove()`);
  - 4: n/d (wewnętrznie, tymczasowo, `memset()` traktuje dane jako typu `char[]`)
- `unsigned int u = *p;`
  - 1: n/d (brak zadeklarowanego typu);
  - 2, 3: n/d (nie mamy do czynienia z zapisem, tylko z odczytem);
  - 4: typ l-wartości to `unsigned int`
- Wniosek: Inicjalizacja zmiennej `u` nie narusza zasad ścisłego aliasingu, jednak interpretacja przypisanej do `u` wartości zależy od implementacji (m.in. od wyrównania bajtów i ich kolejności).

Przykład 2

```
void* d = malloc(50);
*(double*) d = 1.23;
memset(d, 0x55, 50);
unsigned int u = *(unsigned int *) d;
```

- `void* d = malloc(50);`: obiekt `d` nie posiada zadeklarowanego typu
- `*(double *)d = 1.23;`: 2 – efektywny typ `d` to teraz `double*`
- `memset(...);`: analogiczna sytuacja, jak w przykładzie 1
- `unsigned int u = *(unsigned int *) d;`: efektywny typ `d` to wciąż `double*` – **niedozwolone**
- Wniosek: Inicjalizacja zmiennej `u` narusza zasady ścisłego aliasingu.

15.8 Organizacja pamięci programu

Pamięć jest podzielona na trzy podstawowe segmenty – pamięć statyczną, stos i stertę:

- **Pamięć statyczna** służy do przechowywania zmiennych o statycznym okresie trwania.
- **Stos** (ang. stack) to miejsce służące do przechowywania zmiennych o automatycznym okresie trwania. W ogólnym ujęciu stos jest strukturą danych typu LIFO (Last-In-First-Out) – ostatni element umieszczony na stosie będzie jednocześnie pierwszym, który z niego zdejmiesz; udostępnia ona dwie przeciwstawne operacje: *push* („umieść element na stosie”) i *pop* („zdejmij element ze stosu”). W momencie, gdy definiowana jest nowa zmienna o automatycznym okresie trwania, zostaje wykonana operacja *push* (powodująca przydzielenie pamięci); gdy zmienna ta jest usuwana, wykonywana jest

operacja *pop* (powodująca zwolnienie pamięci). Zarządzaniem stosem zajmuje się procesor, a nie programista. Stos ma ograniczoną, niewielką pojemność (różniącą się pomiędzy systemami operacyjnymi, lecz domyślnie – dla komputerów osobistych – rzędu kilku megabajtów); próba umieszczenia na stosie zbyt wielu zmiennych powoduje **przepełnienie stosu** (ang. stack overflow) – taki błąd pojawia się m.in. w przypadku błędnego użycia rekurencji.

- **Sterta** (ang. heap) to pula pamięci dużych rozmiarów, która może być używana w sposób dynamiczny. Zarządzaniem stertą zajmuje się programista, przydzielając i zwalniając pamięć (np. z użyciem odpowiednio funkcji z rodziny `*alloc()` i funkcji `free()`). Do obiektów zaalokowanych na stercie można się odwoływać w dowolnym miejscu w programie. W przeciwieństwie do stosu, co do zasady nie ma innych ograniczeń na rozmiar sterty niż fizyczny rozmiar pamięci operacyjnej.

15.9 Problemy związane z (nieumiejętnym) zarządzaniem pamięcią

Nieumiejętne zarządzanie pamięcią prowadzi do występowania licznych poważnych problemów, takich jak wycieki pamięci oraz błędy naruszenia ochrony pamięci. Na szczęście istnieją narzędzia pozwalające diagnozować takie problematyczne sytuacje.

15.9.1 Wycieki pamięci

W sytuacji, gdy przydzielony dynamicznie („ręcznie”) blok pamięci nie został zwolniony, a utracona została informacja o położeniu takiego bloku (co uniemożliwia jego zwolnienie), mówimy o **wycieku pamięci** (ang. memory leak) – taki blok pamięci będzie traktowany jako „zajęty” i nie będzie możliwości jego ponownego użycia przez ten sam lub inne procesy. Do wycieku pamięci dochodzi na przykład w każdym z poniższych trzech przypadków:

- brak zapamiętania adresu dynamicznie przydzielonego bloku pamięci, np.

```
malloc(1);
```

- utrata adresu dynamicznie przydzielonego bloku pamięci poprzez nadpisanie, np.

```
void* ptr = malloc(1);
ptr = NULL;  // utrata adresu
```

- utrata adresu dynamicznie przydzielonego bloku pamięci poprzez wyjście z bloku, np.

```
void foo(void) {
    void* ptr = malloc(1);
    /* inne instrukcje, ale nie `free(ptr)` */
}
```

Zwróć uwagę, że choć wyciek 1 bajta pamięci z pozoru nie wygląda groźnie, w przypadku programu działającego przez długi czas lub często wykonującego fragment kodu marnujący pamięć może się to z czasem skończyć błędem wyczerpania pamięci sterty!

Ważne

System operacyjny wyznacza limit pamięci, jaki może wykorzystać dany program – jego przekroczenie powoduje „ubicie” programu. Zapominanie o zwolnieniu przydzielonej pamięć prowadzi do wycieku pamięci i w konsekwencji może skutkować przedwczesnym zakończeniem programu (przez system operacyjny).

15.9.2 Błędy naruszenia ochrony pamięci

Do innych częstych przyczyn błędów związanych z zarządzaniem pamięcią, które mogą doprowadzić do wystąpienia błędu **naruszenia ochrony pamięci**[3] (ang. memory access violation) w żargonie określanego jako **segfault** (od ang. segmentation fault), należą:

[3] Błędy te są zgłaszane przez sprzęt posiadający mechanizm ochrony pamięci – sygnalizuje on systemowi operacyjnemu próby dostępu do „chronionych” obszarów pamięci, czyli np. obszarów pamięci, do których dany proces nie powinien mieć dostępu.

- Odwoływanie się do elementów o indeksach spoza zakresu danej tablicy, np.:

```
int arr[3];
arr[-1] = 0;  // BŁĄD
```

- Niezainicjalizowanie wskaźnika poprawnym adresem przed pierwszą próbą dostępu do wskazywanych przez niego danych – czyli próba skorzystania z tzw. „dzikiego" wskaźnika (ang. wild pointer) wskazującego na losowe miejsce w pamięci – np.:

```
int* ptr;
*ptr = 0;  // BŁĄD
```

- Próba dereferencji wskaźnika pustego[4], np.:

```
int* ptr = NULL;
*ptr = 0;  // BŁĄD
```

- Próba uzyskania dostępu do przydzielonego bloku pamięci po jego zwolnieniu – w szczególności próba ponownego zwolnienia tego samego bloku dynamicznie zaalokowanej pamięci, np.:

```
int* ptr = (int*) malloc(sizeof(int));
free(ptr);
*ptr = 0;   // BŁĄD
free(ptr);  // BŁĄD
```

  W powyższym przykładzie po wykonaniu pierwszej instrukcji `free(ptr)` wskaźnik `ptr` staje się tzw. **wiszącym wskaźnikiem** (ang. dangling pointer) – czyli wskaźnikiem wskazującym na miejsce pamięci, które zostało zwolnione.

Większość z powyższych błędów wynika z próby dostępu do „losowego" miejsca w pamięci: do „nieistniejącego" miejsca w pamięci (tj. poza przestrzenią adresową danego procesu) lub do obszaru pamięci procesu „tylko do odczytu" (np. do segmentu kodu).

15.9.3 Diagnostyka błędów

Do identyfikowania problemów z zarządzaniem pamięcią służy np. darmowe narzędzie Valgrind – sprawdza ono ile pamięci zostało przydzielone, ile zostało uwolnione oraz wychwytuje próby nieuprawnionego dostępu do w miejsc w pamięci.

[4] Wskaźnik pusty zwykle wskazuje na miejsce poza przestrzenią adresową danego procesu.

Rozdział 16

Formatowane wejście: funkcja `scanf()`

Naturalnie dane wprowadzane ze standardowego wejścia (zazwyczaj z klawiatury) są zawsze tekstem, ponieważ klawisze generują znaki tekstowe: litery, cyfry i znaki przestankowe. Gdy chcesz wpisać np. liczbę całkowitą 2002, wciskasz kolejno klawisze 2, 0, 0 i 2. Jeśli jednak chcesz przechować wpisany tekst jako wartość liczbową, a nie jako łańcuch, konieczna jest konwersja – i właśnie do tego służy funkcja `scanf()`. Przetwarza ona tekst wejściowy na wartości różnych typów: liczby całkowite, liczby zmiennoprzecinkowe, znaki i łańcuchy znakowe. Jest ona przeciwieństwem funkcji `printf()`, która przetwarza liczby całkowite, liczby zmiennoprzecinkowe, znaki i łańcuchy znakowe na tekst przeznaczony do wyświetlenia na ekranie.

16.1 Funkcja `scanf()`: ogólna postać

Prototyp funkcji `scanf()` wygląda następująco:

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

przy czym zgodnie ze standardem języka C zapis `...` oznacza, że funkcja może przyjmować tzw. zmienną liczbę argumentów[1].

Korzystanie z niej przypomina zatem korzystanie z funkcji `printf()`, z tym że oprócz łańcucha sterującego określającego sposób interpretacji znaków odczytanych ze standardowego wejścia `scanf()` przyjmuje (jako dalsze argumenty) *wskaźniki* do miejsc w pamięci, w których mają zostać zapisane odczytane (i zinterpretowane) dane – każdemu specyfikatorowi konwersji powinien odpowiadać dokładnie jeden argument-wskaźnik.

Aby podzielić łańcuch wejściowy na pojedyncze pola (wartości) funkcja `scanf()` wykorzystuje znaki niedrukowane (tj. znaki nowej linii, tabulatory i odstępy). Dopasowuje ona kolejne specyfikatory konwersji do kolejnych pól, pomijając wszystkie znaki niedrukowane znajdujące się pomiędzy polami. (Jedynym wyjątkiem od tej zasady jest specyfikator `%c`, który zawsze odczytuje jeden znak, nawet jeśli jest to znak niedrukowany.)

Ważne

Podstawowe specyfikatory konwersji są analogiczne do wykorzystywanych w funkcji `printf()`, z tym że `scanf()` dodatkowo *wymusza* stosowanie `%lf` do wczytywania wartości typu `double`.

Oto przykład programu zawierającego instrukcję `scanf()`, wczytującą identyfikator i współrzędne punktu na płaszczyźnie:

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

int main(void) {
    int id;
    double x, y;

    printf("Podaj ID i wspolrzedne punktu: ");
    scanf("%d %lf %lf", &id, &x, &y);
```

[1] zob. variadic functions

```
    printf("Odczytano: P#%d = [%f, %f]\n", id, x, y);

    return EXIT_SUCCESS;
}
```

Jego przykładowe uruchomienie wygląda następująco:

```
Podaj ID i wspolrzedne punktu: 1 2.0 3.4
Odczytano: P#1 = [2.000000, 3.400000]
```

16.2 Dane wejściowe z punktu widzenia funkcji `scanf()`

W jaki funkcja `scanf()` odczytuje dane wejściowe? Funkcja `scanf()` odczytuje dane znak po znaku. Załóżmy, że użyliśmy specyfikatora `%d`, aby pobrać wpisaną przez użytkownika liczbę całkowitą.

- Jeśli na początku łańcucha wejściowego występują znaki niedrukowane (odstępy, tabulatory lub znaki nowej linii), zostaną one pominięte. Ponieważ zleciliśmy pobranie liczby całkowitej, funkcja `scanf()` oczekuje, że pierwszy znak drukowany w łańcuchu wejściowym będzie cyfrą, znakiem plus lub znakiem minus. Jeśli pierwszy znak drukowany spełnia ten warunek, zostanie on zachowany, a funkcja odczyta kolejny znak z łańcucha. Proces ten będzie kontynuowany aż do momentu odczytania znaku nie będącego cyfrą – wówczas funkcja `scanf()` uzna, że dotarła do końca liczby całkowitej i umieści ostatnio pobrany znak z powrotem w łańcuchu wejściowym. Oznacza to, że ewentualne pobieranie dalszych danych przez program rozpocznie się od tego właśnie odrzuconego znaku. Na końcu funkcja `scanf()` oblicza wartość liczbową odpowiadającą pobranym cyfrom i umieszcza ją w zmiennej.
- Co się stanie, jeśli pierwszym znakiem drukowanym jest nie cyfra, ale na przykład litera „A”? w takim przypadku funkcja `scanf()` natychmiast zatrzyma odczytywanie danych i umieści znak „A” z powrotem w łańcuchu wejściowym. Zmienna nie otrzyma żadnej wartości, a jeśli program pobiera dalsze dane, pierwszym wczytanym znakiem będzie właśnie „A”. Jeśli w programie występują wyłącznie specyfikatory `%d`, funkcja `scanf()` nigdy nie odczyta nic poza literą „A”. Istotne jest także, że nawet jeśli instrukcja `scanf()` zawiera kilka specyfikatorów, to – zgodnie ze standardem jezyka C – odczytywanie danych zostanie zatrzymane w momencie wystąpienia pierwszego błędu.

Specyfikator `%s` powoduje odczytanie przez funkcję `scanf()` dokładnie jednego słowa (łańcucha nie zawierającego znaków niedrukowanych). Funkcja `scanf()`, umieszczając łańcuch w tablicy znaków, dodaje na jego końcu znak zerowy `\0` zgodnie z wymaganiami języka C.

W przypadku użycia specyfikatora `%c` przyjmowane są *wszystkie* wpisywane znaki (także znaki niedrukowane). Jeśli odczytany znak jest odstępem lub znakiem nowej linii, to właśnie odstęp lub znak nowej linii zostanie przypisany zmiennej.

16.3 Zwykłe znaki w łańcuchu sterującym

Funkcja `scanf()` pozwala umieszczać w łańcuchu sterującym również zwykłe znaki, przy czym wszystkie takie znaki (z wyjątkiem odstępów) muszą równocześnie znajdować się w łańcuchu wejściowym. Przykładowo jeśli między dwoma specyfikatorami znajduje się przecinek:

```
scanf("%d, %d", &n, Sm);
```

dla funkcji `scanf()` będzie to oznaczało, że najpierw wpisana zostanie liczba, potem przecinek, a następnie druga liczba. Ponieważ przecinek znajduje się bezpośrednio po symbolu `%d`, należy wpisać go zaraz po liczbie 88. Ponieważ jednak funkcja `scanf()` pomija znaki niedrukowane, jeśli występują one przed liczbą, po przecinku możnaby wstawić odstęp lub znak nowej linii. Zatem przykładowe (poprawne) dane to m.in.: `1,2` oraz `1, 2`.

16.4 Wartość zwracana przez funkcję `scanf()`

Funkcja `scanf()` zwraca liczbę poprawnie odczytanych pozycji. Jeśli nie powiodło się odczytanie żadnego elementu – co może się zdarzyć choćby w przypadku, gdy funkcja `scanf()` spodziewała się liczby, a wpisany został łańcuch składający się z liter – wartością zwracaną jest 0.

16.5 Czyszczenie bufora wejściowego

W przypadku, kiedy przykładowo nie udało się odczytać wszystkich danych z bufora wejściowego – gdyż nie odpowiadały one specyfikatorom konwersji – część z tych danych pozostaje w buforze, co może być zachowaniem niepożądanym:

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

int main(void) {
    char str[80], ch;

    scanf("%s", str);
    ch = getchar();

    printf("STR = \"%s\"\n", str);
    printf("CH = '%c'", ch);

    return EXIT_SUCCESS;
}
```

Wynik przykładowego wykonania powyższego programu:

```
abc
abc
STR = "abc"
CH = '
'
```

Jak widzisz, wywołanie funkcji `scanf()` pozostawiło w buforze znak nowej linii, który następnie został odczytany przez funkcję `getchar()`.

Aby wyczyścić bufor, można użyć poniższej pętli odczytującej znaki do momentu napotkania znaku nowej linii (generowanego przez wciśnięcie klawisza Enter) włącznie lub znaku EOF (end-of-file) oznaczającego, że nie ma więcej znaków do odczytania:

```
int c;
while ((c = getchar()) != '\n' && c != EOF) { }
```

Rozdział 17

Łańcuchy znakowe i funkcje łańcuchowe

W tym rozdziale dowiesz się więcej o tworzeniu i używaniu łańcuchów. Nauczysz się korzystać z kilku funkcji łańcuchowych i znakowych z biblioteki języka C.

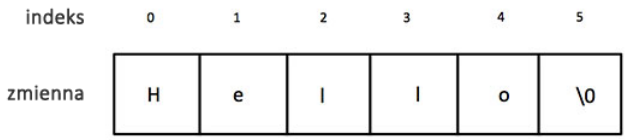
17.1 Czym jest łańcuch znaków?

Łańcuch znakowy jest ciągiem składającym się z jednego lub więcej znaków. Oto przykład łańcucha:

```
"Ala ma kota."
```

Znaki cudzysłowu nie są częścią łańcucha – informują one jedynie kompilator, o tym, że fragment tekstu objęty cudzysłowami jest łańcuchem (podobnie jak apostrofy sygnalizują stałą znakową).

Język C nie posiada specjalnego typu łańcuchowego – zamiast tego łańcuchy przechowywane są w tablicach elementów typu **char**. Kolejne znaki, z których składa się łańcuch, znajdują się w kolejnych komórkach pamięci, a po nich znajduje się tzw. znak zerowy (zob. rys. 17.1).



Rysunek 17.1. Reprezentacja łańcucha znaków `"Hello"` w pamięci (jako tablica elementów typu **char**, zakończona znakiem zerowym)

Znak zerowy (ang. null character) nie jest cyfrą zero, tylko znakiem niedrukowanym o kodzie ASCII równym 0 (jego literał to `'\0'`). Łańcuchy w języku C są przechowywane razem z tym znakiem. Oznacza to, że tablica musi mieć długość przynajmniej o jeden element większą niż długość zapisanego w niej łańcucha. Nie pomyl wskaźnika zerowego ze znakiem zerowym – choć obu obiektom odpowiada wartość 0, to różnią się one z formalnego punktu widzenia:

- wskaźnik zerowy wyraża adres (obiekt typu **int**)
- znak zerowy wyraża. . . znak (obiekt typu **char**)

Literał łańcuchowy, inaczej **stała łańcuchowa** (ang. string constant), jest ciągiem znaków zawartym pomiędzy znakami cudzysłowu. Ujęte w cudzysłów znaki wraz ze znakiem zerowym dodawanym automatycznie przez kompilator są przechowywane w pamięci jako łańcuch znakowy. Cała fraza w cudzysłowie działa jak wskaźnik do miejsca, w którym zapisany jest łańcuch (literał łańcuchowy przypomina pod tym względem nazwę tablicy, która jest równocześnie adresem w pamięci). Literały łańcuchowe należą do klasy statycznej – są przechowywane przez cały okres działania programu, przy czym wielokrotne odwołanie się tego samego literału łańcuchowego nie powoduje utworzenia jego nowej kopii (nie ma takiej potrzeby, gdyż wartość literału z definicji nie może ulec zmianie).

17.2 Łańcuchy a znaki

Stała łańcuchowa `"x"` nie jest tym samym, co stała znakowa `'x'`! Literał `'x'` należy do typu podstawowego (**char**), natomiast `"x"` należy do typu pochodnego (to tablica elementów typu **char**), przy czym `"x"` tak naprawdę składa się z *dwóch* znaków – `'x'` oraz `'\0'` (tj. znaku zerowego).

17.3 Definiowanie zmiennych umożliwiających korzystanie z łańcuchów

Istnieje wiele sposobów na zdefiniowanie zmiennych umożliwiających korzystanie z łańcuchów, m.in. użycie tablicy typu `char` oraz użycie wskaźnika do `char`.

Definiując tablicę znakową możesz zainicjalizować ją z użyciem literału łańcuchowego:

```
char sl[] = "ABC";
```

Forma tablicowa (`s1[]`) tworzy tablicę 4 elementów (jeden element dla każdego znaku łańcucha plus jeden dla znaku zerowego) i przypisuje tym elementom wartości kolejnych znaków z literału łańcuchowego (łącznie ze znakiem zerowym). Powyższy wiersz jest zatem skróconą formą standardowej składni inicjalizacji tablicy:

```
char sl[] = {'A', 'B', 'C', '\0'};
```

Zwróć uwagę, że `s1` jest *kopią literału łańcuchowego* a zarazem *stałą adresową*, której nie można zmienić – oznaczałoby to bowiem zmianę miejsca (adresu), w którym przechowywana jest tablica.

Użycie formy wskaźnikowej (`*s2`):

```
char* s2 = "ABC";
```

powoduje zarezerwowanie miejsca wyłącznie na *zmienną wskaźnikową wskazującą na miejsca w pamięci, w którym przechowywany jest literał* – oznacza to, że wartość tej zmiennej może ulec zmianie.

17.4 Długie literały łańcuchowe

Zdarza się, że literały łańcuchowe nie mieszczą się w jednym wierszu – dotyczy to zwłaszcza łańcuchów sterujący w wywołaniach funkcji `printf()` i `scanf()`. Ponieważ możesz łączyć literały łańcuchowe – jeśli po jednej stałej łańcuchowej (zawartej w cudzysłowie) następuje druga, oddzielona od pierwszej tylko znakami niedrukowanymi, język C traktuje tę kombinację jako jeden łańcuch – możesz rozbić jeden długi literał łańcuchowy na kilka krótszych, z których każdy będzie umieszczony w osobnym wierszu.

W przypadku wyświetlania komunikatów z użyciem funkcji `printf()` możesz dodatkowo użyć kilku instrukcji `printf()`, z których jedynie ostatnia kończyć się będzie znakiem `'\n'`.

Obie powyższe możliwości zostały pokazane na listingu 17.1. Stosując którąkolwiek z powyższych pamiętaj, aby w ramach łańcuchów umieszczać wszystkie potrzebne odstępy: `"Ala" "ma" "kota."` jest równoważne `"Alamakota."`.

Listing 17.1. Wyświetlanie długich łańcuchów znaków

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

int main(void) {
    printf("Ala ma ");
    printf("kota.\n");

    printf("Ala ma "
           "kota.\n");  // wcięcie podkreśla, że to kontynuacja instrukcji

    return EXIT_SUCCESS;
}
```

Nie wolno dzielić na kilka wierszy łańcucha zawartego w cudzysłowie, przykładowo:

```
"Ala
ma kota."
```

gdyż wówczas kompilator zgłosi szereg błędów (m.in. zauważy brak zamykającego cudzysłowu w pierwszej linii, potraktuje pierwszy token w drugiej linii – `ma` – jako nazwę typów itp.).

17.5 Wyświetlanie łańcuchów

Do wyświetlania łańcuchów znaków z użyciem funkcji `printf()` służy specyfikator konwersji `%s`:

```
printf("Miasto: %s\n", "Krakow");
```

17.6 Własne funkcje łańcuchowe

Załóżmy, że chcesz własnoręcznie napisać funkcję wyświetlającą łańcuch znaków `s`, o działaniu takim jak wywołanie standardowej funkcji `printf("%s", s)`. Listing 17.2 przedstawia jedną z możliwości jej zrealizowania.

Listing 17.2. Własna funkcja wyświetlająca łańcuch znaków znak po znaku

```
#include <stdio.h>

void print_str(const char* s) {
    while (*s != '\0') {
        putchar(*s);  // putchar() z pliku stdio.h wyświetla pojedynczy znak
        ++s;
    }
}
```

Wskaźnik do `char` o nazwie `s` początkowo wskazuje na pierwszy element przekazanego do funkcji argumentu. Ponieważ funkcja nie powinna modyfikować łańcucha, w definicji parametru `s` użyliśmy modyfikatora `const`. Po wyświetleniu zawartości pierwszego elementu, wskaźnik ulega zwiększeniu i wskazuje na kolejny element. Proces ten jest powtarzany dopóty, dopóki wskaźnik nie będzie wskazywał na element zawierający znak zerowy.

Funkcję `print_str()` możesz traktować jako model tworzenia funkcji przetwarzających łańcuchy – ponieważ każdy łańcuch kończy się znakiem zerowym, funkcja nie musi znać rozmiaru łańcucha; zamiast tego może ona przetwarzać kolejne znaki aż do napotkania znaku zerowego.

Wielu programistów użyłoby w pętli `while` następującego wyrażenia testowego:

```
while (*s) ...
```

Ponieważ pętla wykonywana jest dopóki wyrażenie testowe ma wartość *niezerową*, w momencie gdy `s` wskazuje na znak zerowy pętla zostanie przerwana. To podejście to rodzaj idiomu programistycznego, czyli szeroko szeroko rozpowszechnionego sposobu na realizację pewnej czynności (czasem nawet kosztem czytelności).

Podobnie część programistów napisałoby ciało pętli w postaci jednej instrukcji:

```
putchar(*s++);
```

jednak, jak wspomniano w rozdziale *7.10 Skutki uboczne i punkty sekwencyjne*, lepiej unikać łączenia operatorów postinkrementacji i postdekrementacji w złożone wyrażenia.

Zwróć uwagę, że listing 17.2 deklaruje parametr jako `const char* s`, a nie `const char s[]`. Choć z technicznego punktu widzenia obie formy są identyczne (dają ten sam rezultat), argumentem wywołania funkcji `print_str()` może być nie tylko tablica zadeklarowana przez programistę, ale także stała łańcuchowa lub dowolna inna zmienna należącą do typu `char*`.

17.7 Standardowe funkcje łańcuchowe

Biblioteka standardowa zawiera szereg funkcji umożliwiających wygodną pracę z łańcuchami znaków – są one zawarte głównie w plikach nagłówkowych `string.h` oraz `ctype.h`.

17.7.1 Długość łańcucha: `strlen()`

Do badania długości łańcucha znaków – czyli liczby znaków faktycznie tworzących łańcuch, z pominięciem znaku zerowego – służy funkcja `strlen()`, zadeklarowana w pliku nagłówkowym `string.h`, o poniższym prototypie:

```
size_t strlen(const char* str);
```

Zwróć uwagę, że `sizeof` i `strlen()` zwracają różne wartości dla tablicy przechowującej łańcuch znaków, gdyż łańcuch może nie zajmować całej tablicy (uruchom poniższy program):

```
#define __USE_MINGW_ANSI_STDIO 1
```

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

int main ()
{
    char s1[10] = "ABC"; // łańcuch zajmuje 4 elementy tablicy
    char s2[] = "ABC";

    printf("s1: sizeof = %2zu , strlen = %2zu\n", sizeof(s1), strlen(s1));
    printf("s2: sizeof = %2zu , strlen = %2zu\n", sizeof(s2), strlen(s2));

    return EXIT_SUCCESS;
}
```

Wynik wykonania powyższego programu:

```
s1: sizeof =  10 , strlen =   3
s2: sizeof =   4 , strlen =   3
```

Rozdział 18

Struktury i inne formy danych

W tym rozdziale dowiesz się, czym są struktury, jak uzyskiwać dostęp do ich poszczególnych składników, oraz jak pisać funkcje przeznaczone do ich przetwarzania. W końcowej części rozdziału będziesz miał również okazję przyjrzeć się uniom, instrukcji `typedef`, oraz wskaźnikom do funkcji.

18.1 Struktury

Struktura (ang. structure) to w języku C zdefiniowany przez użytkownika złożony typ danych umożliwiający łączenie ze sobą danych różnych typów.

Oto przykład sytuacji, w której znajdują zastosowanie struktury:
Mamy za zadanie stworzyć system informatyczny do obsługi hurtowni. Kierownik hurtowni chciałby móc szybko zorientować się ile ma sztuk którego produktu oraz ile wynosi jego cena. Niektóre z tych danych (np. nazwa produktu) mogą być zapisane w tablicy łańcuchów, inne (np. liczba sztuk) jako wartość całkowita, a jeszcze inne (np. cena) jako wartość zmiennoprzecinkowa. Gdyby utworzyć osobną tablicę na każdą z tych kategorii danych, łatwo byłoby się w nich pogubić – lepszym rozwiązaniem byłoby skorzystanie tylko z jednej tablicy, za to takiej, w której każdy element zawiera wszystkie potrzebne informacje na temat jednego produktu. Formą danych umożliwiającą realizację takiego scenariusza jest właśnie struktura.

Oto zalążek programu realizującego wymaganą przez kierownika hurtowni funkcjonalność:

Listing 18.1. Program do obsługi hurtowni

```
/* hurtownia.c - program do obslugi hurtowni */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_NAME_LENGTH 41  // maksymalna dlugosc nazwy produktu + 1

struct Product { // szablon struktury o nazwie `Product`
    char name[MAX_NAME_LENGTH];
    int number;
    double price;
};

int main(void) {
    struct Product product; // declaracja `product` jako zmiennej
                            //   typu `Product`

    strncpy(product.name, "Egg", MAX_NAME_LENGTH);
    product.number = 12;
    product.price = 0.50;

    printf("%s (%d pcs): %.2f zl\n", product.name,
           product.number, product.price);

    return EXIT_SUCCESS;
}
```

Wynik wykonania programu:

```
Egg (12 pcs): 0.50 zl
```

Struktura utworzona w listingu 18.1 składa się z trzech części zwanych **składowymi** (ang. members) lub **polami** (ang. fields), przechowujących nazwę, liczebność i cenę produktu.

18.1.1 Definicja typu strukturalnego

Definicja typu strukturalnego opisuje budowę każdego obiektu struktury – nie tworzy ona rzeczywistego obiektu w pamięci, a jedynie określa jego składowe. Typowa postać takiej definicji wygląda następująco:

```
struct [identyfikator_typu_strukturalnego] {
  typ_pola_1 nazwa_pola_1;  // deklaracja 1-ego pola
  ...
  typ_pola_n nazwa_pola_n;  // deklaracja n-tego pola
} [nazwa_zmiennej_1, nazwa_zmiennej_2, ...];
```

Przykładowo:

```
struct Point2D {
    double x;
    double y;
};
```

Powyższa definicja opisuje typ strukturalny złożony z dwóch zmiennych typu zmiennoprzecinkowego – takiego typu strukturalnego można użyć do opisu punktu na płaszczyźnie.
Zwróć uwagę, że podanie identyfikatora typu strukturalnego jest opcjonalne (co zostanie później wykorzystane podczas definiowania aliasów). Opcjonalnie można na końcu definicji typu strukturalnego struktury podać zmienne, które mają zostać zdefiniowane z użyciem tego typu, przykładowo:

```
struct Point2D {
    double x;
    double y;
} p;  // od razu zdefiniuj zmienną `p`
```

Identyfikatora typu strukturalnego używa się następnie do tworzenia obiektów danego typu strukturalnego:

```
struct Point2D p;  // `p` to zmienna o budowie zgodnej z szablonem `Point2D`
```

Pamiętaj, że w przypadku zdefiniowania typu strukturalnego wewnątrz funkcji identyfikator tego typu jest dostępny tylko w obrębie tej funkcji!

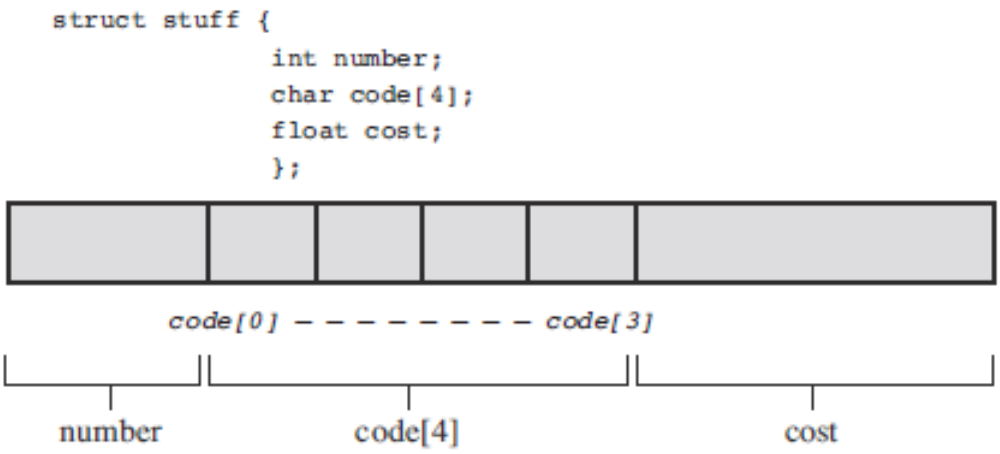
18.1.2 Definiowanie zmiennej strukturalnej

Słowo „struktura” jest używane w dwóch znaczeniach:

- jako *typ strukturalny* informujący kompilator o tym, w jaki sposób mają zostać przedstawione dane, oraz
- jako *zmienna strukturalna*, czyli obiekt typu strukturalnego.

W definicji zmiennej strukturalnej `struct S` pełni dokładnie tę samą rolę, co `int` lub `double` w „zwykłych” deklaracjach (gdyż definicja typu strukturalnego o identyfikatorze `S`, tworzy nowy typ o nazwie `struct S`).

Pola struktury są przechowywane w pamięci w sposób ciągły (zob. rys. 18.1).



Rysunek 18.1. Sposób przechowywania tablicy struktur w pamięci

18.1.3 Inicjalizacja struktury

Aby zainicjalizować strukturę korzystamy ze składni podobnej do tej stosowanej w przypadku tablic:

```
struct Product table = {
    "Table",
    2,
    120.30
};
```

czyli korzystamy z ujętej w klamry listy wartości rozdzielonych przecinkami. Każda wartość powinna należeć do tego samego typu, co odpowiadający jej składnik struktury. Dla zwiększenia czytelności warto zapisywać każdą wartość w osobnym wierszu, jednak z punktu widzenia kompilatora istotne są tylko przecinki.

Standard C99 dopuścił możliwość odwoływania się podczas inicjalizacji struktury do jej pól z użyciem zapisu `.<nazwa_pola>`, co dodatkowo poprawia czytelność instrukcji inicjalizacji, przykładowo:

```
struct Product product = {
        .name = "Egg",
        .number = 1,
        .price = 1.5,
};
```

18.1.4 Uzyskiwanie dostępu do składników struktury

Dostęp do składników struktury uzyskuje się za pomocą **operatora przynależności** (ang. member access operator), którego symbolem jest kropka (`.`), np. wyrażenie `table.number` oznacza „pole `number` struktury `table`”. Z wyrażenia `table.number` można korzystać dokładnie w taki sam sposób, jak z każdej innej zmiennej typu `double`.

Zwróć uwagę, że operator przynależności ma wyższy priorytet niż operator adresu `&`, zatem wyrażenie `&table.number` jest równoważne `&(table.number)`.

18.1.5 Struktury zagnieżdżone

Czasami zachodzi potrzeba utworzenia struktury zagnieżdżonej, czyli takiej, która zawiera w sobie inną strukturę. Przykładowo, dla każdego obiektu możesz chcieć przechowywać informację o jego położeniu na płaszczyźnie. Przyjmując, że obiekt jest identyfikowany za pomocą liczbowego identyfikatora, definicja typu strukturalnego reprezentującego obiekt będzie miała następującą postać:

```
struct Object {
    int id;
    struct Point2D location;
};
```

Zwróć uwagę, że zagnieżdżona struktura została umieszczona w definicji typu strukturalnego w identyczny sposób, jak składowa dowolnego innego typu (np. typu prostego).

Dostęp do składowej struktury zagnieżdżonej odbywa się przez ponowne użycie operatora przynależności:

```
struct Object obj;
obj.location.x = 3;
```

Wyrażenie `obj.location.x` jest interpretowane w kierunku od lewej do prawej:

```
(obj.location).x
```

18.1.6 Wskaźniki do struktur

Definicja wskaźnika do typu strukturalnego wygląda identycznie, co definicja wskaźnika do typu prostego, przykładowo:

```
struct Point2D* p;  // wskaźnik do struktury `Point2D`
```

Dostęp do składowych struktury wskazywanej przez `p` możesz uzyskać na dwa sposoby:

- z użyciem „zwykłego” operatora dereferencji `*`: `(*p).x` albo
- z użyciem operatora dereferencji struktury `->`: `p->x`

Operator dereferencji struktury składa się z dwóch znaków: dywizu (−) i następującego po nim symbolu „większy niż” (>).

Zauważ, że dodanie 1 do wskaźnika `p` jest równoznaczne z dodaniem 16 bajtów do przechowywanego w nim adresu, ponieważ każdy obiekt typu `Point2D` wymaga po 8 bajtów na przechowanie każdej ze składowych[1].

18.1.7 Przypisanie struktury strukturze

Jeśli `s1` i `s2` są strukturami tego samego typu `struct S`, wówczas dozwolone są poniższe instrukcje przypisania:

```
struct S s1;
struct S s2 = s1;  // inicjalizacja struktury inną strukturą
s2 = s1;  // przypisanie jednej struktury drugiej strukturze
```

W obu powyższych przypadkach przypisanie odbywa się w ten sposób, że każdemu polu struktury `s2` zostaje przypisana wartość analogicznego pola w strukturze `s1`.

18.1.8 Struktury a funkcje

Do funkcji można przekazać albo całą strukturę, albo jej adres, albo (gdy istotna jest tylko część struktury) pojedynczy składnik struktury; funkcja może zwracać obiekt danych typu strukturalnego (zob. listing 18.2).

Listing 18.2. Struktury a funkcje

```
/* geometria.c */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct Point2D {
    double x;
    double y;
};

double distance(struct Point2D p1, struct Point2D p2) {
    // Oblicz odleglosc miedzy dwoma punktami.
```

[1] Na niektórych komputerach rozmiar struktury może być większy niż suma rozmiarów jej składników. Powodem tego jest wyrównywanie adresów wszystkich pól na przykład do liczb parzystych albo do wielokrotności liczby 4. W takim przypadku struktury mogą zawierać w sobie nieużywane „dziury”.

```
    return sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));
}

void reflect_coordinate(double* coord) {
    *coord = -(*coord);
}

struct Point2D reflect_point(struct Point2D* p) {
    // Odbij punkt wzgledem poczatku ukladu wspolrzednych.
    reflect_coordinate(&p->x);
    reflect_coordinate(&p->y);
    return *p;
}

int main(void) {
    struct Point2D p1 = {1, -2};
    struct Point2D p2 = {0, 1};

    printf("d(p1, p2) = %.2f\n", distance(p1, p2));
    reflect_point(&p1);
    printf("P(x,y) = [%.2f, %.2f]\n", p1.x, p1.y);

    return EXIT_SUCCESS;
}
```

Ponieważ struktury mogą mieć duże rozmiary (rozmiar struktury zależy od liczby i typu jej składowych), w przypadku gdy bardzo istotna jest wydajność programu, warto rozważyć przekazywanie struktur do funkcji przez wskaźnik – nawet, jeśli przekazany argument nie będzie wewnątrz niej modyfikowany (przekazywanie przez adres wiąże się bowiem z przekazaniem jedynie adresu struktury, który jest zawsze pojedynczą liczbą całkowitą).

18.1.9 Tablice struktur

Definicja tablicy struktur ma taką samą formę, jak w przypadku każdej innej tablicy:

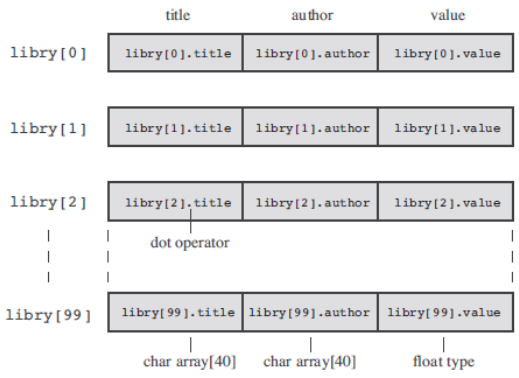
```
struct Product products[10];  // tablica 10 obiektów struktury
                              //   typu `struct Product`
```

Chcąc odwołać się do pola elementu w tablicy struktur stosuje się te same zasady, co w przypadku pojedynczej struktury:

```
products[1].name
```

Zauważ, że indeks tablicy znajduje się przed operatorem przynależności, a nie na końcu wyrażenia – przyrostek `.name` możemy bowiem dodać tylko do nazwy struktury; nazwą taką jest `products[1]`, ale nie `products` (nazwa `products` oznacza *wskaźnik* na pierwszy element struktury). Z kolei zapis `products[1].name[2]` oznaczać będzie trzeci znak w nazwie drugiego produktu.

Rysunek 18.2 przedstawia sposób przechowywania tablicy struktur w pamięci.



Rysunek 18.2. Sposób przechowywania tablicy struktur w pamięci

18.2 Typy wyliczeniowe

Typ wyliczeniowy (ang. enumerated type) to typ, którego zbiór wartości jest jawnie określony za pomocą **stałych wyliczeniowych** (ang. enumeration constants). Typ wyliczeniowy służy m.in. do reprezentowania cech jakościowych nominalnych (np. płeć, kolor oczu itp.); użycie typów wyliczeniowych poprawia czytelność i bezpieczeństwo programów.

Składnia definicji typu wyliczeniowego jest podobna, jak w przypadku typu strukturalnego:

```
enum [identyfikator_typu_wyliczeniowego] { lista_stałych_wyliczeniowych }
```

gdzie `lista_stałych_wyliczeniowych` to lista elementów rozdzielonych przecinkiem, z których każdy ma formę `stała_wyliczeniowa [= stałe_wyrażenie]` (wartość stałego wyrażenia użytego do inicjalizacji stałej wyliczeniowej musi dać się reprezentować jako wartość typu `int`). Zarówno podanie identyfikatora typu wyliczeniowego, jak i dokonanie inicjalizacji którejkolwiek ze stałych wyliczeniowych jest opcjonalne. W rzeczywistości wszystkie stałe wyliczeniowe są typu `int` (zatem można je stosować wszędzie tam, gdzie dozwolone jest użycie wartości typu `int`). Stałe wyliczeniowe są inicjalizowane w kolejności wstąpienia na liście, przy czym stała zainicjalizowana niejawnie otrzymuje albo wartość o 1 większą od ostatniej uprzednio zdefiniowanej stałej, albo 0 w przypadku pierwszej stałej na liście.

Oto przykłady definicji typów wyliczeniowych:

```
enum Foo { A, B, C=10, D, E=1, F, G=F+C };
// A=0, B=1, C=10, D=11, E=1, F=2, G=12

enum Weekday { MO, TU, WE, TH, FR, SA, SU};
```

Z obiektów typu wyliczeniowego korzysta się jak z obiektów typów podstawowych, przykładowo:

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

enum {
    MO, TU, WE, TH, FR, SA, SU
} Weekday;

int main(void) {

    enum Weekday d = TU;

    printf("Jest ");
    switch (d) {
        case MO:
            printf("poniedzialek.\n");
            break;
        case TU:
            printf("wtorek.\n");
            break;
        default:
            printf("inny dzien tygodnia niz poniedzialek i wtorek.\n");
            break;
    }

    return EXIT_SUCCESS;
}
```

Wynik działania programu:

```
Jest wtorek.
```

Chociaż *stałe* wyliczeniowe należą zawsze do typu `int`, język C nie precyzuje ściśle typu *zmiennych* wyliczeniowych – stwierdza jedynie, że musi to być typ kompatybilny z typem `char` lub typem całkowitym (ze znakiem lub bez znaku) oraz że typ ten musi umożliwiać reprezentację wartości wszystkich stałych

wyliczeniowych należących do zbioru wartości tego typu wyliczeniowego. W powyższym przykładzie kompilator mógłby przechowywać zmienną `d` za pomocą typu **`char`**, o ile typ **`char`** miałby szerokość co najmniej 3 bitów niezbędnych do reprezentacji siedmiu unikalnych wartości.

18.3 Unie

Unia (ang. union) jest typem, który pozwala przechowywać dane różnego typu w tym samym obszarze pamięci (jednak nie równocześnie). To szczególnie przydatne w przypadku programowania systemów wbudowanych albo gdy potrzebny jest bezpośredni dostęp do warstwy sprzętowej lub pamięci.

Definiowanie unii przebiega analogiczne, co w przypadku struktur (zamiast słowa kluczowego **`struct`** należy użyć słowa kluczowego **`union`**); podobnie dostęp do składowych unii uzyskuje się z użyciem tych samych operatorów, co w przypadku struktur (czyli `.` i `->`).

Oto przykład unii wyrażającej 2-bajtowy rejestr, umożliwiającej odwoływanie się zarówno do wartości całego rejestru (jako tzw. słowa maszynowego), jak i do poszczególnych jego bajtów[2]:

```
union Register {
    struct {
        unsigned char byte1;
        unsigned char byte2;
    } bytes;
    unsigned short word;
};
union Register reg;
```

Dostęp do wartości tak zdefiniowanego obiektu `reg` wygląda następująco:

```
reg.word = 0x1234;     // ustaw wartość obu bajtów
reg.bytes.byte1 = 4;   // ustaw wartość pierwszego bajtu
```

Tworząc pojedynczą zmienną typu unii kompilator przydziela jej tyle miejsca, aby mogła ona przechować największą ze zmiennych będących częścią szablonu unii – w przypadku unii `Register` oba składniki mają rozmiar 2 bajtów (na naszym komputerze).

Unie można również wykorzystać do efektywnej implementacji w języku C kolejki komunikatów, w przypadku gdy różne typy komunikatów wymagają przechowywania danych różnego typu – komunikat można zdefiniować jako typ strukturalny zawierającą dwa pola: jedno przechowujące typ komunikatu oraz drugie będące unią wszystkich danych możliwych do skojarzenia z komunikatem:

```
enum MessageType {
    INT, FLOAT, DOUBLE
};
struct Message {
    enum MessageType type;
    union {
        int val_int;
        float val_float;
        double val_double;
    };
};

void handle_message(struct Message* msg) {
    switch (msg->type) {
        case INT:  // zrób coś z msg->val_int
            break;

        case FLOAT:  // zrób coś z msg->val_float
            break;

        case DOUBLE:  // zrób coś z msg->val_double
            break;
```

[2] Oczywiście w zależności od architektury procesora należałoby również wziąć pod uwagę m.in. kolejność bajtów (zob. rozdz. *4.1 Czym są obiekty?*).

```
    }
}
```

Powyższa implementacja sprawia, że rozmiar struktury `Message` wynosi jedynie 12 bajtów (miejsce zajęte przez pole typu `MessageType` i unię o rozmiarze **`sizeof`**`(`**`double`**`)`) zamiast 20 bajtów (gdyby struktura `Message` zamiast pola-unii zawierała wszystkie pola tej unii).

18.4 Deklarowanie i definiowanie typów a błędy budowania programu

Ponieważ kompilator analizuje kod jednostki translacji tylko raz (w kolejności od pierwszego do ostatniego wiersza), każdy typ danych musi zostać zadeklarowany *i zdefiniowany* przed jego pierwszym użyciem – próba zbudowania poniższego programu:

```
1  #include <stdlib.h>
2
3  struct S1;      // deklaracja typu `struct S1`
4  struct S1 {};   // definicja typu `struct S1`
5  struct S1 {};   // BŁĄD: redefinicja typu `struct S1`
6
7  struct S2;  // deklaracja typu `struct S2`
8
9  int main(void) {
10     struct S2 s;
11     return EXIT_SUCCESS;
12 }
13
14 struct S2 {  // BŁĄD: definicja typu `struct S2` dopiero po jego użyciu
15     int x;
16 };
```

spowoduje pojawienie się dwóch błędów (na etapie *kompilacji*):

- Błąd „`redefinition of 'struct S1'`” wynika z faktu, że (jak już wspomniano przy okazji omawiania zmiennych i funkcji), w danej jednostce translacji dany typ nie może zostać zdefiniowany kilkukrotnie.
- Błąd „`storage size of 's' isn't known`” wynika z faktu, że w powyższym programie definicja typu **`struct`** `S2` pojawia się dopiero po jego użyciu – zatem w momencie próby *zdefiniowania* zmiennej `s` (w wierszu 10) kompilator nie jest w stanie określić rozmiaru tej zmiennej.

18.5 Specyfikator `typedef`

Specyfikator **`typedef`** pozwala definiować *aliasy* dla typów już istniejących. Przykładowo, poniższa instrukcja:

```
typedef unsigned char byte;
```

spowoduje zdefiniowanie nowego aliasu `byte` (równoważnego typowi **`unsigned char`**), z którego można później korzystać m.in. przy deklarowaniu zmiennych:

```
byte b;  // równoważnie: unsigned char b;
```

Zasięg definicji typu zależy od położenia instrukcji **`typedef`**. Jeśli definicja znajduje się wewnątrz funkcji, jej zasięg jest lokalny; jeśli znajduje się ona poza funkcją, jej zasięg jest globalny. Zasady nazewnictwa typów są takie same, jak zasady nazewnictwa zmiennych.

W powyższym przykładzie utworzenie aliasu dla istniejącego typu pozwala zwiększyć czytelność programu – użycie nazwy `byte` zamiast **`unsigned char`** pozwala podkreślić, że zmienne typu `byte` będą wykorzystywane do przechowywania liczb, a nie kodów znaków.

Instrukcji **`typedef`** można stosować do definiowania aliasów na dowolnie złożone typy, np.

```
typedef unsigned char pesel_t[11];  // alias na 11-elementową tablicę
pesel_t p = {4, 4, 0, 5, 1, 4, 0, 1, 4, 5, 8};
```

W szczególności specyfikator **`typedef`** przydaje się w połączeniu z definicjami typów strukturalny, wyliczeniowych i unii – dzięki niemu nie trzeba później w programie powtarzać słów kluczowych **`struct`**,

`enum` i **`union`** odwołując się odpowiednio do nazwy typu strukturalnego, typu wyliczeniowego lub do nazwy typu unii:

```
typedef struct {
    double x;
    double y;
} Point2D;

Point2D p; // `struct` nie jest teraz potrzebne
```

Zwróć uwagę, że w ramach powyższej instrukcji **`typedef`** w istocie definiujemy *anonimowy* typ strukturalny i nadajemy mu pewien alias.

18.5.1 Przenośność oprogramowania

Definiowanie aliasów pozwala zwiększyć przenośność programów. Przykładowo, w odniesieniu do operatora **`sizeof`** oraz funkcji `time()` standard języka C określa jedynie, że zwracają one wartość całkowitą – określenie dokładnego jej typu odbywa się na poziomie konkretnej implementacji języka C[3]. Wprowadzenie aliasów takich jak `size_t` i `time_t` (które są zdefiniowane w standardowych plikach nagłówkowych i którym każda implementacja może przyporządkować jakiś określony typ za pomocą instrukcji **`typedef`**) umożliwiło ustalenie ogólnych prototypów funkcji, na przykład:

```
time_t time(time_t*);
```

Na jednym systemie `time_t` może oznaczać typ **`unsigned int`**, na innym – **`unsigned long`**, jednak mimo to program zachowuje spójność.

18.5.2 Alias `size_t`

Standard języka C określa, że alias `size_t` odnosi się do typu całkowitego bez znaku będącego rezultatem działania pewnych operatorów, m.in. **`sizeof`**. Standard nie precyzuje, jaki konkretnie typ powinien zostać użyty (np. **`unsigned int`** czy **`unsigned long`**), jednak wymaga, aby był on w stanie przechowywać maksymalną liczbę bajtów zajmowaną przez teoretycznie możliwy do utworzenia obiekt – czyli np. tablicę elementów typu **`char`** zajmującą całą dostępną pamięć operacyjną programu.

W związku z tym z typu tego korzysta się powszechnie m.in. do wyrażania indeksów tablic oraz zmiennej użytej jako iterator w pętli. Użycie „zwykłego” typu bez znaku może powodować błędy, np. program stosujący indeksowanie tablicy z użyciem zmiennej typu **`unsigned int`** (zamiast `size_t`) może działać błędnie na systemie 64-bitowym, jeśli liczba elementów tablicy przekracza wartość `UINT_MAX` – stanie się tak w przypadku, gdy kompilator użyty do zbudowania programu reprezentował typ **`int`** z użyciem 4 bajtów (czyli 32 bitów).

18.5.3 Typy o stałej szerokości, typy o minimalnej szerokości...

W przypadku programowania systemowego szczególnie istotne są aliasy dla tzw. **typów całkowitych o stałej szerokości** (ang. fixed width integer types) – w przypadku niektórych typów całkowitych standard języka C nie definiuje ich rozmiaru (określa jedynie, że np. „typ **`int`** ma umożliwiać reprezentację co najmniej liczby z zakresu $[-32767, +32767]$”, czyli typ ten musi mieć rozmiar co najmniej dwóch bajtów), przez co niektóre operacje zależne od rozmiaru danych mogłyby przebiegać w różny sposób na różnych platformach sprzętowych. Aliasy dla typów całkowitych o stałej szerokości są zdefiniowane w standardowym pliku nagłówkowym `stdint.h` (dostępnym począwszy od standardu C99), przykładowo `uint8_t` to alias na typ całkowity bez znaku o szerokości dokładnie 8 bitów.

Również przydatne mogą być aliasy gwarantujące:

- minimalny rozmiar typu (ang. minimum width types), np. `int_least8_t` oznacza typ całkowity ze znakiem o szerokości co najmniej 8 bitów
- najszybsze obliczenia (ang. fastest minimum width types), np. `int_fast8_t` oznacza najszybszy typ całkowity ze znakiem o szerokości co najmniej 8 bitów

[3] Ta ogólnikowość wypływa z przekonania komitetu standaryzacyjnego języka C, że żaden narzucony z góry typ nie będzie najlepszy dla wszystkich platform.

- największy rozmiar typu (dostępny w danej implementacji języka C), np. `intmax_t` oznacza największy dostępny typ całkowity ze znakiem

Aby wyświetlać wartości typów zdefiniowanych w bibliotece `inttypes.h` należy skorzystać ze specjalnych makr zdefiniowanych w tej bibliotece, np. `PRId32` służy do wyświetlania 32-bitowej wartości całkowitej ze znakiem:

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

int main(void) {
    int32_t me32 = 45933945;
    printf("me32 = %" PRId32 "\n", me32);
    return EXIT_SUCCESS;
}
```

18.5.4 Instrukcja `typedef` a dyrektywa `#define`

Instrukcja **`typedef`** przypomina dyrektywę **`#define`** z trzema istotnymi różnicami:

- W odróżnieniu od dyrektywy **`#define`**, instrukcja **`typedef`** nadaje się do definiowania aliasów wyłącznie na nazwy typów, a nie także aliasów na wartości.
- Instrukcja **`typedef`** jest przetwarzana przez kompilator, a nie przez preprocesor.
- W swoim obszarze zastosowań instrukcja **`typedef`** jest bardziej elastyczna niż dyrektywa **`#define`**.

18.6 Złożone deklaracje typów

Język C pozwala definiować bardzo skomplikowane typy danych – znaczenie definicji wykorzystującej „podstawowe” typy może zostać zmienione przez dodanie co najmniej jednego z modyfikatorów (`id` oznacza identyfikator tego, co definiujemy – zmiennej lub funkcji):

- `* id` – wskaźnik
- `id()` – funkcja
- `id[]` – tablica

przy czym możliwe jest użycie wielu modyfikatorów w jednej deklaracji, co pozwala tworzyć duży zakres różnych typów:

```
int arr2d[8][8];  // tablica tablic typu int
int** ptr;        // wskaźnik do wskaźnika do int
int* arr[2];      // 2-elementowa tablica wskaźników do int
int (*ptr)[5];    // wskaźnik do 5-elementowej tablicy typu int
int* arr2d[3][4]; // tablica 3 x 4 wskaźników do int
int (*ptr)[3][4]; // wskaźnik do tablicy 3 x 4 wartości int
int (*arr[3])[4]; // 3-elementowa tablica wskaźników do 4-elementowych
                  //   tablic typu int
```

Aby rozszyfrować powyższe deklaracje należy pamiętać o następujących zasadach:

- Modyfikatory `()` i `[]` mają ten sam priorytet, wyższy niż priorytet modyfikatora `*`.
  Przykład: **`int`**`* arr[2];` deklaruje tablicę wskaźników, a nie wskaźnik do tablicy.
- Modyfikatory `()` i `[]` działają w kierunku od lewej do prawej.
  Przykład: **`int`** `arr2d[2][3];` deklaruje 2-elementową tablicę tablic 3-elementowych, a nie 3-elementową tablicę tablic 2-elementowych.
- Modyfikator `[]` ma pierwszeństwo przed `*`, ale *operator* nawiasów ma zawsze najwyższy priorytet.
  Przykład: **`int`** `(*ptr)[2];` deklaruje wskaźnik do tablicy.

Przykładowo, proces rozszyfrowywania poniższej deklaracji:

```
int* arr2d[3][4];
```

wygląda następująco:

- Modyfikator `[3]` ma pierwszeństwo przed `*`, a także (z powodu kierunku wiązania) przed `[4]`, zatem `arr2d` jest tablicą trzech elementów.

- Następny w kolejności jest modyfikator `[4]`, zatem elementy tablicy są tablicami złożonymi z czterech elementów. Modyfikator `*` stwierdza, że elementy te są wskaźnikami.
- Ostatnią częścią układanki jest słowo kluczowe **`int`**: `arr2d` jest trójelementową tablicą czteroelementowych tablic wskaźników do **`int`**. Zajmuje ona tyle samo pamięci, co 12 wskaźników.

Przeanalizujmy teraz następującą deklarację:

```
int (*ptr)[3][4];
```

Nawiasy sprawiają, że modyfikator `*` działa jako pierwszy. Tym samym zmienna `ptr` zostaje zadeklarowana jako wskaźnik do tablicy 3×4 wartości typu **`int`**. Przydzielony obszar pamięci ma długość jednego wskaźnika.

18.7 Wskaźniki do funkcji

Wskaźniki do funkcji przydają się m.in. w sytuacji, kiedy chcemy sparametryzować działanie jednej funkcji inną funkcją[4].

Przykładowo, możesz chcieć zdefiniować funkcję, która będzie wykonywała pewną (modyfikującą) operację dla każdego elementu tablicy liczb całkowitych przekazanej jako argument tej funkcji – schemat postępowania w przypadku dowolnej tablicy i dowolnej funkcji będzie identyczny. Aby Twoje rozwiązanie było uniwersalne, możesz oprócz wskaźnika do tablicy i jej rozmiaru przekazywać także wskaźnik do funkcji określający operację do wykonania na elementach:

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

// Ostatni parametr map() to wskaźnik do funkcji przyjmującej
//   wskaźnik do int i nie zwracającej żadnej wartości.
void map(int* arr, const int n, void(* f)(int*)) {
    for (int i = 0; i < n; ++i) {
        f(arr + i);
    }
}

void tripple(int* x) {
    *x *= 3;
}

int main(void) {
    int arr[3] = {1, 2, 3};
    const int n = sizeof(arr) / sizeof(int);
    map(arr, n, tripple);

    for (int i = 0; i < n; ++i) {
        printf("%d ", arr[i]);
    }

    return EXIT_SUCCESS;
}
```

Powyższy przykład ilustruje mechanizm **wywołania zwrotnego** (ang. callback) – sytuacji, w której użytkownik jedynie rejestruje funkcję do późniejszego wywołania, natomiast funkcja rejestrująca wywołają ją w stosownym dla siebie czasie (później, kiedy następuje wywołanie zwrotne, funkcja je wykonująca nie wie nic o tym, co się zdarzy; to zależy od tego, co zostało zarejestrowane do wywołania).

Co przechowuje wskaźnik do funkcji? Wskaźnik do obiektu danych (np. zmiennej typu **`int`**) przechowuje adres w pamięci, pod którym rozpoczyna się wskazywany obiekt – podobnie wskaźnik do funkcji przechowuje adres, pod którym rozpoczyna się kod maszynowy funkcji.

[4] zob. np. standardową funkcję sortującą dane `qsort()`

18.7.1 Definicja wskaźnika do funkcji

Definiując wskaźnik do danych, należy określić typ wskazywanej wartości. Podobnie definiując wskaźnik do funkcji, należy określić typ wskazywanej funkcji – typy jej argumentów oraz typ wartości zwracanej.

Aby zdefiniować wskaźnik do funkcji określonego typu wystarczy wzorować się na jej deklaracji – należy zastąpić nazwę danej funkcji wyrażeniem postaci `(* nazwa_wskaznika)`, wówczas `nazwa_wskaznika` stanie się wskaźnikiem do funkcji. Przykładowo:

```
void tripple(int* x);

void(* f)(int*);  // `f` to wskaźnik na funkcję przyjmującą wskaźnik
                  //   na int i nie zwracającą wartości (może wskazywać
                  //   np. na funkcję `tripple`)
```

Pierwsza para nawiasów (wokół „`* f`”) jest wymagana z uwagi na kolejność działania modyfikatorów:

```
void *f(int*);
```

oznacza, że `f` jest funkcją, która zwraca wskaźnik.

Ponieważ wskaźnik do funkcji pełni jedynie rolę uchwytu (czyli nie musi znać implementacji wskazywanej funkcji), nie ma potrzeby podawania nazw parametrów wskazywanej funkcji – wystarczy podać typ tych parametrów (analogicznie jak w przypadku deklaracji funkcji[5]):

```
void(* f)(int*);     // OK: zalecana postać
void(* f)(int* x);   // ŹLE: postać ze zbędną nazwą parametru
```

18.7.2 Korzystanie ze wskaźnika do funkcji

Nazwa funkcji jest jednocześnie wskaźnikiem do tej funkcji, zatem zarówno nazwy funkcji, jak i nazwy wskaźnika do tej funkcji można stosować zamiennie. W związku z tym dysponując wskaźnikiem do funkcji można ją wywołać na dwa równoważne sposoby:

```
void(* f)(int*) = tripple;
int x = 1;
f(&x);
(*f)(&x);  // (*f) oznacza funkcję `tripple`
```

18.7.3 Wskaźniki do funkcji a złożone typy danych

Oto przykłady złożonych typów danych wykorzystujących wskaźniki do funkcji:

```
char* f();         // funkcja zwracająca wskaźnik do char
char (* f)();      // wskaźnik do funkcji, która zwraca wartość typu char
char (* f[2])();   // tablica 2 wskaźników do funkcji zwracających
                   //   wartość typu char
```

Dla poprawy czytelności kodu warto wprowadzać stosowne aliasy na wskaźniki do funkcji, przykładowo:

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

typedef int (* Comparator)(int, int);  // wskaźnik do funkcji służącej do
                                       //   porównywania dwóch wartości
                                       //   całkowitych

int greater_than(int a, int b) {
    return (a >= b) ? 1 : 0;
}

int main(void) {
```

[5] zob. rozdz. 10.2.1 Deklaracja (prototyp)

```
    Comparator c = greater_than;
    printf("%d\n", c(4, 1));

    return EXIT_SUCCESS;
}
```

Rozdział 19

Manipulowanie bitami

W tym rozdziale poznasz dwa mechanizmy języka C pozwalające wpływać na poszczególne bity w wartościach przechowywanych w pamięci komputera: operatory bitowe oraz pola bitowe.

19.1 Operatory bitowe

Przy omawianiu operatorów bitowych wartości będą zapisywane w systemie binarnym aby lepiej uwidocznić zmiany zachodzące w poszczególnych bitach. Natomiast ponieważ standard języka C nie wspiera literałów binarnych (choć niektóre kompilatory, np. GCC, zwierają takie niestandardowe rozszerzenie), w kodzie programu musisz korzystać z literałów całkowitych, ósemkowych lub szesnastkowych[1].

19.1.1 Bitowe operatory logiczne

Wszystkie cztery bitowe operatory logiczne (zebrane w tabeli 19.1) przetwarzają dane należące do typów całkowitych, włącznie z typem `char`. Nazywamy je **operatorami bitowymi** (ang. bitwise operators), ponieważ działają one na każdy bit danej wartości niezależnie od bitów sąsiednich. Operatory te mają znaczenie analogiczne do operatorów znanych z algebry Boole'a. Nie należy mylić *bitowych* operatorów logicznych (`~`, `&` i `|`) ze *zwykłymi* operatorami logicznymi (`!`, `&&` i `||`) operującymi na *całych* wartościach. Operatory logiczne nie zmieniają wartości żadnego z operandów.

Tablica 19.1. Bitowe operatory logiczne

Znaczenie	Operator	Przykład
bitowa negacja	`~x`	`~(1010) == (0101)`
bitowa koniunkcja (AND)	`a & b`	`(0011) & (0101) == (0001)`
bitowa alternatywa (OR)	`a | b`	`(0011) | (0101) == (0111)`
bitowa alternatywa wyłączająca (XOR)	`a ^ b`	`(0011) ^ (0101) == (0110)`

Język C udostępnia również wersje bitowych operatorów logicznych połączonych z przypisaniem (analogicznie do arytmetycznych operatorów przypisania), np. `&=`, `|=` itd.

19.1.2 Maski

Maska (ang. mask) jest po prostu układem bitów, w którym niektóre bity są włączone (1), a niektóre wyłączone (0). Maski można używać w połączeniu z operatorami bitowymi do uzyskiwania wartości, włączania, wyłączenia lub odwrócenia określonych bitów przy jednoczesnym pozostawieniu pozostałych bitów bez zmian[2].

Oto przykład korzystania z masek:

```
const unsigned char MASK = 0x01;  // 00000001
```

[1] zob. rozdz. *3.7.1 Podstawowy typ całkowity: `int`*

[2] Taka operacja może być przydatna np. gdy włączenie pewnego urządzenia zewnętrznego podłączonego do komputera wymaga włączenia jednego konkretnego bitu i pozostawienia bez zmian pozostałych.

```
unsigned char val = 0xFF;
val & MASK;     // uzyskanie wartości
val |= MASK;    // włączenie bitu
val &= ~MASK;   // wyłączenie bitu
val ^= MASK;    // odwrócenie bitu
```

Proces uzyskiwania wartości z użyciem maki nazywamy **stosowaniem maski**[3].

Zwróć uwagę, że chcąc sprawdzić wartość określonego bitu nie wystarczy porównać tej zmiennej z maską:

```
if (val == MASK) { /* ... */ }
```

Nawet, jeśli bit nr 1 w zmiennej `val` jest równy 1, wyrażenie `val == MASK` może być fałszywe z powodu różnic w innych bitach. Należy więc najpierw zamaskować pozostałe bity w zmiennej `val`, tak aby porównywany był tylko bit nr 1:

```
if ((val & MASK) == MASK) { /* ... */ }
```

Operatory bitowe mają niższy priorytet niż operator porównywania ==, stąd konieczność ujęcia wyrażenia `val & MASK` w nawias.

19.1.3 Bitowe operatory przesunięcia

Język C udostępnia dwa **operatory przesunięcia** (ang. shift operators), które pozwalają przesuwać bity w lewo lub w prawo.

Operator **przesunięcia w lewo** << (ang. left shift) przesuwa bity pierwszego operandu w lewo o liczbę miejsc określoną przez drugi operand. Zwolnione miejsca są wypełniane zerami, a bity wykraczające poza lewą granicę pierwszego operandu są usuwane, przykładowo:

`(10001010) << 2 == (00101000)`

Operator **przesunięcia w prawo** >> (ang. right shift) przesuwa bity pierwszego operandu wprawo o liczbę miejsc określoną przez drugi operand. Bity wykraczające poza prawą granicę pierwszego operandu są usuwane. W przypadku wartości bez znaku zwalniane miejsca są wypełniane zerami. W przypadku wartości ze znakiem efekt zależy od komputera – zwalniane miejsca mogą być wypełniane zerami lub kopiami bitu przechowującego znak (bitu najbardziej znaczącego). Przykładowo:

```
(10001010) >> 2 == (00100010)  // wartość ze znakiem, pierwsza możliwość
(10001010) >> 2 == (11100010)  // wartość ze znakiem, druga możliwość
(10001010) >> 2 == (00100010)  // wartość bez znaku, wszystkie systemy
```

Bitowe operatory przesunięcia nie zmieniają wartości żadnego z operandów.

Język C udostępnia również wersje bitowych operatorów przesunięcia połączonych z przypisaniem (analogicznie do arytmetycznych operatorów przypisania): <<= i >>=.

Operatory przesunięcia udostępniają szybki i wydajny (w zależności od sprzętu) sposób mnożenia i dzielenia przez potęgi dwójki:

- `val << n` mnoży liczbę przez 2^n
- `val >> n` dzieli liczbę przez 2^n (jeśli liczba nie jest ujemna)

Operacje przesunięcia są analogiczne do przesuwania przecinka przy mnożeniu lub dzieleniu w systemie dziesiętnym.

Operatory przesunięcia mogą również służyć do definiowania masek, przykładowo:

```
const unsigned char MASK = (1 << 3);
```

Operatory przesunięcia mogą również służyć do wydobywania grup bitów z większych jednostek: Załóżmy na przykład, że mamy wartość typu **`unsigned long`** wyrażającą kolor w przestrzeni barw RGB, w której bajty (począwszy od najmniej znaczącego) przechowują kolejno intensywność składowej czerwonej, zielonej i niebieskiej. Powiedzmy, że chcemy przechować intensywność każdej barwy składowej w oddzielnej zmiennej typu **`unsigned char`**. Aby to uczynić, moglibyśmy użyć następującego kodu:

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

[3] Można wyobrażać sobie, że maska to nieprzezroczysta płytka z otworami wyciętymi w miejscu włączonych bitów. Wyrażenie `val & MASK` przypomina przykrycie układu bitów taką płytką – widoczne są tylko te bity, które znajdują się pod „otworami" maski

```
typedef unsigned char byte;

int main(void) {
    const byte BYTE_MAX = 0xFF;

    unsigned long color = 0x002A162F;
    byte red = (byte) (color & BYTE_MAX);
    byte green = (byte) ((color >> 8) & BYTE_MAX);
    byte blue = (byte) ((color >> 16) & BYTE_MAX);

    printf("RGB = %#lx = [%#x, %#x, %#x]\n", color, red, green, blue);

    return EXIT_SUCCESS;
}
```

Wynik uruchomienia powyższego programu:

```
RGB = 0x2a162f = [0x2f, 0x16, 0x2a]
```

Użycie bitowej koniunkcji z wartością `0xFF` (czyli 11111111_2) powoduje „przycięcie" do jednego bajta – pozostałe bity wartości wyrażenia są zerowane.

19.2 Pola bitowe

Druga metoda manipulowania pojedynczymi bitami polega na skorzystaniu z **pola bitowego** (ang. bit field), które jest po prostu zbiorem sąsiadujących ze sobą bitów w ramach jednego słowa maszynowego (zwykle: jednej wartości typu `unsigned int`). Pola bitowe tworzymy definiując odpowiedni typ strukturalny – pola bitowe to takie pola struktury, dla których określona została ich szerokość (w bitach). Przykładowo, poniższa definicja tworzy zmienną strukturalną o nazwie `q` zawierającą dwa pola o rozmiarze jednego bitu i jedno pole o rozmiarze dwóch bitów:

```
struct {
  unsigned int a : 1;
  unsigned int b : 1;
  unsigned int c : 2;
} q;
```

Liczba bitów danego pola określa zakres wartości, jakie mogą być do niego przypisywane – przypisywana wartość nie może przekroczyć pojemności pola (np. w powyższym przykładzie: $2^1 - 1 = 1$ dla pól `a` i `b`, $2^2 - 1 = 3$ dla pola `c`).

Co się dzieje, jeśli całkowita liczba bitów przydzielonych polom przekracza szerokość słowa maszynowego? Wówczas rezerwowana jest kolejna jednostka o szerokości słowa maszynowego. Żadne z pól nie może znajdować się na granicy pomiędzy dwoma jednostkami, dlatego kompilator automatycznie rozmieszcza pola tak, aby taka sytuacja nie miała miejsca, w razie potrzeby pozostawiając między nimi puste obszary. Aby celowo umieścić taki pusty obszar, wystarczy zdefiniować samą szerokość pola, nie podając jego nazwy. Użycie szerokości pola równej 0 sprawia, że następne pole zostaje wyrównane do początku kolejnej jednostki:

```
#define __USE_MINGW_ANSI_STDIO 1

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

int main(void) {

    struct {
        unsigned int f1 : 1;
        unsigned int f2 : 1;
    } q1;

    struct {
```

```
        unsigned int f1 : 1;
        unsigned int : 32;  // pusty obszar o szerokości 4 bajtów
        unsigned int f2 : 1;
    } q2;

    printf("sizeof(q1) = %zu , sizeof(q2) = %zu\n", sizeof(q1), sizeof(q2));

    return EXIT_SUCCESS;
}
```

Uruchomienie powyższego programu na komputerze o 32-bitowym słowie maszynowym da jako wynik:

```
sizeof(q1) = 4 , sizeof(q2) = 12
```

Jak widzisz, struktura `q2` zajmuje więcej miejsca w pamięci, choć efektywnie przechowuje tyle samo danych, co struktura `q1`.

Ponieważ rozmieszczenie poszczególnych pól w ramach jednostki pamięci oraz położenie granic między polami zależy od architektury komputera, pola bitowe nie są przenośne. Zwykle jednak nie stanowi to problemu, gdyż są one wykorzystywane do celów, które same w sobie nie są przenośne (np. przesyłanie danych dokładnie w postaci wymaganej przez określone urządzenie sprzętowe).

19.2.1 Pola bitowe a kompresja danych

Zdarza się, że pola bitowe są wykorzystywane jako bardziej oszczędny sposób przechowywania danych. Załóżmy na przykład, że tworzymy aplikację do rysowania kształtów i chcemy przechować informację o stylu linii, na który składają się następujące parametry: grubość (1–7 pikseli), wzór (linia ciągła, przerywana, albo kropkowana) oraz kolor (czarny, czerwony, zielony, żółty, niebieski, fioletowy, niebieskozielony albo biały). Moglibyśmy wprawdzie wyrazić każdą cechę za pomocą osobnej zmiennej lub pełnowymiarowego pola struktury, jednak np. w przypadku systemów wbudowanych (o niewielkiej pamięci operacyjnej) takie marnotrawstwo bitów mogłoby być niedopuszczalne.

Przykładowo, w naszym problemie do przechowywania kompletu informacji wystarczy 8 bitów:

- grubości (wartości od 1 do 7, czyli siedem różnych wartości) można przechowywać w postaci z użyciem 3 bitów
- wzór (trzy różne wartości) można przechowywać z użyciem 2 bitów np. za pomocą kodowania „0 – ciągła, 1 – przerywana, 2 – kropkowana”
- kolor (osiem różnych wartości) można przechowywać z użyciem 3 bitów

Jeśli chodzi o kolory, możemy skorzystać z prostej reprezentacji RGB (ang. red-green-blue). Polega ona na wyrażeniu kolorów za pomocą trzech barw podstawowych: czerwonej, zielonej i niebieskiej. Z plamek o tych kolorach zbudowany jest obraz, który widzisz na swoim monitorze. W czasach początków komputerowego koloru, każda plamka mogła być albo włączona albo wyłączona, a więc natężenie każdej z trzech barw składowych możemy dla uproszczenia wyrazić za pomocą jednego bitu. Niech lewy bit odpowiada kolorowi niebieskiemu, bit środkowy – kolorowi zielonemu, a bit prawy – kolorowi czerwonemu. Osiem możliwych kombinacji kolorów linii przedstawia tabela 19.2.

Tablica 19.2. Prosta reprezentacja koloru

Układ bitów	Dziesiętnie	Kolor
000	0	czarny
001	1	czerwony
010	2	zielony
011	3	żółty
100	4	niebieski
101	5	fioletowy
110	6	niebieskozielony
111	7	biały

Oto jedna z możliwych reprezentacji danych. Definicja typu strukturalnego `line_style_t` korzysta z „pustych” pól, aby umieścić cechy związane z grubością i wzorem linii w jednym bajcie, a cechy związane z kolorem – w drugim.

```
typedef struct {
    unsigned int thickness : 3;
    unsigned int : 5;
    unsigned int pattern : 2;
    unsigned int color : 3;
    unsigned int : 3;
} line_style_t;
```

Listing 19.1 przedstawia prosty przykład wykorzystania typu strukturalnego `line_style_t`. Możliwe wartości składników są w nim wyrażone za pomocą stałych wyliczeniowych. Zauważ, że w przypadku barw podstawowych włączony jest tylko jeden z trzech bitów przeznaczonych do przechowania koloru. Inne barwy uzyskiwane są przez łączenie barw podstawowych. Na przykład, kolor fioletowy powstaje przez włączenie bitu „czerwonego” i bitu „niebieskiego”, można więc go wyrazić za pomocą wyrażenia `RED | BLUE`. Zauważ, że tablica kolory została tak zdefiniowana, aby każdy jej element zawierał nazwę koloru o takim samym numerze, jak indeks elementu – przykładowo, element o indeksie 1 zawiera nazwę „czerwony", a kolor czerwony ma numer 1.

Listing 19.1. Przykład: Obsługa stylu linii

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

typedef enum {
    SOLID = 0,
    DASHED = 1 << 0,
    DOTTED = 1 << 1
} pattern_t;

typedef enum {
    BLACK = 0,
    RED = 1 << 0,
    GREEN = 1 << 1,
    BLUE = 1 << 2,
    YELLOW = RED | GREEN,
    PURPLE = RED | BLUE,
    CYAN = GREEN | BLUE,
    WHITE = RED | GREEN | BLUE
} color_t;

typedef struct {
    unsigned int thickness : 3;
    unsigned int : 5;
    unsigned int pattern : 2;
    unsigned int color : 3;
    unsigned int : 3;
} line_style_t;

const char* color_names[8] = {"czarny", "czerwony", "zielony", "zolty",
                              "niebieski", "fioletowy", "zielononiebieski",
                              "bialy"};

int main(void) {
    line_style_t style = {.thickness = 4, .pattern = DOTTED, .color = CYAN};

    printf("Grubosc: %d\n", style.thickness);

    printf("Kolor wypelnienia: %s\n", color_names[style.color]);
```

```
    printf("Wzor: ");
    switch (style.pattern) {
        case SOLID:
            printf("ciagla\n");
            break;
        case DASHED:
            printf("kreskowana\n");
            break;
        case DOTTED:
            printf("kropkowana\n");
            break;
        default :
            printf("nieznanego typu\n");
    }

    return EXIT_SUCCESS;
}
```

Oto dane wyjściowe:

```
Grubosc: 4
Kolor wypelnienia: zielononiebieski
Wzor: kropkowana
```

Zwróć uwagę, że inicjalizacja i korzystanie ze struktury złożonej z pól bitowych wygląda identycznie, jak w przypadku „zwykłej” struktury.

19.2.2 Pola bitowe a unie

Pola bitowe są przydatne w połączeniu z uniami – poniższy przykład demonstruje typ unii pozwalającej zarówno na manipulowanie poszczególnymi bitami 8-bitowej wartości (poprzez odwoływanie się do składowych pola-struktury bits), jak i traktowanie jej jako całości i manipulowanie z użyciem masek (poprzez odwoływanie się do pola `byte`):

```
typedef union
{
    struct {
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char reserved:4;
    } bits;
    unsigned char byte;
} Register;
Register reg;
```

Z użyciem typu unii `Register` możesz zarówno operować na poszczególnych bitach pamięci:

```
reg.bits.b2 = 1;
```

jak i jednocześnie na wielu bitach:

```
reg.byte = 0xFA;
```

Rozdział 20

Biblioteka języka C

Standard języka C definiuje szereg przydatnych funkcji i makr, dostępnych w każdej (zgodnej ze standardem) implementacji języka C; część z nich już znasz, np. funkcję `printf()` czy makro `EXIT_SUCCESS`. W niniejszym rozdziale poznasz kilka innych funkcji oraz nauczysz się, jak efektywnie korzystać z biblioteki.

20.1 Uzyskiwanie dostępu do biblioteki języka C

Korzystanie z funkcji bibliotecznych wymaga zwykle dołączenia odpowiedniego pliku nagłówkowego – informację o tym, który plik należy dołączyć w przypadku danej funkcji, znajdziesz w dokumentacji kompilatora (dany plik nagłówkowy zwykle grupuje funkcjonalności zbliżone tematycznie). Niektóre kompilatory oraz środowiska programistyczne automatycznie (niejawnie) dołączają do programu najczęściej wykorzystywane pliki nagłówkowe (np. `stdlib.h` albo `stdio.h`), jednak do dobrych praktyk należy jawne wyszczególnienie *wszystkich* niezbędnych plików nagłówkowych, z których funkcjonalności korzysta dany fragment kodu – dzięki temu program będzie się kompilował niezależnie od użytego środowiska.

20.2 Korzystanie z dokumentacji biblioteki

O tym, jak korzystać z opisów funkcji bibliotecznych przeczytasz tu.

20.3 Funkcje matematyczne (*math.h*)

Pliku nagłówkowy `math.h` zawiera deklaracje m.in. funkcji: trygonometrycznych, i cyklometrycznych, $\ln(x)$, $\log_{10}(x)$, x^y, $\sqrt{x}$, $\exp(x)$, $|x|$, $\lfloor x \rfloor$ i $\lceil x \rceil$. Zwróć uwagę, że wszystkie kąty są mierzone w radianach ($1\,\mathrm{rad} = 180°/\pi = 57.296°$).

20.4 Narzędzia ogólnego użytku (*stdlib.h*)

Plik nagłówkowy `stdlib.h` zawiera deklaracje funkcji m.in. do zarządzania pamięcią, do generowania liczb pseudolosowych oraz do wyszukiwania elementu w tablicy i do jej sortowania.

20.4.1 Funkcja `exit()`

Funkcja `exit()` powoduje zakończenie programu, przy czym wykonuje jednocześnie pewne „czynności porządkowe”: opróżnia wszystkie bufory wyjściowe, zamyka wszystkie otwarte strumienie i usuwa pliki tymczasowe utworzone za pomocą standardowej funkcji `tmpfile()`; następnie zwraca kontrolę nad komputerem systemowi operacyjnemu, przekazując mu równocześnie kod zakończenia programu. W systemie UNIX kod `0` oznacza pomyślne zakończenie, a kod niezerowy – niepowodzenie (np. wystąpienie błędu). Ponieważ kody zakończenia programu systemu UNIX nie muszą działać we wszystkich środowiskach, standard języka C definiuje stałe `EXIT_SUCCESS` i `EXIT_FAILURE`, które pozwalają w bardziej przenośny sposób sygnalizować odpowiednio zakończenie prawidłowe i błędne.

20.5 Funkcje znakowe (*ctype.h*)

Plik nagłówkowy `ctype.h` zawiera m.in. deklaracje:

- funkcji analizujących typ znaku – zwracają one wartość niezerową („prawdę”) jeśli znak należy do określonej kategorii, w przeciwnym wypadku wartością zwracaną jest zero (fałsz); przykłady takich funkcji to: `isdigit()` – czy znak jest cyfrą? `islower()` – czy znak jest małą literą? itp.

- funkcje odwzorowujące `tolower()` i `toupper()`; działają one następująco: jeśli argument jest literą, zwracają odpowiednio mały lub wielki odpowiednik tej litery, w przeciwnym razie funkcje zawracją pierwotny argument (funkcje nie zmieniają argumentu).

Listing 20.1. Przykład użycia funkcji z biblioteki ctype.h

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

int main(void) {

    const char* s = "Ala ma kota!";
    int n_letters = 0;

    for (const char* p = s; *p != '\0'; ++p) {
        if (isalpha(*p)) {
            ++n_letters;
        }
    }

    printf("Fraza `%s` zawiera %d liter.\n", s, n_letters);

    return EXIT_SUCCESS;
}
```

20.6 Asercje (*assert.h*)

Plik nagłówkowy `assert.h` zawiera jedno makro o nazwie `assert()`, pomocne przy debugowaniu programów. Pobiera ono jako argument wyrażenie całkowite: jeśli wyrażenie to jest fałszywe (różne od zera), makro `assert()` wysyła komunikat do standardowego wyjścia dla błędów (`stderr`) i wywołuje funkcję `abort()`, która kończy program. (Prototyp funkcji `abort()` znajduje się w pliku `stdlib.h`.) Zastosowanie makra `assert()` polega na zlokalizowaniu tych miejsc w programie, w których spełnione muszą być pewne warunki, a następnie umieszczeniu w tych miejscach instrukcji `assert()` tak, aby program ulegał zakończeniu, jeśli nie są one spełnione. Argument makra `assert()` jest zazwyczaj wyrażeniem logicznym lub relacyjnym. Przed zakończeniem programu makro wyświetla warunek, który nie został spełniony, nazwę pliku, który go zawiera, oraz numer wiersza. Krótki przykład użycia makra `assert()` przedstawia listing 20.2; przed obliczeniem pierwiastka kwadratowego z liczby x program upewnia się, czy jest spełniona nierówność $x \geq 0$.

Listing 20.2. Wykorzystanie asercji

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

int main(void) {

    double x = -1.0;

    assert(x >= 0);
    printf("sqrt(x) = %lf\n", x);

    return EXIT_SUCCESS;
}
```

Wynik wykonania programu:

```
Assertion failed!

Program: C:\...\cmake-build-debug\c_playground.exe
```

```
File: C:\...\c_playground\main.c, Line 10

Expression: x >= 0

(...)
```

Komunikat `z >= 0` stwierdza, że warunek ten nie był spełniony. Podobny efekt możesz uzyskać za pomocą instrukcji **`if`**:

```
if (z < 0) {
  puts("z jest mniejsze od 0");
  abort();
}
```

Użycie makra `assert()` ma jednak kilka istotnych zalet:

- `assert()` automatycznie podaje numer wiersza, w którym wystąpił problem, oraz
- istnieje mechanizm pozwalający włączać i wyłączać makro `assert()` bez modyfikowania kodu.

Po usunięciu z programu wszystkich błędów, przed miejscem, w którym dołączony jest plik `assert.h`, wystarczy dodać wiersz

```
#define NODEBUG
```

a następnie ponownie skompilować program – kompilator zignoruje wszystkie instrukcje `assert()` w pliku źródłowym. W razie wystąpienia problemów wystarczy usunąć dyrektywę **`#define`** i jeszcze raz skompilować program.

Podziękowania

Osoby, które pomogły w tworzeniu niniejszego skryptu, są wymienione w kolejności alfabetycznej.

Korekta

Jacek Ankowski
Hubert Czader
Filip Gacek
Krzysztof Kordal
Dorota Kowalczyk
Kacper Kozaczko
Piotr Łuba
Artur Morys-Magiera
Paweł Mańka
Jakub Mazur
Kacper Motyka
Tomasz Niedziela
Marcin Pilarski
Łukasz Rams
Radosław Szpot
Daniel Węgrzynowski
Olaf Zdziebko