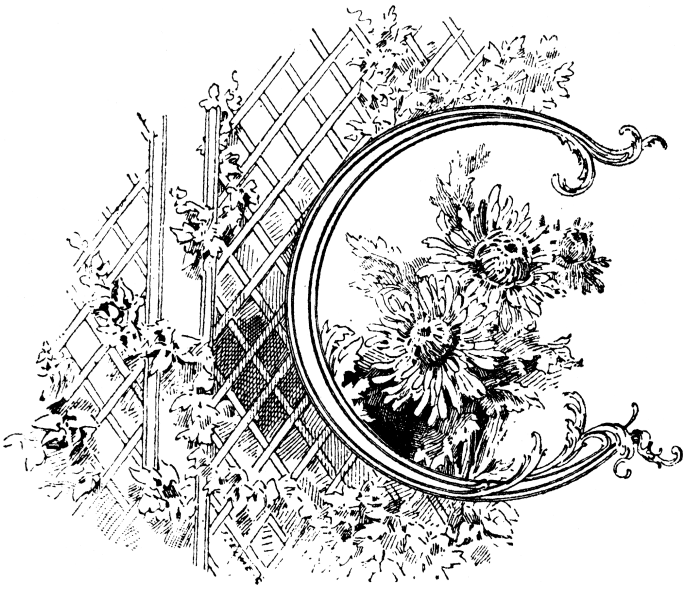
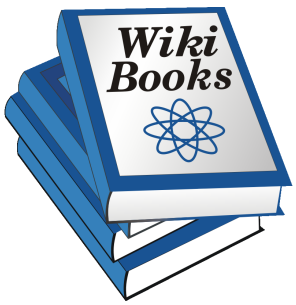
Programowanie w C





Stworzone na Wikibooks,
bibliotece wolnych podręczników.

Wydanie I z dnia 17 lutego 2008

Spis treści

Spis tabel

Rozdział 1

O podręczniku

O czym mówi ten podręcznik?

Niniejszy podręcznik stanowi przewodnik dla początkujących programistów po języku programowania C.

Co trzeba wiedzieć, żeby skorzystać z niniejszego podręcznika?

Ten podręcznik ma nauczyć programowania w C od podstaw do poziomu zaawansowanego. Do zrozumienia rozdziału dla początkujących wymagana jest jedynie znajomość podstawowych pojęć z zakresu algebry oraz terminów komputerowych. Doświadczenie w programowaniu w innych językach bardzo pomaga, ale nie jest konieczne.

Konwencje przyjęte w tym podręczniku

Informacje ważne oznaczamy w następujący sposób:

> Ważna informacja!

Dodatkowe informacje, które odrobinę wykraczają poza zakres podręcznika, a także wyjaśniają kwestie niezwiązane bezpośrednio z językiem C oznaczamy tak:

> Wyjaśnienie

Ponadto kod w języku C będzie prezentowany w następujący sposób:

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  return 0;
}
```

Innego rodzaju przykłady, dialog użytkownika z konsolą i programem, wejście / wyjście programu, informacje teoretyczne będą wyglądały tak:

```
typ zmienna = wartość;
```

Czy mogę pomóc?

Oczywiście że możesz. Mało tego, będziemy zadowoleni z każdej pomocy – możesz pisać rozdziały lub tłumaczyć je z angielskiej wersji tego podręcznika. Nie musisz pytać się nikogo o zgodę — jeśli chcesz, możesz zacząć już teraz. Prosimy jedynie o zapoznanie się ze stylem podręcznika, użytymi w nim szablonami i zachowanie układu rozdziałów. Propozycje zmiany spisu treści należy zgłaszać na stronie dyskusji.

Jeśli znalazłeś jakiś błąd a nie umiesz go poprawić, koniecznie powiadom o tym fakcie autorów tego podręcznika za pomocą strony dyskusji danego modułu książki. Dzięki temu przyczyniasz się do rozwoju tego podręcznika.

Autorzy

Istotny wkład w powstanie podręcznika mają:

- CzarnyZajaczek
- Derbeth
- Kj
- mina86

Dodatkowo w rozwoju podręcznika pomagali między innymi:

- Lrds
- Noisy

Źródła

- podręcznik C Programming na anglojęzycznej wersji Wikibooks, licencja GFDL
- Brian W. Kernighan, Dennis M. Ritchie, *Język ANSI C*
- ISO C Committee Draft, 18 styczna 1999
- Bruce Eckel, *Thinking in C++*. Rozdział Język C w programie C++.

Rozdział 2

O języku C

Zobacz w Wikipedii: C (język programowania)

C jest językiem programowania wysokiego poziomu. Jego nazwę interpretuje się jako następną literę po B (nazwa jego poprzednika), lub drugą literę języka BCPL (poprzednik języka B).

Historia C

W 1947 roku trzej naukowcy z Bell Telephone Laboratories — William Shockley, Walter Brattain i John Bardeen — stworzyli pierwszy tranzystor; w 1956 roku, w MIT skonstruowano pierwszy komputer oparty wyłącznie na tranzystorach: TX-O; w 1958 roku Jack Kilby z Texas Instruments skonstruował układ scalony. Ale zanim powstał pierwszy układ scalony, pierwszy język wysokiego poziomu został już napisany.

W 1954 powstał Fortran (Formula Translator), który zapoczątkował napisanie języka Fortran I (1956). Później powstały kolejno:

- Algol 58 — Algorithmic Language w 1958 r.
- Algol 60 (1960)
- CPL — Combined Programming Language (1963)
- BCPL — Basic CPL (1967)
- B (1969)

i C w oparciu o B.

B został stworzony przez Kena Thompsona z Bell Labs; był to język interpretowany, używany we wczesnych, wewnętrznych wersjach systemu operacyjnego UNIX. Inni pracownicy Bell Labs, Thompson i Dennis Richie, rozwinęli B, nazywając go NB; dalszy rozwój NB dał C — język kompilowany. Większa część UNIXa została ponownie napisana w NB, a następnie w C, co dało w efekcie bardziej przenośny system operacyjny. W 1978 roku wydana została książka pt. “The C Programming Language”, która stała się pierwszym podręcznikiem do nauki języka C.

Możliwość uruchamiania UNIX-a na różnych komputerach była główną przyczyną początkowej popularności zarówno UNIX-a, jak i C; zamiast tworzyć nowy system operacyjny, programiści mogli po prostu napisać tylko te części systemu, których wymagał inny sprzęt, oraz napisać kompilator C dla nowego systemu. Odkąd większa

część narzędzi systemowych była napisana w C, logiczne było pisanie kolejnych w tym samym języku.

Kilka z obecnie powszechnie stosowanych systemów operacyjnych takich jak Linux, Microsoft Windows zostały napisane w języku C.

Standaryzacje

W 1978 roku Ritchie i Kerninghan opublikowali pierwszą książkę nt. języka C — "The C Programming Language". Owa książka przez wiele lat była swoistym "wyznacznikiem", jak programować w języku C. Była więc to niejako pierwsza standaryzacja, nazywana od nazwisk twórców "K&R". Oto nowości, wprowadzone przez nią do języka C w stosunku do jego pierwszych wersji (pochodzących z początku lat 70.):

- możliwość tworzenia struktur (słowo **struct**)
- dłuższe typy danych (modyfikator **long**)
- liczby całkowite bez znaku (modyfikator **unsigned**)
- zmieniono operator "=+" na "+="

Ponadto producenci kompilatorów (zwłaszcza AT&T) wprowadzali swoje zmiany, nieobjęte standardem:

- funkcje nie zwracające wartości (void) oraz typ void*
- funkcje zwracające struktury i unie
- przypisywanie wartości strukturom
- wprowadzenie słowa kluczowego **const**
- utworzenie biblioteki standardowej
- wprowadzenie słowa kluczowego **enum**

Owe nieoficjalne rozszerzenia zagroziły spójności języka, dlatego też powstał standard, regulujący wprowadzone nowinki. Od 1983 roku trwały prace standaryzacyjne, aby w 1989 roku wydać standard C89 (poprawna nazwa to: ANSI X3.159-1989). Niektóre zmiany wprowadzono z języka C++, jednak rewolucję miał dopiero przynieść standard C99, który wprowadził m.in.:

- funkcje inline
- nowe typy danych (np. long long int)
- nowy sposób komentowania, zapożyczony od C++ (//)
- przechowywanie liczb zmiennoprzecinkowych zostało zaadaptowane do norm IEEE
- utworzono kilka nowych plików nagłówkowych (stdbool.h, inttypes.h)

Na dzień dzisiejszy normą obowiązującą jest norma C99.

Zastosowania języka C

Język C został opracowany jako strukturalny język programowania do celów ogólnych. Przez całą swą historię (czyli ponad 30 lat) służył do tworzenia przeróżnych programów — od systemów operacyjnych po programy nadzorujące pracę urządzeń przemysłowych. C, jako język dużo szybszy od języków interpretowanych (Perl, Python) oraz uruchamianych w maszynach wirtualnych (np. C#, Java) może bez problemu wykonywać złożone operacje nawet wtedy, gdy nałożone są dość duże limity czasu wykonywania pewnych operacji. Jest on przy tym bardzo przenośny — może działać praktycznie na każdej architekturze sprzętowej pod warunkiem opracowania odpowiedniego kompilatora. Często wykorzystywany jest także do oprogramowywania mikrokontrolerów i systemów wbudowanych. Jednak w niektórych sytuacjach język C okazuje się być mało przydatny, zwłaszcza chodzi tu o obliczenia matematyczne, wymagające dużej precyzji (w tej dziedzinie znakomicie spisuje się Fortran) lub też dużej optymalizacji dla danego sprzętu (wtedy niezastąpiony jest język asemblera).

Kolejną zaletą C jest jego dostępność — właściwie każdy system typu UNIX posiada kompilator C, w C pisane są funkcje systemowe.

Problemem w przypadku C jest zarządzanie pamięcią, które nie wybacza programiście błędów, niewygodne operowanie napisami i niestety pewna liczba "kruczków", które mogą zaskakiwać nowicjuszy. Na tle młodszych języków programowania, C jest językiem dosyć niskiego poziomu więc wiele rzeczy trzeba w nim robić ręcznie, jednak zarazem umożliwia to robienie rzeczy nieprzewidzianych w samym języku (np. implementację liczb 128 bitowych), a także łatwe łączenie C z Asemblerem.

Przyszłość C

Pomimo sędziwego już wieku (C ma ponad 30 lat) nadal jest on jednym z najczęściej stosowanych języków programowania. Doczekał się już swoich następców, z którymi w niektórych dziedzinach nadal udaje mu się wygrywać. Widać zatem, że pomimo pozornej prostoty i niewielkich możliwości język C nadal spełnia stawiane przed nim wymagania. Warto zatem uczyć się języka C, gdyż nadal jest on wykorzystywany (i nic nie wskazuje na to, by miało się to zmienić), a wiedza którą zdobędziesz ucząc się C na pewno się nie zmarnuje. Składnia języka C, pomimo że przez wielu uważana za nieczytelną, stała się podstawą dla takich języków jak C++, C# czy też Java.

Rozdział 3

Czego potrzebujesz

Czego potrzebujesz

Wbrew powszechnej opinii nauczenie się któregoś z języków programowania (w tym języka C) nie jest takie trudne. Do nauki wystarczą Ci:

- komputer z dowolnym systemem operacyjnym, takim jak FreeBSD, Linux, Windows;
- Język C jest bardzo przenośny, więc będzie działał właściwie na każdej platformie sprzętowej i w każdym nowoczesnym systemie operacyjnym.
- kompilator języka C
- Kompilator języka C jest programem, który tłumaczy kod źródłowy napisany przez nas do języka asembler, a następnie do postaci zrozumiałej dla komputera (maszyny cyfrowej) czyli do postaci ciągu zer i jedynek które sterują pracą poszczególnych elementów komputera. Kompilator języka C można dostać za darmo. Przykładem są: **gcc** pod systemy uniksowe, **DJGPP** pod systemy DOS, **MinGW** oraz **lcc** pod systemy typu Windows. Jako kompilator C może dobrze służyć kompilator języka C++ (różnice między tymi językami przy pisaniu prostych programów są nieistotne). Spokojnie możesz więc użyć na przykład Microsoft Visual C++® lub kompilatorów firmy Borland. Jeśli lubisz eksperymentować, wypróbuj Tiny C Compiler, bardzo szybki kompilator o ciekawych funkcjach. Możesz ponadto wypróbować interpreter języka C. Więcej informacji na Wikipedii.
- Linker
- Linker jest to program który uruchamiany jest po etapie kompilacji jednego lub kilku plików źródłowych (pliki z rozszerzeniem *.c, *.cpp lub innym) skompilowanych dowolnym kompilatorem. Taki program łączy wszystkie nasze skompilowane pliki źródłowe i inne funkcje (np. printf, scanf) które były użyte (dołączone do naszego programu poprzez użycie dyrektywy *#include*) w naszym programie, a nie były zdefiniowane(napisane przez nas) w naszych plikach źródłowych lub nagłówkowych. Linker jest to czasami jeden program połączony z kompilatorem. Wywoływany jest on na ogół automatycznie przez kompilator, w wyniku czego dostajemy gotowy program do uruchomienia.

- Debuger
- Debugger jest to program, który umożliwia prześledzenie(określenie wartości poszczególnych zmiennych na kolejnych etapach wykonywania programu) linijka po linijce wykonywania skompilowanego i zlinkowanego (skonsolidowanego) programu. Używa się go w celu określenia czemu nasz program nie działa po naszej myśli lub czemu program niespodziewanie kończy działanie bez powodu. Aby użyć debuggera kompilator musi dołączyć kod źródłowy do gotowego skompilowanego programu. Przykładowymi debuggerami są: *gdb* pod Linuksem, lub *debugger firmy Borland* pod Windowsa.
- edytora tekstowego;
- Systemy uniksowe oferują wiele edytorów przydatnych dla programisty, jak choćby vim i Emacs w trybie tekstowym, Kate w KDE czy gedit w GNOME. Windows ma edytor całkowicie wystarczający do pisania programów w C — nieśmiertelny Notatnik — ale z łatwością znajdziesz w Internecie wiele wygodniejszych narzędzi takich jak np. Notepad++. Odpowiednikiem Notepad++ w systemie uniksowym jest SciTE.
- dużo chęci i dobrej motywacji.

Zintegrowane Środowiska Programistyczne

Zamiast osobnego kompilatora i edytora, możesz wybrać Zintegrowane Środowisko Programistyczne (Integrated Development Environment, IDE). IDE jest zestawem wszystkich programów, które potrzebuje programista, najczęściej z interfejsem graficznym. IDE zawiera kompilator, linker i edytor, z reguły również debugger.

Bardzo popularny IDE to płatny Microsoft Visual C++ (MS VC++); popularne darmowe IDE to np.:

- Dev-C++ dla Windows, dostępny na stronie www.bloodshed.net,
- Code::Blocks dla Windows jak i Linux, dostępny na stronie www.codeblocks.org,
- KDevelop dla KDE
- Pelles C, www.smorgasbordet.com.
- Eclipse z wtyczką CDT (współpracuje z MinGW i GCC)

Często używanym środowiskiem jest też Borland C++ Builder (dostępny za darmo do użytku prywatnego).

Dodatkowe narzędzia

Wśród narzędzi, które nie są niezbędne, ale zasługują na uwagę, można wymienić Valgrinda – specjalnego rodzaju debugger. Valgrind kontroluje wykonanie programu i wykrywa nieprawidłowe operacje w pamięci oraz wycieki pamięci. Użycie Valgrinda jest proste — kompiluje się program tak, jak do debugowania i podaje jako argument Valgrindowi.

Rozdział 4

Używanie kompilatora

Język C jest językiem kompilowanym, co oznacza, że potrzebuje specjalnego programu — kompilatora — który tłumaczy kod źródłowy, pisany przez człowieka, na język rozkazów danego komputera. W skrócie działanie kompilatora sprowadza się do czytania tekstowego pliku z kodem programu, raportowania ewentualnych błędów i produkowania pliku wynikowego.

Kompilator uruchamiamy ze Zintegrowanego Środowiska Programistycznego lub z konsoli (linii poleceń). Przejść do konsoli można dla systemów typu UNIX w trybie graficznym użyć programów gterminal, konsole albo xterm, w Windows "Wiersz polecenia" (można go znaleźć w menu Akcesoria albo uruchomić wpisując w Start -> Uruchom... "cmd").

GCC

Zobacz w Wikipedii: GCC

GCC jest to darmowy zestaw kompilatorów, m.in. języka C rozwijany w ramach projektu GNU. Dostępny jest on na dużą ilość platform sprzętowych, obsługiwanych przez takie systemy operacyjne jak: AIX, *BSD, Linux, Mac OS X, SunOS, Windows.

Aby skompilować kod języka C za pomocą kompilatora GCC, napisany wcześniej w dowolnym edytorze tekstu, należy uruchomić program z odpowiednimi parametrami. Podstawowym parametrem, który jest wymagany, jest nazwa pliku zawierającego kod programu który chcemy skompilować.

```
gcc kod.c
```

Rezultatem kompilacji będzie plik wykonywalny, z domyślną nazwą (w systemach Unix jest to "a.out"). Jest to metoda niezalecana ponieważ jeżeli skompilujemy w tym samym katalogu kilka plików z kodem, kolejne pliki wykonywalne zostaną nadpisane i w rezultacie otrzymamy tylko jeden (ten ostatni) skompilowany kod. Aby wymusić na GCC nazwę pliku wykonywalnego musimy skorzystać z parametru "-o <nazwa>":

```
gcc -o program kod.c
```

W rezultacie otrzymujemy plik wykonywalny o nazwie program.

Jeżeli chodzi o formę nazwy podawanego pliku z kodem (w naszym przypadku kod.c), to kompilator sam rozpoznaje język programowania wynikający z rozszerzenia pliku. Jednak jest możliwość skompilowania pliku o dowolnym rozszerzeniu, narzucając

kompilatorowi dany język. Aby wymusić na GCC korzystanie z języka C, używamy parametru "-x <język>":

```
gcc -x c -o program kod
```

W rezultacie kompilator potraktuje plik "kod", jako plik z kodem źródłowym w języku C.

Pracując nad złożonym programem składającym się z kilku plików źródłowych (.c), możemy skompilować je niezależnie od siebie, tworząc tak zwane pliki typu obiekt (ang. *Object File*), są to pliki skompilowane ale nie poddane linkowaniu. Następnie stworzyć z nich jednolity program. Jest to bardzo wygodne i praktyczne rozwiązanie ze względu na to, iż nie jesteśmy zmuszeni kompilować wszystkich plików tworzące program za każdym razem na nowo a jedyne te w których wprowadziliśmy zmiany. Aby skompilować plik bez linkowania używamy parametru "-c <plik>":

```
gcc -o program1.o -c kod1.c
gcc -o program2.o -c kod2.c
```

Otrzymujemy w ten sposób pliki typu obiekt program1.o i program2.o. A następnie tworzymy z nich program główny:

```
gcc -o program program1.o program2.o
```

Aby włączyć dokładne, rygorystyczne sprawdzanie napisanego kodu, co może być przydatne, jeśli chcemy dążyć do perfekcji, używamy przełączników:

```
gcc kod.c -o program -Werror -Wall -W -pedantic -ansi
```

Więcej informacji na temat parametrów i działania kompilatora GCC można znaleźć na:

- Strona domowa projektu GNU GCC
- Krótki przekrojowy opis GCC po polsku
- Strona podręcznika systemu UNIX (man)

Borland

Zobacz podręcznik Borland C++ Compiler.

Czytanie komunikatów o błędach

Jedną z najbardziej podstawowych umiejętności, które musi posiąść początkujący programista jest umiejętność rozumienia komunikatów o różnego rodzaju błędach, które sygnalizuje kompilator. Wszystkie te informacje pomogą Ci szybko wychwycić ewentualne błędy (których na początku zawsze jest bardzo dużo). Nie martw się, że na początku dość często będziesz oglądał wydruki błędów, zasygnalizowanych przez kompilator — nawet zaawansowanym programistom się to zdarza. Kompilator ma za zadanie pomóc Ci w szybkiej poprawie ewentualnych błędów, dlatego też umiejętność analizy komunikatów o błędach jest tak ważna.

GCC

Kompilator jest w stanie wychwycić błędy składniowe, które z pewnością będziesz popełniał. Kompilator GCC wyświetla je w następującej formie:

```
nazwa_pliku.c:numer_linijki:opis błędu
```

Kompilator dość często podaje także nazwę funkcji, w której wystąpił błąd. Przykładowo, błąd deklaracji zmiennej w pliku test.c:

```
#include <stdio.h>

int main ()
{
  intr r;
  printf ("%d\n", r);
}
```

Spowoduje wygenerowanie następującego komunikatu o błędzie:

```
test.c: In function ‘main’:
test.c:5: error: ‘intr’ undeclared (first use in this function)
test.c:5: error: (Each undeclared identifier is reported only once
test.c:5: error: for each function it appears in.)
test.c:5: error: syntax error before ‘r’
test.c:6: error: ‘r’ undeclared (first use in this function)
```

Co widzimy w raporcie o błędach? W linii 5 użyliśmy nieznanego (undeclared) identyfikatora `intr` — kompilator mówi, że nie zna tego identyfikatora, jest to pierwsze użycie w danej funkcji i że więcej nie ostrzeże o użyciu tego identyfykatora w tej funkcji. Ponieważ `intr` nie został rozpoznany jako żaden znany typ, linijka `intr r;` nie została rozpoznana jako deklaracja zmiennej i kompilator zgłasza błąd składniowy (syntax error). W konsekwencji r nie zostało rozpoznane jako zmienna i kompilator zgłosi to jeszcze w następnej linijce, gdzie używamy r.

Rozdział 5

Pierwszy program

Twój pierwszy program

Przyjęło się, że pierwszy program napisany w dowolnym języku programowania, powinien wyświetlić tekst np. "Hello World!" (Witaj Świecie!). Zauważ, że sam język C nie ma żadnych mechanizmów przeznaczonych do wprowadzania i wypisywania danych — musimy zatem skorzystać ze specjalnie napisanych w tym celu funkcji — w tym przypadku *printf*, zawartej w standardowej bibliotece C (ang. *C Standard Library*) (podobnie jak w Pascalu używa się do tego procedur. Pascalowskim odpowiednikiem funkcji *printf* są procedury *write*/*writeln*).

W języku C deklaracje funkcji zawarte są w *plikach nagłówkowych* posiadających najczęściej rozszerzenie *.h*, choć można także spotkać rozszerzenie *.hpp*, przy czym to drugie zwykło się stosować w języku C++ (rozszerzenie nie ma swych "technicznych" korzeni — jest to tylko pewna konwencja). Żeby włączyć plik nagłówkowy do swojego kodu, trzeba użyć dyrektywy kompilacyjnej *#include*. Ta dyrektywa powoduje, że przed procesem kompilacji danego pliku źródłowego, deklaracje funkcji z pliku nagłówkowego zostają dołączone do twojego kodu celem zweryfikowania poprawności wywoływanych funkcji.

Poniżej przykład, jak użyć dyrektywy #include żeby wkleić definicję funkcji *printf* z pliku nagłówkowego stdio.h (Standard Input/Output.Headerfile):

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

W nawiasach trójkątnych < > umieszcza się nazwy standardowych plików nagłówkowych. Żeby włączyć inny plik nagłówkowy (np. własny), znajdujący się w katalogu z kodem programu, trzeba go wpisać w cudzysłów:

```
#include "mój_plik_nagłówkowy.h"
```

Mamy więc funkcję *printf*, jak i wiele innych do wprowadzania i wypisywania danych, czas na pisanie programu. [1]

Programy w C zaczyna się funkcją *main*, w której umieszcza się właściwy kod programu. Żeby rozpocząć tę funkcję, należy wpisać:

[1] Domyślne pliki nagłówkowe znajdują się w katalogu z plikami nagłówkowymi kompilatora. W systemach z rodziny Unix będzie to katalog /usr/include, natomiast w systemie Windows ów katalog będzie umieszczony w katalogu z kompilatorem.

```
int main (void)
{
```

int oznacza, że funkcja zwróci (tzn. przyjmie wartość po zakończeniu) liczbę całkowitą — w przypadku *main* będzie to kod wyjściowy programu; *main* to nazwa funkcji, w nawiasach umieszczamy *parametry* programu. Na tym etapie parametry programu nie będą nam potrzebne (*void* oznacza brak parametrów). Używa się ich do odczytywania argumentów linii polecenia, z jakimi program został uruchomiony.

Kod funkcji umieszcza się w nawiasach klamrowych \ i \.

Wewnątrz funkcji należy wpisać poniższy kod:

```
printf("Hello World!");
return 0;
```

Wszystkie polecenia kończymy średnikiem. *return 0;* określa wartość jaką zwróci funkcja (program); Liczba zero zwracana przez funkcję *main()* oznacza, że program zakończył się bez błędów; błędne zakończenie często (choć nie zawsze) określane jest przez liczbę jeden[2]. Funkcję *main* kończymy nawiasem klamrowym zamykającym.

Twój kod powinien wyglądać jak poniżej:

```
#include <stdio.h>
int main (void)
{
  printf ("Hello World!");
  return 0;
}
```

Teraz wystarczy go tylko skompilować i uruchomić.

Rozwiązywanie problemów

Jeśli nie możesz skompilować powyższego programu, to najprawdopodobniej popełniłeś literówkę przy ręcznym przepisywaniu go. Zobacz też instrukcje na temat używania kompilatora.

Może też się zdarzyć, że program skompiluje się, uruchomi, ale jego efektu działania nie będzie widać. Dzieje się tak, ponieważ nasz pierwszy program po prostu wypisuje komunikat i od razu kończy działanie, nie czekając na reakcję użytkownika. Nie jest to problemem, gdy program jest uruchamiany z konsoli tekstowej, ale w innych przypadkach nie widzimy efektów jego działania.

Jeśli korzystasz ze Zintegrowanego Środowiska Programistycznego (ang. IDE), możesz zaznaczyć, by nie zamykało ono programu po zakończeniu jego działania. Innym sposobem jest dodanie instrukcji, które wstrzymywałyby zakończenie programu. Można to zrobić dodając przed linią z `return` kod (wraz z okalającymi klamrami):

[2] Jeżeli chcesz mieć pewność, że twój program będzie działał poprawnie również na platformach, gdzie *1* oznacza poprawne zakończenie (lub nie oznacza nic), możesz skorzystać z makr `EXIT_SUCCESS` i `EXIT_FAILURE` zdefiniowanych w pliku nagłówkowym stdlib.h.

```
{
  int chr;
  puts("Wcisnij ENTER...");
  while ((chr = getchar())!=EOF && chr!='\n');
}
```

Jest też prostszy (choć nieprzenośny) sposób, mianowicie wywołanie polecenia systemowego. W zależności od używanego systemu operacyjnego mamy do dyspozycji różne polecenia powodujące różne efekty. Do tego celu skorzystamy z funkcji system(), która jako parametr przyjmuje polecenie systemowe które chcemy wykonać, np:

Rodzina systemów Unix/Linux:

```
system("sleep 10"); /* oczekiwanie 10 s */
system("read discard"); /* oczekiwanie na wpisanie tekstu */
```

Rodzina systemów DOS oraz MS Windows:

```
system("pause"); /* oczekiwanie na wciśnięcie dowolnego klawisza */
```

Funkcja ta jest o wiele bardziej pomocna w systemach operacyjnych Windows w których to z reguł pracuje się w trybie okienkowym a z konsoli korzystamy tylko podczas uruchamianiu programu. Z kolei w systemach Unix/Linux jest ona praktycznie w ogóle nie używana w tym celu, ze względu na uruchamianie programu bezpośrednio z konsoli.

Rozdział 6

Podstawy

Dla właściwego zrozumienia języka C nieodzowne jest przyswojenie sobie pewnych ogólnych informacji.

Kompilacja: Jak działa C?

Jak każdy język programowania, C sam w sobie jest niezrozumiały dla procesora. Został on stworzony w celu umożliwienia ludziom łatwego pisania kodu, który może zostać przetworzony na kod maszynowy. Program, który zamienia kod C na wykonywalny kod binarny, to ***kompilator***. Jeśli pracujesz nad projektem, który wymaga kilku plików kodu źródłowego (np. pliki nagłówkowe), wtedy jest uruchamiany kolejny program — ***linker***. Linker służy do połączenia różnych plików i stworzenia jednej aplikacji lub ***biblioteki*** (***library***). Biblioteka jest zestawem procedur, który sam w sobie nie jest wykonywalny, ale może być używana przez inne programy. Kompilacja i łączenie plików są ze sobą bardzo ściśle powiązane, stąd są przez wielu traktowane jako jeden proces. Jedną rzecz warto sobie uświadomić — kompilacja jest jednokierunkowa: przekształcenie kodu źródłowego C w kod maszynowy jest bardzo proste, natomiast odwrotnie — nie. Dekompilatory co prawda istnieją, ale rzadko tworzą użyteczny kod C.

Najpopularniejszym wolnym kompilatorem jest prawdopodobnie GNU Compiler Collection, dostępny na stronie gcc.gnu.org.

Co może C?

Pewnie zaskoczy Cię to, że tak naprawdę "czysty" język C nie może zbyt wiele. Język C w grupie języków programowania wysokiego poziomu jest stosunkowo nisko. Dzięki temu kod napisany w języku C można dość łatwo przetłumaczyć na kod asemblera. Bardzo łatwo jest też łączyć ze sobą kod napisany w języku asemblera z kodem napisanym w C. Dla bardzo wielu ludzi przeszkodą jest także dość duża liczba i częsta dwuznaczność operatorów. Początkujący programista, czytający kod programu w C może odnieść bardzo nieprzyjemne wrażenie, które można opisać cytatem "ja nigdy tego nie opanuję". Wszystkie te elementy języka C, które wydają Ci się dziwne i nielogiczne w miarę, jak będziesz nabierał doświadczenia nagle okażą się całkiem przemyślanie dobrane i takie, a nie inne konstrukcje przypadną Ci do gustu. Dalsza

lektura tego podręcznika oraz zaznajamianie się z funkcjami z różnych bibliotek ukażą Ci całą gamę możliwości, które daje język C doświadczonemu programiście.

Struktura blokowa

Teraz omówimy **podstawową strukturę programu napisanego w C**. Jeśli miałeś styczność z językiem Pascal, to pewnie słyszałeś o nim, że jest to język programowania **strukturalny**. W C nie ma tak ścisłej struktury blokowej, mimo to jest bardzo ważne zrozumienie, co oznacza **struktura blokowa**. **Blok** jest grupą instrukcji, połączonych w ten sposób, że są traktowane jak jedna całość. W C, blok zawiera się pomiędzy nawiasami klamrowymi \ \. Blok może także zawierać kolejne bloki.

Zawartość bloku. Generalnie, blok zawiera ciąg kolejno wykonywanych poleceń. Polecenia zawsze (z nielicznymi wyjątkami) kończą się średnikiem (;). W jednej linii może znajdować się wiele poleceń, choć dla zwiększenia czytelności kodu najczęściej pisze się pojedynczą instrukcję w każdej linii. Jest kilka rodzajów poleceń, np. instrukcje przypisania, warunkowe czy pętli. W dużej części tego podręcznika będziemy zajmować się właśnie instrukcjami.

Pomiędzy poleceniami są również odstępy — spacje, tabulacje, oraz przejścia do następnej linii, przy czym dla kompilatora te trzy rodzaje odstępów mają takie samo znaczenie. Dla przykładu, poniższe trzy fragmenty kodu źródłowego, dla kompilatora są takie same:

```
printf("Hello world"); return 0;
```

```
printf("Hello world");
return 0;
```

```
printf("Hello world");<br><br><br>
return 0;
```

W tej regule istnieje jednak jeden wyjątek. Dotyczy on stałych tekstowych. W powyższych przykładach stałą tekstową jest “Hello world”. Gdy jednak rozbijemy ten napis, kompilator zasygnalizuje błąd:

```
printf("Hello
world");
return 0;
```

Należy tylko zapamiętać, że stałe tekstowe powinny zaczynać się i kończyć w tej samej lini (można ominąć to ograniczenie — więcej w rozdziale Napisy). Oprócz tego jednego przypadku dla kompilatora ma znaczenie samo **istnienie** odstępu, a nie jego wielkość czy rodzaj. Jednak stosowanie odstępów jest bardzo ważne, dla zwiększenia czytelności kodu — dzięki czemu możemy zaoszczędzić sporo czasu i nerwów, ponieważ znalezienie błędu (które się zdarzają każdemu) w nieczytelnym kodzie może być bardzo trudne.

Zasięg

Pojęcie to dotyczy zmiennych (które przechowują dane przetwarzane przez program). W każdym programie (oprócz tych najprostszych) są zarówno zmienne wykorzystywane przez cały czas działania programu, oraz takie które są używane przez pojedynczy blok programu (np. funkcję). Na przykład, w pewnym programie w pewnym

momencie jest wykonywane skomplikowane obliczenie, które wymaga zadeklarowania wielu zmiennych do przechowywania pośrednich wyników. Ale przez większą część tego działania, te zmienne są niepotrzebne, i zajmują tylko miejsce w pamięci — najlepiej gdyby to miejsce zostało zarezerwowane tuż przed wykonaniem wspomnianych obliczeń, a zaraz po ich wykonaniu zwolnione. Dlatego w C istnieją zmienne **globalne**, oraz **lokalne**. Zmienne globalne mogą być używane w każdym miejscu programu, natomiast lokalne — tylko w określonym bloku czy funkcji (oraz blokach w nim zawartych). Generalnie — zmienna zadeklarowana w danym bloku, jest dostępna tylko wewnątrz niego.

Funkcje

Funkcje są ściśle związane ze strukturą blokową — funkcją jest po prostu blok instrukcji, który jest potem wywoływany w programie za pomocą pojedynczego polecenia. Zazwyczaj funkcja wykonuje pewne określone zadanie, np. we wspomnianym programie wykonującym pewne skomplikowane obliczenie. Każda funkcja ma swoją nazwę, za pomocą której jest potem wywoływana w programie, oraz blok wykonywanych poleceń. Wiele funkcji pobiera pewne dane, czyli argumenty funkcji, wiele funkcji także zwraca pewną wartość, po zakończeniu wykonywania. Dobrym nawykiem jest dzielenie dużego programu na zestaw mniejszych funkcji — dzięki temu będziesz mógł łatwiej odnaleźć błąd w programie.

Jeśli chcesz użyć jakiejś funkcji, to powinieneś wiedzieć:

- jakie zadanie wykonuje dana funkcja
- rodzaj wczytywanych argumentów, i do czego są one potrzebne tej funkcji
- rodzaj zwróconych danych, i co one oznaczają.

W programach w języku C jedna funkcja ma szczególne znaczenie — jest to **main()**. Funkcję tę, zwaną funkcją główną, musi zawierać każdy program. W niej zawiera się główny kod programu, przekazywane są do niej argumenty, z którymi wywoływany jest program (jako parametry `argc` i `argv`). Więcej o funkcji main() dowiesz się później w rozdziale Funkcje.

Biblioteki standardowe

Język C, w przeciwieństwie do innych języków programowania (np. Fortranu czy Pascala) nie posiada absolutnie **żadnych** słów kluczowych, które odpowiedzialne by były za obsługę wejścia i wyjścia. Może się to wydawać dziwne — język, który sam w sobie nie posiada podstawowych funkcji, musi być językiem o ograniczonym zastosowaniu. Jednak brak podstawowych funkcji wejścia-wyjścia jest jedną z największych zalet tego języka. Jego składnia opracowana jest tak, by można było bardzo łatwo przełożyć ją na kod maszynowy. To właśnie dzięki temu programy napisane w języku C są takie szybkie. Pozostaje jednak pytanie — jak umożliwić programom komunikację z użytkownikiem ?

W 1983 roku, kiedy zapoczątkowano prace nad standaryzacją C, zdecydowano, że powinien być zestaw instrukcji identycznych w każdej implementacji C. Nazwano je Biblioteką Standardową (czasem nazywaną “libc”). Zawiera ona podstawowe funkcje,

które umożliwiają wykonywanie takich zadań jak wczytywanie i zwracanie danych, modyfikowanie zmiennych łańcuchowych, działania matematyczne, operacje na plikach, i wiele innych, jednak nie zawiera żadnych funkcji, które mogą być zależne od systemu operacyjnego czy sprzętu, jak grafika, dźwięk czy obsługa sieci. W programie "Hello World" użyto funkcji z biblioteki standardowej — printf, która wyświetla na ekranie sformatowany tekst.

Komentarze i styl

Komentarze — to tekst włączony do kodu źródłowego, który jest pomijany przez kompilator, i służy jedynie dokumentacji. W języku C, komentarze zaczynają się od

```
/*
```

a kończą

```
*/
```

Dobre komentowanie ma duże znaczenie dla rozwijania oprogramowania, nie tylko dlatego, że inni będą kiedyś potrzebowali przeczytać napisany przez ciebie kod źródłowy, ale także możesz chcieć po dłuższym czasie powrócić do swojego programu, i możesz zapomnieć, do czego służy dany blok kodu, albo dlaczego akurat użyłeś tego polecenia a nie innego. W chwili pisania programu, to może być dla ciebie oczywiste, ale po dłuższym czasie możesz mieć problemy ze zrozumieniem własnego kodu. Jednak nie należy też wstawiać zbyt dużo komentarzy, ponieważ wtedy kod może stać się jeszcze mniej czytelny — najlepiej komentować fragmenty, które nie są oczywiste dla programisty, oraz te o szczególnym znaczeniu. Ale tego nauczysz się już w praktyce.

Dobry styl pisania kodu jest o tyle ważny, że powinien on być czytelny i zrozumiały; po to w końcu wymyślono języki programowania wysokiego poziomu (w tym C), aby kod było łatwo zrozumieć ;). I tak — należy stosować wcięcia dla odróżnienia bloków kolejnego poziomu (zawartych w innym bloku; podrzędnych), nawiasy klamrowe otwierające i zamykające blok powinny mieć takie same wcięcia, staraj się, aby nazwy funkcji i zmiennych kojarzyły się z zadaniem, jakie dana funkcja czy zmienna pełni w programie. W dalszej części podręcznika możesz napotkać więcej zaleceń dotyczących stylu pisania kodu. Staraj się stosować do tych zaleceń — dzięki temu kod pisanych przez ciebie programów będzie łatwiejszy do czytania i zrozumienia.

Jeśli masz doświadczenia z językiem C++ pamiętaj, że w C nie powinno się stosować komentarzy zaczynających się od dwóch znaków slash:
`// tak nie komentujemy w C`
Jest to niezgodne ze standardem ANSI C i niektóre kompilatory mogą nie skompilować kodu z komentarzami w stylu C++ (choć standard ISO C99 dopuszcza komentarze w stylu C++).

Innym zastosowaniem komentarzy jest chwilowe usuwanie fragmentów kodu. Jeśli część programu źle działa i chcemy ją chwilowo wyłączyć, albo fragment kodu jest nam już niepotrzebny, ale mamy wątpliwości, czy w przyszłości nie będziemy chcieli go użyć — umieszczamy go po prostu wewnątrz komentarza.

Podczas obejmowania chwilowo niepotrzebnego kodu w komentarz trzeba uważać na jedną subtelność. Otóż komentarze `/* * /` w języku C nie mogą być zagnieżdżone.

Trzeba na to uważać, gdy chcemy objąć komentarzem obszar w którym już istnieje komentarz (należy wtedy usunąć wewnętrzny komentarz). W nowszym standardzie C dopuszcza się, aby komentarz typu `/* */` zawierał w sobie komentarz `//`.

Po polsku czy angielsku?

Jak już wcześniej było wspomniane, zmiennym i funkcjom powinno się nadawać nazwy, które odpowiadają ich znaczeniu. Zdecydowanie łatwiej jest czytać kod, gdy średnią liczb przechowuje zmienna `srednia` niż `a` a znajdowaniem maksimum w ciągu liczb zajmuje się funkcja `max` albo `znajdz_max` niż nazwana `f`. Często nazwy funkcji to właśnie czasowniki.

Powstaje pytanie, w jakim języku należy pisać nazwy. Jeśli chcemy, by nasz kod mogły czytać osoby nieznające polskiego — warto użyć języka angielskiego. Jeśli nie — można bez problemu użyć polskiego. Bardzo istotne jest jednak, by nie mieszać języków. Jeśli zdecydowaliśmy się używać polskiego, używajmy go od początku do końca; przeplatanie ze sobą dwóch języków robi złe wrażenie.

Preprocesor

Nie cały napisany przez ciebie kod będzie przekształcany przez kompilator bezpośrednio na kod wykonywalny programu. W wielu przypadkach będziesz używać poleceń "skierowanych do kompilatora", tzw. dyrektyw kompilacyjnych. Na początku procesu kompilacji, specjalny podprogram, tzw. preprocesor, wyszukuje wszystkie dyrektywy kompilacyjne, i wykonuje odpowiednie akcje — które polegają notabene na edycji kodu źródłowego (np. wstawieniu deklaracji funkcji, zamianie jednego ciągu znaków na inny). Właściwy kompilator, zamieniający kod C na kod wykonywalny, nie napotka już dyrektyw kompilacyjnych, ponieważ zostały one przez preprocesor usunięte, po wykonaniu odpowiednich akcji.

W C dyrektywy kompilacyjne zaczynają się od znaku *hash* (#). Przykładem najczęściej używanej dyrektywy, jest `#include`, która jest użyta nawet w tak prostym programie jak "Hello, World!". `#include` nakazuje preprocesorowi włączyć (ang. include) w tym miejscu zawartość podanego pliku, tzw. pliku nagłówkowego; najczęściej to będzie plik zawierający funkcje z którejś biblioteki standardowej (stdio.h — STandard Input-Output, rozszerzenie .h oznacza plik nagłówkowy C). Dzięki temu, zamiast wklejać do kodu swojego programu deklaracje kilkunastu, a nawet kilkudziesięciu funkcji, wystarczy wpisać jedną magiczną linijkę!

Nazwy zmiennych, stałych i funkcji

Identyfikatory, czyli nazwy zmiennych, stałych i funkcji mogą składać się z liter (bez polskich znaków), cyfr i znaku podkreślenia z tym, że nazwa taka nie może zaczynać się od cyfry. Nie można używać nazw zarezerwowanych (patrz: Składnia).

Przykłady błędnych nazw:

```
2liczba      (nie można zaczynać nazwy od cyfry)
moja funkcja (nie można używać spacji)
$i           (nie można używać znaku $)
if           (if to słowo kluczowe)
```

Aby kod był bardziej czytelny, przestrzegajmy poniższych (umownych) reguł:

- nazwy zmiennych piszemy małymi literami: `i, file`
- nazwy stałych (zadeklarowanych przy pomocy #define) piszemy wielkimi literami: `SIZE`
- nazwy funkcji piszemy małymi literami: `print`
- wyrazy w nazwach oddzielamy znakiem podkreślenia: `open_file, close_all_files`

Są to tylko konwencje — żaden kompilator nie zgłosi błędu, jeśli wprowadzimy swój własny system nazewnictwa. Jednak warto pamiętać, że być może nad naszym kodem będą pracowali także inni programiści, którzy mogą mieć trudności z analizą kodu niespełniającego pewnych zasad.

Rozdział 7

Zmienne

Procesor komputera stworzony jest tak, aby przetwarzał dane, znajdujące się w pamięci komputera. Z punktu widzenia programu napisanego w języku C (który jak wiadomo jest językiem wysokiego poziomu) dane umieszczane są w postaci tzw. **zmiennych**. Zmienne ułatwiają programiście pisanie programu. Dzięki nim programista nie musi się przejmować gdzie w pamięci owe zmienne się znajdują, tzn. nie operuje fizycznymi adresami pamięci, jak np. 0x14613467, tylko prostą do zapamiętania nazwą zmiennej.

Czym są zmienne?

Zmienna jest to pewien fragment pamięci o ustalonym rozmiarze, który posiada własny identyfikator (nazwę) oraz może przechowywać pewną wartość, zależną od typu zmiennej.

Deklaracja zmiennych

Aby móc skorzystać ze zmiennej należy ją przed użyciem zadeklarować, to znaczy poinformować kompilator, jak zmienna będzie się nazywać i jaki typ ma mieć. Zmienne deklaruje się w sposób następujący:

```
typ nazwa_zmiennej;
```

Oto deklaracja zmiennej o nazwie “wiek” typu “int” czyli liczby całkowitej:

```
int wiek;
```

Zmiennej w momencie zadeklarowania można od razu przypisać wartość:

```
int wiek = 17;
```

W języku C zmienne deklaruje się na samym początku bloku (czyli przed pierwszą instrukcją).

```
int wiek = 17;
printf("%d\n", wiek);
```

```
int kopia_wieku; /* tu stary kompilator C zgłosi błąd */
                 /* deklaracja występuje po instrukcji (printf). */
kopia_wieku = wiek;
```

Według nowszych standardów możliwe jest deklarowanie zmiennej w dowolnym miejscu programu, ale wtedy musimy pamiętać, aby zadeklarować zmienną przed jej użyciem. To znaczy, że taki kod jest niepoprawny:

```
printf ("Mam %d lat\n", wiek);
int wiek = 17;
```

Należy go zapisać tak:

```
int wiek = 17;
printf ("Mam %d lat\n", wiek);
```

Język C nie inicjalizuje zmiennych lokalnych. Oznacza to, że w nowo zadeklarowanej zmiennej znajdują się śmieci - to, co wcześniej zawierał przydzielony zmiennej fragment pamięci. Aby uniknąć ciężkich do wykrycia błędów, dobrze jest inicjalizować (przypisywać wartość) wszystkie zmienne w momencie zadeklarowania.

Zasięg zmiennej

Zmienne mogą być dostępne dla wszystkich funkcji programu — nazywamy je wtedy **zmiennymi globalnymi**. Deklaruje się je przed wszystkimi funkcjami programu:

```
#include <stdio.h>

int a,b; /* nasze zmienne globalne */

void func1 ()
{
  /* instrukcje */
  a=3;
  /* dalsze instrukcje */
}

int main ()
{
  b=3;
  a=2;
  return 0;
}
```

Zmienne globalne, jeśli programista nie przypisze im innej wartości podczas definiowania, są inicjalizowane wartością 0.

Zmienne, które funkcja deklaruje do "własnych potrzeb" nazywamy **zmiennymi lokalnymi**. Nasuwa się pytanie: "czy będzie błędem nazwanie tą samą nazwą zmiennej globalnej i lokalnej?". Otóż odpowiedź może być zaskakująca: nie. Natomiast w danej funkcji da się używać tylko jej zmiennej lokalnej. Tej konstrukcji należy, z wiadomych względów, unikać.

```
int a=1; /* zmienna globalna */

int main()
{
  int a=2;          /* to już zmienna lokalna */
  printf("%d", a); /* wypisze 2 */
}
```

Czas życia

Czas życia to czas od momentu przydzielenia dla zmiennej miejsca w pamięci (stworzenie obiektu) do momentu zwolnienia miejsca w pamięci (likwidacja obiektu).

Zakres ważności to część programu, w której nazwa znana jest kompilatorowi.

```
main()
{
  int a = 10;
  {                             /* otwarcie lokalnego bloku */
    int b = 10;
    printf("%d %d", a, b);
  }                             /* zamknięcie lokalnego bloku, zmienna b jest usuwana */

  printf("%d %d", a, b);        /* BŁĄD: b juz nie istnieje */
}                                /* tu usuwana jest zmienna a */
```

Zdefiniowaliśmy dwie zmienne typu int. Zarówno a i b istnieją przez cały program (czas życia). Nazwa zmiennej a jest znana kompilatorowi przez cały program. Nazwa zmiennej b jest znana tylko w lokalnym bloku, dlatego nastąpi błąd w ostatniej instrukcji.

Niektóre kompilatory (prawdopodobnie można tu zaliczyć Microsoft Visual C++ do wersji 2003) uznają powyższy kod za poprawny! W dodatku można ustawić w opcjach niektórych kompilatorów zachowanie w takiej sytuacji, włącznie z zachowaniami niezgodnymi ze standardem języka!

Możemy świadomie ograniczyć ważność zmiennej do kilku linijek programu (tak jak robiliśmy wyżej) tworząc blok. Nazwa zmiennej jest znana tylko w tym bloku.

```
{
  ...
}
```

Stałe

Stała, różni się od zmiennej tylko tym, że nie można jej przypisać innej wartości w trakcie działania programu. Wartość stałej ustala się w kodzie programu i nigdy ona nie ulega zmianie. Stałą deklaruje się z użyciem słowa kluczowego **const** w sposób następujący:

```
const typ nazwa_stałej=wartość;
```

Dobrze jest używać stałych w programie, ponieważ unikniemy wtedy przypadkowych pomyłek a kompilator może często zoptymalizować ich użycie (np. od razu podstawiając ich wartość do kodu).

```
const int WARTOSC_POCZATKOWA=5;
int i=WARTOSC_POCZATKOWA;
WARTOSC_POCZATKOWA=4;  /* tu kompilator zaprotestuje */
int j=WARTOSC_POCZATKOWA;
```

Przykład pokazuje dobry zwyczaj programistyczny, jakim jest zastępowanie umieszczonych na stałe w kodzie liczb stałymi. W ten sposób będziemy mieli większą kontrolę nad kodem — stałe umieszczone w jednym miejscu można łatwo modyfikować, zamiast szukać po całym kodzie liczb, które chcemy zmienić.

Nie mamy jednak pełnej gwarancji, że stała będzie miała tę samą wartość przez cały czas wykonania programu. Możliwe jest dostanie się do wartości stałej pośrednio — za pomocą wskaźników. Można zatem dojść do wniosku, że słowo kluczowe const służy tylko do poinformowania kompilatora, aby ten nie zezwalał na jawną zmianę wartości stałej. Z drugiej strony, zgodnie ze standardem, próba modyfikacji wartości stałej ma niezdefiniowane działanie (tzw. *undefined behaviour*) i w związku z tym może się powieść lub nie, ale może też spowodować jakieś subtelne zmiany, które w efekcie spowodują, że program będzie źle działał.

Podobnie do zdefiniowania stałej możemy użyć dyrektywy preprocesora #define (opisanej w dalszej części podręcznika). Tak zdefiniowaną stałą nazywamy **stałą symboliczną**. W przeciwieństwie do stałej zadeklarowanej z użyciem słowa **const** stała zdefiniowana przy użyciu #define jest zastępowana daną wartością w każdym miejscu, gdzie występuje, dlatego też może być używana w miejscach, gdzie "normalna" stała nie mogłaby dobrze spełnić swej roli.

W przeciwieństwie do języka C++, w C stała to cały czas zmienna, której kompilator pilnuje, by nie zmieniła się. Z tego powodu w C nie można użyć stałej do określenia wielkości tablicy[1] i należy się w takim wypadku odwołać do wcześniej wspomnianej dyrektywy `#define`.

Typy zmiennych

Każdy program w C operuje na zmiennych — wydzielonych w pamięci komputera obszarach, które mogą reprezentować obiekty nam znane, takie jak liczby, znaki, czy też bardziej złożone obiekty. Jednak dla komputera każdy obszar w pamięci jest taki sam — to ciąg zer i jedynek, w takiej postaci zupełnie nieprzydatny dla programisty i użytkownika. Podczas pisania programu musimy wskazać, w jaki sposób ten ciąg ma być interpretowany.

Typ zmiennej wskazuje właśnie sposób, w jaki pamięć, w której znajduje się zmienna będzie wykorzystywana. Określając go przekazuje się kompilatorowi informację, ile pamięci trzeba zarezerwować dla zmiennej, a także w jaki sposób wykonywać na nim operacje.

Każda zmienna musi mieć określony swój typ w miejscu deklaracji i tego typu nie może już zmienić. Lecz co jeśli mamy zmienną jednego typu, ale potrzebujemy w pewnym miejscu programu innego typu danych? W takim wypadku stosujemy **konwersję**

[1] Jest to możliwe w standardzie C99

(**rzutowanie**) jednej zmiennej na inną zmienną. Rzutowanie zostanie opisane później, w rozdziale Operatory.

Istnieją wbudowane i zdefiniowane przez użytkownika typy danych. Wbudowane typy danych to te, które zna kompilator, są one w nim bezpośrednio "zaszyte". Można też tworzyć własne typy danych, ale należy je kompilatorowi opisać. Więcej informacji znajduje się w rozdziale Typy złożone.

W języku C wyróżniamy 4 podstawowe typy zmiennych. Są to:

char — jednobajtowe liczby całkowite, służy do przechowywania znaków; **int**- typ całkowity, o długości domyślnej dla danej architektury komputera; **float** — typ zmiennopozycyjny (4 bajty 6 miejsc po przecinku); **double** — typ zmiennopozycyjny podwójnej precyzji (8 bajtów 15 miejsc po przecinku);

Typy zmiennoprzecinkowe zostały dokładnie opisane w IEEE 754.

int

Ten typ przeznaczony jest do liczb całkowitych. Liczby te możemy zapisać na kilka sposobów:

- System dziesiętny

```
12 ; 13 ; 45 ; 35 itd
```

- System ósemkowy (oktalny)

```
010    czyli 8
016    czyli 8 + 6 = 14
019    BŁĄD
```

System ten operuje na cyfrach od 0 do 7. Tak wiec 9 jest niedozwolona. Jeżeli chcemy użyć takiego zapisu musimy zacząć liczbę od 0.

- System szesnastkowy (heksadycemalny)

```
0x10   czyli 1*16 + 0 = 16
0x12   czyli 1*16 + 2 = 18
0xff   czyli 15*16 + 15 = 255
```

W tym systemie możliwe cyfry to 0...9 i dodatkowo a, b, c, d, e, f, które oznaczają 10, 11, 12, 13, 14, 15. Aby użyć takiego systemu musimy poprzedzić liczbę ciągiem 0x. Wielkość znaków w takich literałach nie ma znaczenia.

float

Ten typ oznacza liczby zmiennoprzecinkowe czyli ułamki. Istnieją dwa sposoby zapisu:

- System dziesiętny

```
3.14 ; 45.644 ; 23.54 ; 3.21 itd
```

- System "naukowy" — wykładniczy

```
   6e2 czyli 6 * 10^2 czyli 600
1.5e3 czyli 1.5 * 10^3 czyli 1500
3.4e-3 czyli 3.4 * 10^-3 czyli 0.0034
```

Należy wziąć pod uwagę, że reprezentacja liczb rzeczywistych w komputerze jest niedoskonała i możemy otrzymywać wyniki o zauważalnej niedokładności.

double

Double — czyli "podwójny" — oznacza liczby zmiennoprzecinkowe podwójnej precyzji. Oznacza to, że liczba taka zajmuje zazwyczaj w pamięci dwa razy więcej miejsca niż float (np. 64 bity wobec 32 dla float), ale ma też dwa razy lepszą dokładność.

Domyślnie ułamki wpisane w kodzie są typu double. Możemy to zmienić dodając na końcu literę "f":

```
1.5f   (float)
1.5    (double)
```

char

Jest to typ znakowy, umożliwiający zapis znaków ASCII. Może też być traktowany jako liczba z zakresu 0..255. Znaki zapisujemy w pojedynczych cudzysłowach, by odróżnić je od łańcuchów tekstowych (pisanych w podwójnych cudzysłowach).

```
'a' ; '7' ; '!' ; '$'
```

Cudzysłów ' zapisujemy tak: `'\''` a NULL (czyli zero, które między innymi kończy napisy) tak: `'\0'`. Więcej znaków specjalnych.

Warto zauważyć, że typ char to zwykły typ liczbowy i można go używać tak samo jak typu int (zazwyczaj ma jednak mniejszy zakres). Co więcej literały znakowe (np. 'a') są traktowane jako liczby i w języku C są typu int (w języku C++ są typu char).

void

Słowa kluczowego `void` można w określonych sytuacjach użyć tam, gdzie oczekiwana jest nazwa typu. `void` nie jest właściwym typem, bo nie można utworzyć zmiennej takiego typu; jest to "pusty" typ (ang. void znaczy "pusty"). Typ `void` przydaje się do zaznaczania, że funkcja nie zwraca żadnej wartości lub że nie przyjmuje żadnych parametrów (więcej o tym w rozdziale Funkcje). Można też tworzyć zmienne będące typu "wskaźnik na void"

Specyfikatory

Specyfikatory to słowa kluczowe, które postawione przy typie danych zmieniają jego znaczenie.

signed i unsigned

Na początku zastanówmy się, jak komputer może przechować liczbę ujemną. Otóż w przypadku przechowywania liczb ujemnych musimy w zmiennej przechować jeszcze jej znak. Jak wiadomo, zmienna składa się z szeregu bitów. W przypadku użycia

zmiennej pierwszy bit z lewej strony (nazywany także bitem **najbardziej znaczącym**) przechowuje znak liczby. Efektem tego jest spadek "pojemności" zmiennej, czyli zmniejszenie największej wartości, którą możemy przechować w zmiennej.

Signed oznacza liczbę ze znakiem, unsigned — bez znaku (nieujemną). Mogą być zastosowane do typów: char i int i łączone ze specyfikatorami short i long (gdy ma to sens).

Jeśli przy signed lub unsigned nie napiszemy, o jaki typ nam chodzi, kompilator przyjmie wartość domyślną czyli int.

Przykładowo dla zmiennej `char`(zajmującej 8 bitów zapisanej w formacie uzupełnień do dwóch) wygląda to tak:

```
signed char a;      /* zmienna a przyjmuje wartości od -128 do 127 */
unsigned char b;    /* zmienna b przyjmuje wartości od 0 do 255    */
unsigned short c;
unsigned long int d;
```

Jeżeli nie podamy żadnego ze specyfikatora wtedy liczba jest domyślnie przyjmowana jako signed (nie dotyczy to typu char, dla którego jest to zależne od kompilatora).

```
signed int i = 0;
// jest równoznaczne z:
int i = 0;
```

Liczby bez znaku pozwalają nam zapisać większe liczby przy tej samej wielkości zmiennej — ale trzeba uważać, by nie zejść z nimi poniżej zera — wtedy "przewijają" się na sam koniec zakresu, co może powodować trudne do wykrycia błędy w programach.

short i long

Short i long są wskazówkami dla kompilatora, by zarezerwował dla danego typu mniej (odpowiednio — więcej) pamięci. Mogą być zastosowane do dwóch typów: int i double (tylko long), mając różne znaczenie.

Jeśli przy short lub long nie napiszemy, o jaki typ nam chodzi, kompilator przyjmie wartość domyślną czyli int.

Należy pamiętać, że to jedynie życzenie wobec kompilatora — w wielu kompilatorach typy int i long int mają ten sam rozmiar. Standard języka C nakłada jedynie na kompilatory następujące ograniczenia: `int` — nie może być krótszy niż 16 bitów; `int` — musi być dłuższy lub równy `short` a nie może być dłuższy niż `long`; `short int` — nie może być krótszy niż 16 bitów; `long int` — nie może być krótszy niż 32 bity;

Zazwyczaj typ `int` jest typem danych o długości odpowiadającej wielkości rejestrów procesora, czyli na procesorze szesnastobitowym ma 16 bitów, na trzydziestodwubitowym — 32 itd.[2]. Z tego powodu, jeśli to tylko możliwe, do reprezentacji liczb całkowitych preferowane jest użycie typu int bez żadnych specyfikatorów rozmiaru.

[2]Wiąże się to z pewnymi uwarunkowaniami historycznymi. Podręcznik do języka C duetu K&R zakładał, że typ int miał się odnosić do **typowej** dla danego procesora długości liczby całkowitej. Natomiast, jeśli procesor mógł obsługiwać typy dłuższe lub krótsze stosownego znaczenia nabierały modyfikatory short i long. Dobrym przykładem może być architektura i386, która umożliwia obliczenia na liczbach 16-bitowych. Dlatego też modyfikator short powoduje skrócenie zmiennej do 16 bitów.

Modyfikatory

volatile

`volatile` znaczy ulotny. Oznacza to, że kompilator wyłączy dla takiej zmiennej optymalizacje typu zastąpienia przez stałą lub zawartość rejestru, za to wygeneruje kod, który będzie odwoływał się zawsze do komórek pamięci danego obiektu. Zapobiegnie to błędowi, gdy obiekt zostaje zmieniony przez część programu, która nie ma zauważalnego dla kompilatora związku z danym fragmentem kodu lub nawet przez zupełnie inny proces.

```
volatile float liczba1;
float liczba2;
{
  printf ("%d\n%d", liczba1, liczba2);
  /* instrukcje nie związane ze zmiennymi */
  printf ("%d\n%d", liczba1, liczba2);
}
```

Jeżeli zmienne liczba1 i liczba2 zmienią się niezauważalnie dla kompilatora to odczytując :

- liczba1 — nastąpi odwołanie do komórek pamięci. Kompilator pobierze nową wartość zmiennej.
- liczba2 — kompilator może wypisać poprzednią wartość, którą przechowywał w rejestrze.

Modyfikator `volatile` jest rzadko stosowany i przydaje się w wąskich zastosowaniach, jak współbieżność i współdzielenie zasobów oraz przerwania systemowe.

register

Jeżeli utworzymy zmienną, której będziemy używać w swoim programie bardzo często, możemy wykorzystać modyfikator `register`. Kompilator może wtedy umieścić zmienną w rejestrze, do którego ma szybki dostęp, co przyśpieszy odwołania do tej zmiennej

```
register int liczba ;
```

W nowoczesnych kompilatorach ten modyfikator praktycznie nie ma wpływu na program. Optymalizator sam decyduje czy i co należy umieścić w rejestrze. Nie mamy żadnej gwarancji, że zmienna tak zadeklarowana rzeczywiście się tam znajdzie, chociaż dostęp do niej może zostać przyspieszony w inny sposób. Raczej powinno się unikać tego typu konstrukcji w programie.

static

Pozwala na zdefiniowanie zmiennej statycznej. “Statyczność” polega na zachowaniu wartości pomiędzy kolejnymi definicjami tej samej zmiennej. Jest to przede wszystkim przydatne w funkcjach. Gdy zdefiniujemy zmienną w ciele funkcji, to zmienna ta będzie od nowa definiowana wraz z domyślną wartością (jeżeli taką podano). W wypadku zmiennej określonej jako statyczna, jej wartość się nie zmieni przy ponownym wywołaniu funkcji. Na przykład:

```
void dodaj(int liczba)
{
  int zmienna = 0;
  zmienna = zmienna + liczba;
  printf ("Wartosc zmiennej %d\n", zmienna);
}
```

Gdy wywołamy tę funkcję np. 3 razy w ten sposób:

```
dodaj(3);
dodaj(5);
dodaj(4);
```

to ujrzymy na ekranie:

```
Wartość zmiennej Zmienna:3
Wartość zmiennej Zmienna:5
Wartość zmiennej Zmienna:4
```

jeżeli jednak deklarację zmiennej zmienimy na `static int zmienna = 0`, to wartość zmiennej zostanie zachowana i po podobnym wykonaniu funkcji powinnyśmy ujrzeć:

```
Wartość zmiennej Zmienna:3
Wartość zmiennej Zmienna:8
Wartość zmiennej Zmienna:12
```

Zupełnie co innego oznacza `static` zastosowane dla zmiennej globalnej. Jest ona wtedy widoczna tylko w jednym pliku. Zobacz też: rozdział Biblioteki.

extern

Przez `extern` oznacza się zmienne globalne zadeklarowane w innych plikach — informujemy w ten sposób kompilator, żeby nie szukał jej w aktualnym pliku. Zobacz też: rozdział Biblioteki.

auto

Zupełnym archaizmem jest modyfikator `auto`, który oznacza tyle, że zmienna jest lokalna. Ponieważ zmienna zadeklarowana w dowolnym bloku zawsze jest lokalna, modyfikator ten nie ma obecnie żadnego zastosowania praktycznego. `auto` jest spadkiem po wcześniejszych językach programowania, na których oparty jest C (np. B).

Uwagi

- Język C++ pozwala na mieszanie deklaracji zmiennych z kodem. Więcej informacji w C++/Zmienne.

Rozdział 8

Operatory

Przypisanie

Operator przypisania ("="), jak sama nazwa wskazuje, przypisuje wartość prawego argumentu lewemu, np.:

```
int a = 5, b;
b = a;
printf("%d\n", b); /* wypisze 5 */
```

Operator ten ma łączność prawostronną tzn. obliczanie przypisań następuje z prawa na lewo i zwraca on przypisaną wartość, dzięki czemu może być użyty kaskadowo:

```
int a, b, c;
a = b = c = 3;
printf("%d %d %d\n", a, b, c);  /* wypisze "3 3 3" */
```

Skrócony zapis

C umożliwia też skrócony zapis postaci `a #= b;`, gdzie # jest jednym z operatorów: +, -, *, /, &, —, ^, << lub >> (opisanych niżej). Ogólnie rzecz ujmując zapis `a #= b;` jest równoważny zapisowi `a = a # (b);`, np.:

```
int a = 1;
a += 5;     /* to samo, co a = a + 5;       */
a /= a + 2; /* to samo, co a = a / (a + 2); */
a %= 2;     /* to samo, co a = a % 2;       */
```

Początkowo skrócona notacja miała następującą składnię: a =# b, co często prowadziło do niejasności, np. i =-1 (i = -1 czy też i = i-1?). Dlatego też zdecydowano się zmienić kolejność operatorów.

Rzutowanie

Zadaniem rzutowania jest konwersja danej jednego typu na daną innego typu. Konwersja może być niejawna (domyślna konwersja przyjęta przez kompilator) lub jawna (podana explicite przez programistę). Oto kilka przykładów konwersji niejawnej:

```
int i = 42.7;            /* konwersja z double do int */
float f = i;             /* konwersja z int do float */
double d = f;            /* konwersja z float do double */
unsigned u = i;          /* konwersja z int do unsigned int */
f = 4.2;                 /* konwersja z double do float */
i = d;                   /* konwersja z double do int */
char *str = "foo";       /* konwersja z const char* do char*  [1] */
const char *cstr = str;  /* konwersja z char* do const char* */
void *ptr = str;         /* konwersja z char* do void* */
```

Podczas konwersji zmiennych zawierających większe ilości danych do typów prostszych (np. double do int) musimy liczyć się z utratą informacji, jak to miało miejsce w pierwszej linijce — zmienna int nie może przechowywać części ułamkowej toteż została ona *odcięta* i w rezultacie zmiennej została przypisana wartość 42.

Zaskakująca może się wydać linijka oznaczona przez 1. Niejawna konwersja z typu const char* do typu char* nie jest dopuszczana przez standard C. Jednak literały napisowe (które są typu const char*) stanowią tutaj wyjątek. Wynika on z faktu, że były one używane na długo przed wprowadzeniem słówka const do języka i brak wspomnianego wyjątku spowodowałby, że duża część kodu zostałaby nagle zakwalifikowana jako niepoprawny kod.

Do jawnego wymuszenia konwersji służy jednoargumentowy operator rzutowania, np.:

```
double d = 3.14;
int pi = (int)d;         /* 1 */
pi = (unsigned)pi >> 4;  /* 2 */
```

W pierwszym przypadku operator został użyty, by zwrócić uwagę na utratę precyzji. W drugim, dlatego że bez niego operator przesunięcia bitowego zachowuje się trochę inaczej.

Obie konwersje przedstawione powyżej są dopuszczane przez standard jako jawne konwersje (tj. konwersja z double do int oraz z int do unsigned int), jednak niektóre konwersje są błędne, np.:

```
const char *cstr = "foo";
char *str = cstr;
```

W takich sytuacjach można użyć operatora rzutowania by wymusić konwersję:

```
const char *cstr = "foo";
char *str = (char*)cstr;
```

Należy unikać jednak takich sytuacji i **nigdy** nie stosować rzutowania by *uciszyć kompilator*. Zanim użyjemy operatora rzutowania należy się zastanowić co tak naprawdę będzie on robił i czy nie ma innego sposobu wykonania danej operacji, który nie wymagałby podejmowania tak drastycznych kroków.

Operatory arytmetyczne

Język C definiuje następujące dwuargumentowe operatory arytmetyczne:

- dodawanie ("+"),
- odejmowanie ("-"),
- mnożenie ("*"),
- dzielenie ("/"),
- reszta z dzielenia ("%") określona tylko dla liczb całkowitych (tzw. *dzielenie modulo*).

```
int a=7, b=2, c;
c = a % b;
printf ("%d\n",c); /* wypisze "1" */
```

Należy pamiętać, że (w pewnym uproszczeniu) wynik operacji jest typu takiego jak *największy* z argumentów. Oznacza to, że operacja wykonana na dwóch liczbach całkowitych nadal ma typ całkowity nawet jeżeli wynik przypiszemy do zmiennej rzeczywistej. Dla przykładu, poniższy kod:

```
float a = 7 / 2;
printf("%f\n", a);
```

wypisze (wbrew oczekiwaniu początkujących programistów) `3`, a nie `3.5`. Odnosi się to nie tylko do dzielenia, ale także mnożenia, np.:

```
float a = 1000 * 1000 * 1000 * 1000 * 1000 * 1000;
printf("%f\n", a);
```

prawdopodobnie da o wiele mniejszy wynik niż byśmy się spodziewali. Aby wymusić obliczenia rzeczywiste należy zmienić typ jednego z argumentów na liczbę rzeczywistą po prostu zmieniając literał lub korzystając z rzutowania, np.:

```
float a = 7.0 / 2;
float b = (float)1000 * 1000 * 1000 * 1000 * 1000 * 1000;
printf("%f\n", a);
printf("%f\n", b);
```

Operatory dodawania i odejmowania są określone również, gdy jednym z argumentów jest wskaźnik, a drugim liczba całkowita. Ten drugi jest także określony, gdy oba argumenty są wskaźnikami. O takim użyciu tych operatorów dowiesz się więcej C/Wskaźniki—w dalszej części książki.

Inkrementacja i dekrementacja

Aby skrócić zapis wprowadzono dodatkowe operatory: inkrementacji ("++") i dekrementacji ("–"), które dodatkowo mogą być pre- lub postfiksowe. W rezultacie mamy więc cztery operatory:

- pre-inkrementacja ("++i"),

- post-inkrementacja (“i++”),
- pre-dekrementacja (“–i”) i
- post-dekrementacja (“i–”).

Operatory inkrementacji zwiększa, a dekrementacji zmniejsza argument o jeden. Ponadto operatory pre- zwracają nową wartość argumentu, natomiast post- starą wartość argumentu.

```
int a, b, c;
a = 3;
b = a--; /* po operacji b=3 a=2 */
c = --b; /* po operacji b=2 c=2 */
```

Czasami (szczególnie w C++) użycie operatorów stawianych za argumentem jest nieco mniej efektywne (bo kompilator musi stworzyć nową zmienną by przechować wartość tymczasową).

Bardzo ważne jest, abyśmy poprawnie stosowali operatory dekrementacji i inkrementacji. Chodzi o to, aby w jednej instrukcji nie umieszczać kilku operatorów, które modyfikują ten sam obiekt (zmienną). Jeżeli taka sytuacja zaistnieje, to efekt działania instrukcji jest nieokreślony. Prostym przykładem mogą być następujące instrukcje:

```
int a = 1;
a = a++;
a = ++a;
a = a++ + ++a;
printf("%d %d\n", ++a, ++a);
printf("%d %d\n", a++, a++);
```

Kompilator GCC potrafi ostrzegać przed takimi błędami - aby to czynił należy podać mu jako argument opcję `-Wsequence-point`.

Operacje bitowe

Oprócz operacji znanych z lekcji matematyki w podstawówce, język C został wyposażony także w operatory bitowe, zdefiniowane dla liczb całkowitych. Są to:

- negacja bitowa (“ ”),
- koniunkcja bitowa (“&”),
- alternatywa bitowa (“—”) i
- alternatywa rozłączna (XOR) (“^”).

Działają one na poszczególnych bitach przez co mogą być szybsze od innych operacji. Działanie tych operatorów można zdefiniować za pomocą poniższych tabel:

```
 "~" | 0 1    "&" | 0 1    "|" | 0 1    "^" | 0 1
-----+-----  -----+-----  -----+-----  -----+-----
     | 1 0      0 | 0 0      0 | 0 1      0 | 0 1
              1 | 0 1      1 | 1 1      1 | 1 0

   a   | 0101  =  5
   b   | 0011  =  3
-------+------
  ~a   | 1010  = 10
  ~b   | 1100  = 12
 a & b | 0001  =  1
 a | b | 0111  =  7
 a ^ b | 0110  =  6
```

Lub bardziej opisowo:

- negacja bitowa daje w wyniku liczbę, która ma bity równe jeden tylko na tych pozycjach, na których argument miał bity równe zero;
- koniunkcja bitowa daje w wyniku liczbę, która ma bity równe jeden tylko na tych pozycjach, na których oba argumenty miały bity równe jeden (mnemonik: 1 gdy wszystkie 1);
- alternatywa bitowa daje w wyniku liczbę, która ma bity równe jeden na wszystkich tych pozycjach, na których jeden z argumentów miał bit równy jeden (mnemonik: 1 jeśli jest 1);
- alternatywa rozłączna daje w wyniku liczbę, która ma bity równe jeden tylko na tych pozycjach, na których tylko jeden z argumentów miał bit równy jeden (mnemonik: 1 gdy różne).

Przy okazji warto zauważyć, że `a ^ b ^ b` to po prostu `a`. Właściwość ta została wykorzystana w różnych algorytmach szyfrowania oraz funkcjach haszujących. Alternatywę wyłączną stosuje się np. do szyfrowania kodu wirusów polimorficznych.

Przesunięcie bitowe

Dodatkowo, język C wyposażony jest w operatory przesunięcia bitowego w lewo (“<<”) i prawo (“>>”). Przesuwają one w danym kierunku bity lewego argumentu o liczbę pozycji podaną jako prawy argument. Brzmi to może strasznie, ale wcale takie nie jest. Rozważmy 4 bitowe liczby bez znaku (taki hipotetyczny unsigned int), wówczas:

```
  a   | a<<1 | a<<2 | a>>1 | a>>2
------+------+------+------+------
 0001 | 0010 | 0100 | 0000 | 0000
 0011 | 0110 | 1100 | 0001 | 0000
 0101 | 1010 | 0100 | 0010 | 0001
 1000 | 0000 | 0000 | 0100 | 0010
 1111 | 1110 | 1100 | 0111 | 0011
 1001 | 0010 | 0100 | 0100 | 0010
```

Nie jest to zatem takie straszne na jakie wygląda. Widać, że bity będące na skraju są tracone, a w "puste" miejsca wpisywane są zera.

Inaczej rzecz się ma jeżeli lewy argument jest liczbą ze znakiem. Dla przesunięcia bitowego w lewo `a << b` jeżeli a jest nieujemna i wartość $a \cdot 2^b$ mieści się w zakresie liczby to jest to wynikiem operacji. W przeciwnym wypadku działanie jest niezdefiniowane[1].

Dla przesunięcia bitowego w lewo, jeżeli lewy argument jest nieujemny to operacja zachowuje się tak jak w przypadku liczb bez znaku. Jeżeli jest on ujemny to zachowanie jest zależne od implementacji.

Zazwyczaj operacja przesunięcia w lewo zachowuje się tak samo jak dla liczb bez znaku, natomiast przy przesuwaniu w prawo bit znaku nie zmienia się[2]:

```
  a   | a>>1 | a>>2
------+------+------
 0001 | 0000 | 0000
 0011 | 0001 | 0000
 0101 | 0010 | 0001
 1000 | 1100 | 1110
 1111 | 1111 | 1111
 1001 | 1100 | 1110
```

Przesunięcie bitowe w lewo odpowiada pomnożeniu, natomiast przesunięcie bitowe w prawo podzieleniu liczby przez dwa do potęgi jaką wyznacza prawy argument. Jeżeli prawy argument jest ujemny lub większy lub równy liczbie bitów w typie, działanie jest niezdefiniowane.

```
#include <stdio.h>

int main ()
{
  int a = 6;
  printf ("6 << 2 = %d\n", a<<2);  /* wypisze 24 */
  printf ("6 >> 2 = %d\n", a>>2);  /* wypisze 1 */
  return 0;
}
```

Porównanie

W języku C występują następujące operatory porównania:

- równe ("=="),
- różne ("!="),
- mniejsze ("<"),
- większe (">"),
- mniejsze lub równe ("<=") i

[1] Niezdefiniowane w takim samym sensie jak niezdefiniowane jest zachowanie programu, gdy próbujemy odwołać się do wartości wskazywanej przez wartość NULL czy do zmiennych poza tablicą.

[2] ale jeżeli zależy Ci na przenośności kodu nie możesz na tym polegać

- większe lub równe (“>=”).

Wykonują one odpowiednie porównanie swoich argumentów i zwracają jedynkę jeżeli warunek jest spełniony lub zero jeżeli nie jest.

Częste błędy

Osoby, które poprzednio uczyły się innych języków programowania, często mają nawyk używania w instrukcjach logicznych zamiast operatora **porównania** ==, operatora **przypisania** =. Ma to często zgubne efekty, gdyż przypisanie zwraca wartość przypisaną lewemu argumentowi.

Porównajmy ze sobą dwa warunki:

```
(a = 1)
(a == 1)
```

Pierwszy z nich zawsze będzie prawdziwy, niezależnie od wartości zmiennej a! Dzieje się tak, ponieważ zostaje wykonane przypisanie do a wartości 1 a następnie jako wartość jest zwracane to, co zostało przypisane — czyli jeden. Drugi natomiast będzie prawdziwy tylko, gdy a jest równe 1.

W celu uniknięcia takich błędów niektórzy programiści zamiast pisać `a == 1` piszą `1 == a`, dzięki czemu pomyłka spowoduje, że kompilator zgłosi błąd.

Warto zauważyć, że kompilator GCC potrafi w pewnych sytuacjach wychwycić taki błąd. Aby zaczął to robić należy podać mu argument `-Wparentheses`.

Innym błędem jest użycie zwykłych operatorów porównania do sprawdzania relacji pomiędzy liczbami rzeczywistymi. Ponieważ operacje zmiennoprzecinkowe wykonywane są z pewnym przybliżeniem rzadko kiedy dwie zmienne typu float czy double są sobie równe. Dla przykładu:

```
#include <stdio.h>
int main () {
  float a, b, c;
  a = 1e10;   /* tj. 10 do potęgi 10 */
  b = 1e-10;  /* tj. 10 do potęgi -10 */
  c = b;      /* c = b */
  c = c + a;  /* c = b + a          (teoretycznie) */
  c = c - a;  /* c = b + a - a = b  (teoretycznie) */
  printf("%d\n", c == b); /* wypisze 0 */
}
```

Obejściem jest porównywanie modułu różnicy liczb. Również i takie błędy kompilator GCC potrafi wykrywać — aby to robił należy podać mu argument `-Wfloat-equal`.

Operatory logiczne

Analogicznie do części operatorów bitowych, w C definiuje się operatory logiczne, mianowicie:

- negacja (`!`),

- koniunkcja (`&&`) i
- alternatywa (`||`).

Działają one bardzo podobnie do operatorów bitowych jednak zamiast operować na poszczególnych bitach biorą pod uwagę wartość logiczną argumentów. Wyrażenie ma wartość logiczną 0 wtedy i tylko wtedy, gdy jest równe 0. W przeciwnym wypadku ma wartość 1. Operatory te w wyniku dają albo 0 albo 1.

Skrócone obliczanie wyrażeń logicznych

Język C wykonuje skrócone obliczanie wyrażeń logicznych — to znaczy, oblicza wyrażenie tylko tak długo, jak nie wie, jaka będzie jego ostateczna wartość. To znaczy, idzie od lewej do prawej obliczając kolejne wyrażenia (dodatkowo na kolejność wpływ mają nawiasy) i gdy będzie miał na tyle informacji, by obliczyć wartość całości, nie liczy reszty. Może to wydawać się niejasne, ale przyjrzyjmy się wyrażeniom logicznym:

```
A && B
A || B
```

Jeśli A jest fałszywe to nie trzeba liczyć B w pierwszym wyrażeniu, bo fałsz i dowolne wyrażenie zawsze da fałsz. Analogicznie, jeśli A jest prawdziwe, to wyrażenie 2 jest prawdziwe i wartość B nie ma znaczenia.

Poza zwiększoną szybkością zysk z takiego rozwiązania polega na możliwości stosowania efektów ubocznych. Idea efektu ubocznego opiera się na tym, że w wyrażeniu można wywołać funkcje, które będą robiły poza zwracaniem wyniku inne rzeczy, oraz używać podstawień. Popatrzmy na poniższy przykład:

```
( (a > 0) || (a < 0) || (a = 1) )
```

Jeśli a będzie większe od 0 to obliczona zostanie tylko wartość wyrażenia `(a > 0)` — da ono prawdę, czyli reszta obliczeń nie będzie potrzebna. Jeśli a będzie mniejsze od zera, najpierw zostanie obliczone pierwsze podwyrażenie a następnie drugie, które da prawdę. Ciekawy będzie jednak przypadek, gdy a będzie równe zero — do a zostanie wtedy podstawiona jedynka i całość wyrażenia zwróci prawdę (bo 1 jest traktowane jak prawda).

Efekty uboczne pozwalają na różne szaleństwa i wykonywanie złożonych operacji w samych warunkach logicznych, jednak przesadne używanie tego typu konstrukcji powoduje, że kod staje się nieczytelny i jest uważane za zły styl programistyczny.

Operator wyrażenia warunkowego

C posiada szczególny rodzaj operatora — to operator `?:` zwany też operatorem wyrażenia warunkowego. Jest to jedyny operator w tym języku przyjmujący trzy argumenty.

```
a ? b : c
```

Jego działanie wygląda następująco: najpierw oceniana jest wartość logiczna wyrażenia `a`; jeśli jest ono prawdziwe, to zwracana jest wartość `b`, jeśli natomiast wyrażenie `a` jest nieprawdziwe, zwracana jest wartość `c`.

Praktyczne zastosowanie — znajdywanie większej z dwóch liczb:

```
a = (b>=c) ? b : c;     /* Jeśli b jest większe bądź równe c, to zwróć b.
                           W przeciwnym wypadku zwróć c. */
```

lub zwracanie modułu liczby:

```
a = a < 0 ? -a : a;
```

Wartości wyrażeń są przy tym operatorze obliczane tylko jeżeli zachodzi taka potrzeba, np. w wyrażeniu `1 ? 1 : foo()` funkcja foo() nie zostanie wywołana.

Operator przecinek

Operator przecinek jest dość dziwnym operatorem. Powoduje on obliczanie wartości wyrażeń od lewej do prawej po czym zwrócenie wartości ostatniego wyrażenia. W zasadzie, w normalnym kodzie programu ma on niewielkie zastosowanie, gdyż zamiast niego lepiej rozdzielać instrukcje zwykłymi średnikami. Ma on jednak zastosowanie w instrukcji sterującej for.

Operator sizeof

Operator sizeof zwraca rozmiar w bajtach (gdzie bajtem jest zmienna typu char) podanego typu lub typu podanego wyrażenia. Ma on dwa rodzaje: `sizeof(typ)` lub `sizeof wyrażenie`. Przykładowo:

```
#include <stdio.h>

int main()
{
  printf("sizeof(short ) = %d\n", sizeof(short ));
  printf("sizeof(int   ) = %d\n", sizeof(int   ));
  printf("sizeof(long  ) = %d\n", sizeof(long  ));
  printf("sizeof(float ) = %d\n", sizeof(float ));
  printf("sizeof(double) = %d\n", sizeof(double));
  return 0;
}
```

Operator ten jest często wykorzystywany przy dynamicznej alokacji pamięci, co zostanie opisane w rozdziale poświęconym wskaźnikom.

Pomimo, że w swej budowie operator **sizeof** bardzo przypomina funkcję, to jednak nią nie jest. Wynika to z trudności w implementacji takowej funkcji — jej specyfika musiałaby odnosić się bezpośrednio do kompilatora. Ponadto jej argumentem musiałyby być typy, a nie zmienne. W języku C nie jest możliwe przekazywanie typu jako argumentu. Ponadto często zdarza się, że rozmiar zmiennej musi być wiadomy jeszcze w czasie kompilacji — to ewidentnie wyklucza implementację sizeof() jako funkcji.

Inne operatory

Poza wyżej opisanymi operatorami istnieją jeszcze:

- operator “” opisany przy okazji opisywania tablic;

- jednoargumentowe operatory "*" i "&" opisane przy okazji opisywania wskaźników;
- operatory "." i "->" opisywane przy okazji opisywania struktur i unii;
- operator "()" będący operatorem wywołania funkcji,
- operator "()" grupujący wyrażenia (np. w celu zmiany kolejności obliczania

Priorytety i kolejność obliczeń

Jak w matematyce, również i w języku C obowiązuje pewna ustalona kolejność działań. Aby móc ją określić należy ustalić dwa parametry danego operatora: jego priorytet oraz łączność. Przykładowo operator mnożenia ma wyższy priorytet niż operator dodawania i z tego powodu w wyrażeniu $2+2\cdot 2$ najpierw wykonuje się mnożenie, a dopiero potem dodawanie.

Drugim parametrem jest łączność — określa ona *od której strony* wykonywane są działania w przypadku połączenia operatorów o tym samym priorytecie. Na przykład odejmowanie ma łączność lewostronną i $2-2-2$ da w wyniku -2. Gdyby miało łączność prawostronną w wynikiem byłoby 2. Przykładem matematycznego operatora, który ma łączność prawostronną jest potęgowanie, np. 3^{2^2} jest równe 81.

W języku C występuje dużo poziomów operatorów. Poniżej przedstawiamy tabelkę ze wszystkimi operatorami poczynając od tych z najwyższym priorytetem (wykonywanych na początku).

Tabela 8.1: Priorytety operatorów

Operator	**Łączność**
nawiasy	nie dotyczy
jednoargumentowe przyrostkowe: . -> *wołanie funkcji postinkrementacja postdekrementacja*	lewostronna
jednoargumentowe przedrostkowe:! + — * & sizeof *preinkrementacja predekrementacja rzutowanie*	prawostronna
* / %	lewostronna
+	lewostronna
<< >>	lewostronna
< <= > >=	lewostronna
== !=	lewostronna
&	lewostronna
^	lewostronna
—	lewostronna
&&	lewostronna
||	lewostronna
?:	prawostronna
operatory przypisania	prawostronna
,	lewostronna

Duża liczba poziomów pozwala czasami zaoszczędzić trochę milisekund w trakcie pisania programu i bajtów na dysku, gdyż często nawiasy nie są potrzebne, nie należy

jednak z tym przesadzać, gdyż kod programu może stać się mylący nie tylko dla innych, ale po latach (czy nawet i dniach) również dla nas.

Warto także podkreślić, że operator koniunkcji ma niższy priorytet niż operator porównania[3]. Oznacza to, że kod

```
if (flags & FL_MASK == FL_FOO)
```

zazwyczaj da rezultat inny od oczekiwanego. Najpierw bowiem wykona się porównanie wartości FL_MASK z wartością FL_FOO, a dopiero potem koniunkcja bitowa. W takich sytuacjach należy pamiętać o użyciu nawiasów:

```
if ((flags & FL_MASK) == FL_FOO)
```

Kompilator GCC potrafi wykrywać takie błędy i aby to robił należy podać mu argument `-Wparentheses`.

Kolejność wyliczania argumentów operatora

W przypadku większości operatorów (wyjątkami są tu &&, ||i przecinek) nie da się określić, która wartość argumentu zostanie obliczona najpierw. W większości przypadków nie ma to większego znaczenia, lecz w przypadku wyrażeń, które mają efekty uboczne wymuszenie konkretnej kolejności może być potrzebne. Weźmy dla przykładu program

```
#include <stdio.h>

int foo(int a) {
  printf("%d\n", a);
  return 0;
}

int main(void) {
  return foo(1) + foo(2);
}
```

Otóż, nie wiemy czy najpierw zostanie wywołana funkcja foo z parametrem jeden, czy dwa. Jeżeli ma to znaczenie należy użyć zmiennych pomocniczych zmieniając definicję funkcji main na:

```
int main(void) {
  int tmp = foo(1);
  return tmp + foo(2);
}
```

Teraz już na pewno najpierw zostanie wypisana jedynka, a potem dopiero dwójka. Sytuacja jeszcze bardziej się komplikuje, gdy używamy wyrażeń z efektami ubocznymi jako argumentów funkcji, np.:

[3] Jest to zaszłość historyczna z czasów, gdy nie było logicznych operatorów && oraz ||i zamiast nich stosowano operatory bitowe & oraz |.

```
#include <stdio.h>

int foo(int a) {
  printf("%d\n", a);
  return 0;
}

int bar(int a, int b, int c, int d) {
  return a + b + c + d;
}

int main(void) {
  return foo(1) + foo(2) + foo(3) + foo(4);
}
```

Teraz też nie wiemy, która z 24 permutacji liczb 1, 2, 3 i 4 zostanie wypisana i ponownie należy pomóc sobie zmiennymi tymczasowymi jeżeli zależy nam na konkretnej kolejności:

```
int main(void) {
  int tmp = foo(1);
  tmp += foo(2);
  tmp += foo(3);
  return tmp + foo(4);
}
```

Uwagi

- W języku C++ wprowadzony został dodatkowo inny sposób zapisu rzutowania, który pozwala na łatwiejsze znalezienie w kodzie miejsc, w których dokonujemy rzutowania. Więcej na stronie C++/Zmienne.

Zobacz też

- C/Składnia#Operatory

Rozdział 9

Instrukcje sterujące

C jest językiem imperatywnym — oznacza to, że instrukcje wykonują się jedna po drugiej w takiej kolejności w jakiej są napisane. Aby móc zmienić kolejność wykonywania instrukcji potrzebne są instrukcje sterujące.

Na wstępie przypomnijmy jeszcze, że wyrażenie jest prawdziwe wtedy i tylko wtedy, gdy jest różne od zera, a fałszywe wtedy i tylko wtedy, gdy jest równe zeru.

Instrukcje warunkowe

Instrukcja if

Użycie instrukcji if wygląda tak:

```
if (wyrażenie) {
  /* blok wykonany, jeśli wyrażenie jest prawdziwe */
}
/* dalsze instrukcje */
```

Istnieje także możliwość reakcji na nieprawdziwość wyrażenia — wtedy należy zastosować słowo kluczowe **else**:

```
if (wyrażenie) {
  /* blok wykonany, jeśli wyrażenie jest prawdziwe */
} else {
  /* blok wykonany, jeśli wyrażenie jest nieprawdziwe */
}
/* dalsze instrukcje */
```

Przypatrzmy się bardziej "życiowemu" programowi, który porównuje ze sobą dwie liczby:

```
#include <stdio.h>

int main ()
{
  int a, b;
  a = 4;
```

```
  b = 6;
  if (a==b) {
    printf ("a jest równe b\n");
  } else {
    printf ("a nie jest równe b\n");
  }
  return 0;
}
```

Czasami zamiast pisać instrukcję if możemy użyć operatora wyboru (patrz Operatory):

```
if (a != 0)
  b = 1/a;
else
  b = 0;
```

ma dokładnie taki sam efekt jak:

```
b = (a !=0) ? 1/a : 0;
```

Instrukcja switch

Aby ograniczyć wielokrotne stosowanie instrukcji if możemy użyć **switch**. Jej użycie wygląda tak:

```
switch (wyrażenie) {
  case wartość1: /* instrukcje, jeśli wyrażenie == wartość1 */
    break;
  case wartość2: /* instrukcje, jeśli wyrażenie == wartość2 */
    break;
  /* ... */
  default: /* instrukcje, jeśli żaden z wcześniejszych warunków */
    break; /*  nie został spełniony */
}
```

Należy pamiętać o użyciu `break` po zakończeniu listy instrukcji następujących po `case`. Jeśli tego nie zrobimy, program przejdzie do wykonywania instrukcji z następnego `case`. Może mieć to fatalne skutki:

```
#include <stdio.h>

int main ()
{
  int a, b;
  printf ("Podaj a: ");
  scanf ("%d", &a);
  printf ("Podaj b: ");
  scanf ("%d", &b);
  switch (b) {
    case  0: printf ("Nie można dzielić przez 0!\n"); /* tutaj zabrakło break! */
    default: printf ("a/b=%d\n", a/b);
```

```
  }
  return 0;
}
```

A czasami może być celowym zabiegiem (tzw. "fall-through") — wówczas warto zaznaczyć to w komentarzu. Oto przykład:

```
#include <stdio.h>

int main ()
{
  int a = 4;
  switch ((a%3)) {
    case  0:
      printf ("Liczba %d dzieli się przez 3\n", a);
      break;
    case -2:
    case -1:
    case  1:
    case  2:
      printf ("Liczba %d nie dzieli się przez 3\n", a);
      break;
  }
  return 0;
}
```

Przeanalizujmy teraz działający przykład:

```
#include <stdio.h>

int main ()
{
  unsigned int dzieci = 3, podatek=1000;
  switch (dzieci) {
     case  0: break; /* brak dzieci - czyli brak ulgi */
     case  1: /* ulga  2% */
       podatek = podatek - (podatek/100* 2);
       break;
     case  2: /* ulga  5% */
       podatek = podatek - (podatek/100* 5);
       break;
     default: /* ulga 10% */
       podatek = podatek - (podatek/100*10);
       break;
  }
  printf ("Do zapłaty: %d\n", podatek);
}
```

Pętle

Instrukcja while

Często zdarza się, że nasz program musi wielokrotnie powtarzać ten sam ciąg instrukcji. Aby nie przepisywać wiele razy tego samego kodu można skorzystać z tzw. **pętli**. Pętla wykonuje się dotąd, dopóki prawdziwy jest warunek.

```
while (warunek) {
  /* instrukcje do wykonania w pętli */
}
/* dalsze instrukcje */
```

Całą zasadę pętli zrozumiemy lepiej na jakimś działającym przykładzie. Załóżmy, że mamy obliczyć kwadraty liczb od 1 do 10. Piszemy zatem program:

```
#include <stdio.h>

int main ()
{
  int a = 1;
  while (a <= 10) { /* dopóki a nie przekracza 10 */
    printf ("%d\n", a*a); /* wypisz a*a na ekran*/
    ++a; /* zwiększamy a o jeden*/
  }
  return 0;
}
```

Po analizie kodu mogą nasunąć się dwa pytania:

- Po co zwiększać wartość a o jeden? Otóż gdybyśmy nie dodali instrukcji zwiększającej a, to warunek zawsze byłby spełniony, a pętla "kręciła" by się w nieskończoność.
- Dlaczego warunek to "a <= 10" a nie "a!=10"? Odpowiedź jest dość prosta. Pętla sprawdza warunek przed wykonaniem kolejnego "obrotu". Dlatego też gdyby warunek brzmiał "a!=10" to dla a=10 jest on nieprawdziwy i pętla nie wykonałaby ostatniej iteracji, przez co program generowałby kwadraty liczb od 1 do 9, a nie do 10.

Instrukcja for

Od instrukcji while czasami wygodniejsza jest instrukcja **for**. Umożliwia ona wpisanie ustawiania zmiennej, sprawdzania warunku i inkrementowania zmiennej w jednej linijce co często zwiększa czytelność kodu. Instrukcję for stosuję się w następujący sposób:

```
for (wyrażenie1; wyrażenie2; wyrażenie3) {
  /* instrukcje do wykonania w pętli */
}
/* dalsze instrukcje */
```

Jak widać, pętla for znacznie różni się od tego typu pętli, znanych w innych językach programowania. Opiszemy więc, co oznaczają poszczególne wyrażenia:

- wyrażenie1 — jest to instrukcja, która będzie wykonana przed pierwszym przebiegiem pętli. Zwykle jest to inicjalizacja zmiennej, która będzie służyła jako "licznik" przebiegów pętli.
- wyrażenie2 — jest warunkiem zakończenia pętli. Pętla wykonuje się tak długo, jak prawdziwy jest ten warunek.
- wyrażenie3 — jest to instrukcja, która wykonywana będzie **po** każdym przejściu pętli. Zamieszczone są tu instrukcje, które zwiększają licznik o odpowiednią wartość.

Jeżeli wewnątrz pętli nie ma żadnych instrukcji **continue** (opisanych niżej) to jest ona równoważna z:

```
{
  wyrażenie1;
  while (wyrażenie2) {
    /* instrukcje do wykonania w pętli */
    wyrażenie3;
  }
}
/* dalsze instrukcje */
```

Ważną rzeczą jest tutaj to, żeby zrozumieć i zapamiętać jak tak naprawdę działa pętla for. Początkującym programistom nieznajomość tego faktu sprawia wiele problemów.

W pierwszej kolejności w pętli for wykonuje się `wyrażenie1`. Wykonuje się ono **zawsze**, nawet jeżeli warunek przebiegu pętli jest od samego początku fałszywy. Po wykonaniu `wyrażenie1` pętla for sprawdza warunek zawarty w `wyrażenie2`, jeżeli jest on prawdziwy, to wykonywana jest treść pętli for, czyli najczęściej to co znajduje się między klamrami, lub gdy ich nie ma, następna pojedyncza instrukcja. W szczególności musimy pamiętać, że sam średnik też jest instrukcją — instrukcją pustą. Gdy już zostanie wykonana treść pętli for, następuje wykonanie `wyrazenie3`. Należy zapamiętać, że wyrażenie3 zostanie wykonane, nawet jeżeli był to już ostatni obieg pętli. Poniższe 3 przykłady pętli for w rezultacie dadzą ten sam wynik. Wypiszą na ekran liczby od 1 do 10.

```
for(i=1; i<=10; ++i){
 printf("%d", i);
}
```

```
for(i=1; i<=10; ++i)
 printf("%d", i);
```

```
for(i=1; i<=10; printf("%d", ++i ) );
```

Dwa pierwsze przykłady korzystają z własności struktury blokowej, kolejny przykład jest już bardziej wyrafinowany i korzysta z tego, że jako `wyrażenie3` może zostać podane dowolne bardziej skomplikowane wyrażenie, zawierające w sobie inne podwyrażenia. A oto kolejny program, który najpierw wyświetla liczby w kolejności rosnącej, a następnie wraca.

```
#include <stdio.h>
int main()
{
 int i;
 for(i=1; i<=5; ++i){
   printf("%d", i);
 }

 for( ; i>=1; i--){
   printf("%d", i);
 }

  return 0;
}
```

Po analizie powyższego kodu, początkujący programista może stwierdzić, że pętla wypisze `123454321`. Stanie się natomiast inaczej. Wynikiem działania powyższego programu będzie ciąg cyfr `12345654321`. Pierwsza pętla wypisze cyfry "12345", lecz po ostatnim swoim obiegu pętla for (tak jak zwykle) zinkrementuje zmienną `i`. Gdy druga pętla przystąpi do pracy, zacznie ona odliczać począwszy od liczby i=6, a nie 5. By spowodować wyświetlanie liczb o 1 do 5 i z powrotem wystarczy gdzieś między ostatnim obiegiem pierwszej pętli for a pierwszym obiegiem drugiej pętli for zmniejszyć wartość zmiennej `i` o 1.

Niech podsumowaniem będzie jakiś działający fragment kodu, który może obliczać wartości kwadratów liczb od 1 do 10.

```
#include <stdio.h>

int main ()
{
  int a;
  for (a=1; a<=10; ++a) {
    printf ("%d\n", a*a);
  }
  return 0;
}
```

W kodzie źródłowym spotyka się często inkrementację `i++`. Jest to **zły zwyczaj**, biorący się z wzorowania się na nazwie języka C++. Post-inkrementacja `i++` powoduje, że tworzony jest obiekt tymczasowy, który jest zwracany jako wynik operacji (choć wynik ten nie jest nigdzie czytany). Jedno kopiowanie liczby do zmiennej tymczasowej nie jest drogie, ale w pętli "for" takie kopiowanie odbywa się po każdym przebiegu pętli. Dodatkowo, w C++ podobną konstrukcję stosuje się do obiektów — kopiowanie obiektu może być już czasochłonną czynnością. Dlatego w pętli "for" należy stosować wyłącznie `++i`.

Instrukcja do..while

Pętle while i for mają jeden zasadniczy mankament — może się zdarzyć, że nie wykonają się ani raz. Aby mieć pewność, że nasza pętla będzie miała co najmniej jeden przebieg musimy zastosować pętlę do while. Wygląda ona następująco:

```
do {
  /* instrukcje do wykonania w pętli */
} while (warunek);
/* dalsze instrukcje */
```

Zasadniczą różnicą pętli do while jest fakt, iż sprawdza ona warunek pod koniec swojego przebiegu. To właśnie ta cecha decyduje o tym, że pętla wykona się co najmniej raz. A teraz przykład działającego kodu, który tym razem będzie obliczał trzecią potęgę liczb od 1 do 10.

```
#include <stdio.h>

int main ()
{
  int a = 1;
  do {
    printf ("%d\n", a*a*a);
    ++a;
  } while (a <= 10);
  return 0;
}
```

Może się to wydać zaskakujące, ale również przy tej pętli zamiast bloku instrukcji można zastosować pojedynczą instrukcję, np.:

```
#include <stdio.h>

int main ()
{
  int a = 1;
  do printf ("%d\n", a*a*a); while (++a <= 10);
  return 0;
}
```

Instrukcja break

Instrukcja **break** pozwala na opuszczenie wykonywania pętli w dowolnym momencie. Przykład użycia:

```
int a;
for (a=1 ; a != 9 ; ++a) {
  if (a == 5) break;
  printf ("%d\n", a);
}
```

Program wykona tylko 4 przebiegi pętli, gdyż przy 5 przebiegu instrukcja break spowoduje wyjście z pętli.

Break i pętle nieskończone

W przypadku pętli for nie trzeba podawać warunku. W takim przypadku kompilator przyjmie, że warunek jest stale spełniony. Oznacza to, że poniższe pętle są równoważne:

```
for (;;) { /* ... */ }
for (;1;) { /* ... */ }
for (a;a;a) { /* ... */} /*gdzie a jest dowolną liczba rzeczywistą różną od 0*/
while (1) { /* ... */ }
do { /* ... */ } while (1);
```

Takie pętle nazywamy **pętlami nieskończonymi**, które przerwać może jedynie instrukcja **break**[1](z racji tego, że warunek pętli zawsze jest prawdziwy) [2].

Wszystkie fragmenty kodu działają identycznie:

```
int i = 0;
for (;i!=5;++i) {
  /* kod ... */
}
```

```
int i = 0;
for (;;++i) {
  if (i == 5) break;
}
```

```
int i = 0;
for (;;) {
  if (i == 5) break;
  ++i;
}
```

Instrukcja continue

W przeciwieństwie do break, która przerywa wykonywanie pętli instrukcja **continue** powoduje przejście do następnej iteracji, o ile tylko warunek pętli jest spełniony. Przykład:

```
int i;
for (i = 0 ; i < 100 ; ++i) {
  printf ("Poczatek\n");
  if (i > 40) continue ;
  printf ("Koniec\n");
}
```

[1]Tak naprawdę podobną operacje, możemy wykonać za pomocą polecenia `goto`. W praktyce jednak stosuje się zasadę, że `break` stosuje się do przerwania działania pętli i wyjścia z niej, `goto` stosuje się natomiast wtedy, kiedy chce się wydostać się z kilku zagnieżdżonych pętli za jednym zamachem. Do przerwania pracy pętli mogą nam jeszcze posłużyć polecenia `exit()` lub `return`, ale wówczas zakończymy nie tylko działanie pętli, ale i całego programu/funkcji.

[2]Żartobliwie można powiedzieć, że stosując pętlę nieskończoną to najlepiej korzystać z pętli `for(;;)`\\, gdyż wymaga ona napisania najmniejszej liczby znaków w porównaniu do innych konstrukcji.

Dla wartości i większej od 40 nie będzie wyświetlany komunikat “Koniec”. Pętla wykona pełne 100 przejść.

Oto praktyczny przykład użycia tej instrukcji:

```
#include <stdio.h>
int main()
{
  int i;
  for (i = 1 ; i <= 50 ; ++i) {
    if (i%4==0) continue ;
    printf ("%d, ", i);
  }
  return 0;
}
```

Powyższy program generuje liczby z zakresu od 1 do 50, które nie są podzielne przez 4.

Instrukcja goto

Istnieje także instrukcja, która dokonuje skoku do dowolnego miejsca programu, oznaczonego tzw. **etykietą**.

```
etykieta:
/* instrukcje */
goto etykieta;
```

Uwaga!: kompilator GCC w wersji 4.0 i wyższych jest bardzo uczulony na etykiety zamieszczone przed nawiasem klamrowym, zamykającym blok instrukcji. Innymi słowy: niedopuszczalne jest umieszczanie etykiety zaraz przed klamrą, która kończy blok instrukcji, zawartych np. w pętli for. Można natomiast stosować etykietę przed klamrą kończącą daną funkcję.

Instrukcja **goto** łamie sekwencję instrukcji i powoduje skok do dowolnie odległego miejsca w programie - co może mieć nieprzewidziane skutki. Zbyt częste używanie **goto** może prowadzić do trudnych do zlokalizowania błędów. Oprócz tego kompilatory mają kłopoty z optymalizacją kodu, w którym występują skoki. Z tego powodu zaleca się ograniczenie zastosowania tej instrukcji wyłącznie do opuszczania wielokrotnie zagnieżdżonych pętli.

Przykład uzasadnionego użycia:

```
int i,j;
for (i = 0; i < 10; ++i) {
  for (j = i; j < i+10; ++j) {
    if (i + j % 21 == 0) goto koniec;
  }
}
koniec:
/* dalsza czesc programu */
```

Natychmiastowe kończenie programu — funkcja exit

Program może zostać w każdej chwili zakończony — do tego właśnie celu służy funkcja **exit**. Używamy jej następująco:

```
exit (kod_wyjścia);
```

Liczba całkowita *kod_wyjścia* jest przekazywana do procesu macierzystego, dzięki czemu dostaje on informację, czy program w którym wywołaliśmy tą funkcję zakończył się poprawnie lub czy się tak nie stało. Kody wyjścia są nieustandaryzowane i żeby program był w pełni przenośny należy stosować makra `EXIT_SUCCESS` i `EXIT_FAILURE`, choć na wielu systemach kod 0 oznacza poprawne zakończenie, a kod różny od 0 błędne. W każdym przypadku, jeżeli nasz program potrafi generować wiele różnych kodów, warto je wszystkie udokumentować w ew. dokumentacji. Są one też czasem pomocne przy wykrywaniu błędów.

Uwagi

- W języku C++ można deklarować zmienne w nagłówku pętli "for" w następujący sposób: `for(int i=0; i<10; ++i)` (więcej informacji w C++/Zmienne)

Rozdział 10

Podstawowe procedury wejścia i wyjścia

Wejście/wyjście

Komputer byłby całkowicie bezużyteczny, gdyby użytkownik nie mógł się z nim porozumieć (tj. wprowadzić danych lub otrzymać wyników pracy programu). Programy komputerowe służą w największym uproszczeniu do obróbki danych — więc muszą te dane jakoś od nas otrzymać, przetworzyć i przekazać nam wynik.

Takie wczytywanie i "wyrzucanie" danych w terminologii komputerowej nazywamy **wejściem** (**input**) i **wyjściem** (**output**). Bardzo często mówi się o wejściu i wyjściu danych łącznie — *input/output*, albo po prostu **I/O**.

W C do komunikacji z użytkownikiem służą odpowiednie **funkcje**. Zresztą, do wielu zadań w C służą funkcje. Używając funkcji, nie musimy wiedzieć, w jaki sposób komputer wykonuje jakieś zadanie, interesuje nas tylko to, co ta funkcja robi. Funkcje niejako "wykonują za nas część pracy", ponieważ nie musimy pisać być może dziesiątek linijek kodu, żeby np. wypisać tekst na ekranie (wbrew pozorom — kod funkcji wyświetlającej tekst na ekranie jest dość skomplikowany). Jeszcze taka uwaga — gdy piszemy o jakiejś funkcji, zazwyczaj podając jej nazwę dopisujemy na końcu nawias:

```
printf()
scanf()
```

żeby było jasne, że chodzi o funkcję, a nie o coś innego.

Wyżej wymienione funkcje to jedne z najczęściej używanych funkcji w C — pierwsza służy do wypisywania danych na ekran, natomiast druga do wczytywania danych z klawiatury[1].

[1] W zasadzie standard C nie definiuje czegoś takiego jak ekran i klawiatura — mowa w nim o *standardowym wyjściu* i *standardowym wejściu*. Zazwyczaj jest to właśnie ekran i klawiatura, ale nie zawsze. W szczególności użytkownicy Linuksa lub innych systemów uniksowych mogą być przyzwyczajeniu do przekierowania wejścia/wyjścia z/do pliku czy łączenie komend w potoki (ang. pipe). W takich sytuacjach dane nie są wyświetlane na ekranie, ani odczytywane z klawiatury.

Funkcje wyjścia

Funkcja printf

W przykładzie "Hello World!" użyliśmy już jednej z dostępnych funkcji wyjścia, a mianowicie funkcji printf(). Z punktu widzenia swoich możliwości jest to jedna z bardziej skomplikowanych funkcji, a jednocześnie jest jedną z najczęściej używanych. Przyjrzyjmy się ponownie kodowi programu "Hello, World!".

```
#include <stdio.h>

int main(void)
{
  printf("Hello world!\n");
  return 0;
}
```

Po skompilowaniu i uruchomieniu, program wypisze na ekranie:

```
Hello world!
```

W naszym przykładowym programie, chcąc by funkcja `printf()` wypisała tekst na ekranie, umieściliśmy go w cudzysłowach wewnątrz nawiasów. Ogólnie, wywołanie funkcji `printf()` wygląda następująco:

```
printf(format, argument1, argument2, ...);
```

Przykładowo:

```
int i = 500;
printf("Liczbami całkowitymi są na przykład %i oraz %i.\n", 1, i);
```

wypisze

```
Liczbami całkowitymi są na przykład 1 oraz 500.
```

Format to napis ujęty w cudzysłowy, który określa ogólny kształt, schemat tego, co ma być wyświetlone. Format jest drukowany tak, jak go napiszemy, jednak niektóre znaki specjalne zostaną w nim podmienione na co innego. Przykładowo, znak specjalny `\n` jest zamieniany na znak nowej linii [2]. Natomiast procent jest podmieniany na jeden z argumentów. Po procencie następuje specyfikacja, jak wyświetlić dany argument. W tym przykładzie `%i` (od **i**nt) oznacza, że argument ma być wyświetlony jak liczba całkowita. W związku z tym, że \ i `%` mają specjalne znaczenie, aby wydrukować je, należy użyć ich podwójnie:

```
printf("Procent: %% Backslash: \\");
```

drukuje:

```
Procent: % Backslash: \
```

[2] Zmiana ta następuje w momencie kompilacji programu i dotyczy wszystkich literałów napisowych. Nie jest to jakaś szczególna własność funkcji printf(). Więcej o tego typu sekwencjach i ciągach znaków w szczególności opisane jest w rozdziale Napisy.

(bez przejścia do nowej linii). Na liście argumentów możemy mieszać ze sobą zmienne różnych typów, liczby, napisy itp. w dowolnej liczbie. Funkcja printf przyjmie ich tyle, ile tylko napiszemy. Należy uważać, by nie pomylić się w formatowaniu:

```
int i = 5;
printf("%i %s %i", 5, 4, "napis"); /* powinno być: "%i %i %s" */
```

Przy włączeniu ostrzeżeń (opcja `-Wall` lub `-Wformat` w GCC) kompilator powinien nas ostrzec, gdy format nie odpowiada podanym elementom.

Najczęstsze użycie printf():

- `printf(%i, i);` gdy `i` jest typu `int`; zamiast `%i` można użyć `%d`
- `printf(%f, i);` gdy `i` jest typu `float` lub `double`
- `printf(%c, i);` gdy `i` jest typu `char` (i chcemy wydrukować znak)
- `printf(%s, i);` gdy `i` jest napisem (typu `char*`)

Funkcja printf() nie jest żadną specjalną konstrukcją języka i łańcuch formatujący może być podany jako zmienna. W związku z tym możliwa jest np. taka konstrukcja:

```
#include <stdio.h>

int main(void)
{
  char buf[100];
  scanf("%99s", buf); /* funkcja wczytuje tekst do tablicy buf */
  printf(buf);
  return 0;
}
```

Program wczytuje tekst, a następnie wypisuje go. Jednak ponieważ znak procentu jest traktowany w specjalny sposób, toteż jeżeli na wejściu pojawi się ciąg znaków zawierający ten znak mogą się stać różne dziwne rzeczy. Między innymi z tego powodu w takich sytuacjach lepiej używać funkcji puts() lub fputs() opisanych niżej lub wywołania: `printf(%s, zmienna);`.

Więcej o funkcji printf()

Funkcja puts

Funkcja puts() przyjmuje jako swój argument ciąg znaków, który następnie bezmyślnie wypisuje na ekran kończąc go znakiem przejścia do nowej linii. W ten sposób, nasz pierwszy program mogli byśmy napisać w ten sposób:

```
#include <stdio.h>

int main(void)
{
  puts("Hello world!");
  return 0;
}
```

W swoim działaniu funkcja ta jest w zasadzie identyczna do wywołania: `printf(%s\n, argument);` jednak prawdopodobnie będzie działać szybciej. Jedynym jej mankamentem może być fakt, że zawsze na końcu podawany jest znak przejścia do nowej linii. Jeżeli jest to efekt niepożądany (nie zawsze tak jest) należy skorzystać z funkcji fputs() opisanej niżej lub wywołania `printf(%s, argument);`.

Więcej o funkcji puts()

Funkcja fputs

Opisując funkcję fputs() wybiegamy już trochę w przyszłość (a konkretnie do opisu operacji na plikach), ale warto o niej wspomnieć już teraz, gdyż umożliwia ona wypisanie swojego argumentu bez wypisania na końcu znaku przejścia do nowej linii:

```
#include <stdio.h>

int main(void)
{
  fputs("Hello world!\n", stdout);
  return 0;
}
```

W chwili obecnej możesz się nie przejmować tym zagadkowym stdout wpisanym jako drugi argument funkcji. Jest to określenie strumienia wyjściowego (w naszym wypadku standardowe wyjście — **st**an**d**ard **out**put).

Więcej o funkcji fputs()

Funkcja putchar

Funkcja putchar() służy do wypisywania pojedynczych znaków. Przykładowo jeżeli chcielibyśmy napisać program wypisujący w prostej tabelce wszystkie liczby od 0 do 99 moglibyśmy to zrobić tak:

```
#include <stdio.h>

int main(void) {
  int i = 0;
  for (; i<100; ++i) {
    /* Nie jest to pierwsza liczba w wierszu */
    if (i % 10) {
      putchar(' ');
    }
    printf("%2d", i);
    /* Jest to ostatnia liczba w wierszu */
    if ((i % 10)==9) {
      putchar('\n');
    }
  }
  return 0;
}
```

Więcej o funkcji putchar()

Funkcje wejścia

Funkcja scanf()

Teraz pomyślmy o sytuacji odwrotnej. Tym razem to użytkownik musi powiedzieć coś programowi. W poniższym przykładzie program podaje kwadrat liczby, podanej przez użytkownika:

```
#include <stdio.h>

int main ()
{
  int liczba = 0;
  printf ("Podaj liczbę: ");
  scanf ("%d", &liczba);
  printf ("%d*%d=%d\n", liczba, liczba, liczba*liczba);
  return 0;
}
```

Zauważyłeś, że w tej funkcji przy zmiennej pojawił się nowy operator — **&** (etka). Jest on ważny, gdyż bez niego funkcja scanf() nie skopiuje odczytanej wartości liczby do odpowiedniej zmiennej! Właściwie oznacza przekazanie do funkcji adresu zmiennej, by funkcja mogła zmienić jej wartość. Nie musisz teraz rozumieć, jak to się odbywa, wszystko zostanie wyjaśnione w rozdziale Wskaźniki.

Oznaczenia są podobne takie jak przy printf(), czyli `scanf(%i, &liczba);` wczytuje liczbę typu `int`, `scanf(%f, &liczba);` – liczbę typu `float`, a `scanf(%s, tablica_znaków);` ciąg znaków. Ale czemu w tym ostatnim przypadku nie ma etki? Otóż, gdy podajemy jako argument do funkcji wyrażenie typu tablicowego zamieniane jest ono automatycznie na adres pierwszego elementu tablicy. Będzie to dokładniej opisane w rozdziale poświęconym wskaźnikom.

Brak etki jest częstym błędem szczególnie wśród początkujących programistów. Ponieważ funkcja scanf() akceptuje zmienną liczbę argumentów to nawet kompilator może mieć kłopoty z wychwyceniem takich błędów (konkretnie chodzi o to, że standard nie wymaga od kompilatora wykrywania takich pomyłek), choć kompilator GCC radzi sobie z tym jeżeli podamy mu argument `-Wformat`.

Należy jednak uważać na to ostatnie użycie. Rozważmy na przykład poniższy kod:

```
#include <stdio.h>

int main(void)
{
  char tablica[100];     /* 1 */
  scanf("%s", tablica);  /* 2 */
  return 0;
}
```

Robi on niewiele. W linijce 1 deklarujemy tablicę 100 znaków czyli mogącą przechować napis długości 99 znaków. Nie przejmuj się jeżeli nie do końca to wszystko

rozumiesz — pojęcia takie jak tablica czy ciąg znaków staną się dla Ciebie jasne w miarę czytania kolejnych rozdziałów. W linijce 2 wywołujemy funkcję scanf(), która odczytuje tekst ze standardowego wejścia. Nie zna ona jednak rozmiaru tablicy i nie wie ile znaków może ona przechować przez co będzie czytać tyle znaków, aż napotka biały znak (format %s nakazuje czytanie pojedynczego słowa), co może doprowadzić do przepełnienia bufora. Niebezpieczne skutki czegoś takiego opisane są w rozdziale poświęconym napisom. Na chwilę obecną musisz zapamiętać, żeby zaraz po znaku procentu podawać maksymalną liczbę znaków, które może przechować bufor, czyli liczbę o jeden mniejszą, niż rozmiar tablicy. Bezpieczna wersją powyższego kodu jest:

```
#include <stdio.h>

int main(void)
{
  char tablica[100];
  scanf("%99s", tablica);
  return 0;
}
```

Funkcja scanf() zwraca liczbę poprawnie wczytanych zmiennych lub EOF jeżeli nie ma już danych w strumieniu lub nastąpił błąd. Załóżmy dla przykładu, że chcemy stworzyć program, który odczytuje po kolei liczby i wypisuje ich 3 potęgi. W pewnym momencie dane się kończą lub jest wprowadzana niepoprawna dana i wówczas nasz program powinien zakończyć działanie. Aby to zrobić, należy sprawdzać wartość zwracaną przez funkcję scanf() w warunku pętli:

```
#include <stdio.h>

int main(void)
{
  int n;
  while (scanf("%d", &n)==1) {
    printf("%d\n", n*n*n);
  }
  return 0;
}
```

Podobnie możemy napisać program, który wczytuje po dwie liczby i je sumuje:

```
#include <stdio.h>

int main(void)
{
  int a, b;
  while (scanf("%d %d", &a, &b)==2) {
    printf("%d\n", a+b);
  }
  return 0;
}
```

Rozpatrzmy teraz trochę bardziej skomplikowany przykład. Otóż, ponownie jak poprzednio nasz program będzie wypisywał 3 potęgę podanej liczby, ale tym razem

musi ignorować błędne dane (tzn. pomijać ciągi znaków, które nie są liczbami) i kończyć działanie tylko w momencie, gdy nastąpi błąd odczytu lub koniec pliku[3].

```
#include <stdio.h>

int main(void)
{
  int result, n;
  do {
    result = scanf("%d", &n);
    if (result==1) {
      printf("%d\n", n*n*n);
    } else if (!result) { /* !result to to samo co result==0 */
      result = scanf("%*s");
    }
  } while (result!=EOF);
  return 0;
}
```

Zastanówmy się przez chwilę co się dzieje w programie. Najpierw wywoływana jest funkcja scanf() i następuje próba odczytu liczby typu int. Jeżeli funkcja zwróciła 1 to liczba została poprawnie odczytana i następuje wypisanie jej trzeciej potęgi. Jeżeli funkcja zwróciła 0 to na wejściu były jakieś dane, które nie wyglądały jak liczba. W tej sytuacji wywołujemy funkcję scanf() z formatem odczytującym dowolny ciąg znaków nie będący białymi znakami z jednoczesnym określeniem, żeby nie zapisywała nigdzie wyniku. W ten sposób niepoprawnie wpisana dana jest omijana. Pętla główna wykonuje się tak długo jak długo funkcja scanf() nie zwróci wartości EOF.

Więcej o funkcji scanf()

Funkcja gets

Funkcja gets służy do wczytania pojedynczej linii. Może Ci się to wydać dziwne, ale: funkcji tej **nie należy używać pod żadnym pozorem**. Przyjmuje ona jeden argument — adres pierwszego elementu tablicy, do którego należy zapisać odczytaną linię — i nic poza tym. Z tego powodu nie ma żadnej możliwości przekazania do tej funkcji rozmiaru bufora podanego jako argument. Podobnie jak w przypadku scanf() może to doprowadzić do przepełnienia bufora, co może mieć tragiczne skutki. Zamiast tej funkcji należy używać funkcji fgets().

Więcej o funkcji gets()

Funkcja fgets

Funkcja fgets() jest bezpieczną wersją funkcji gets(), która dodatkowo może operować na dowolnych strumieniach wejściowych. Jej użycie jest następujące:

```
fgets(tablica_znaków, rozmiar_tablicy_znaków, stdin);
```

Na chwilę obecną nie musisz się przejmować ostatnim argumentem (jest to określenie strumienia, w naszym przypadku standardowe wejście — **st**an**d**ard **in**put). Funkcja

[3] Jak rozróżniać te dwa zdarzenia dowiesz się w rozdziale Czytanie i pisanie do plików.

czyta tekst aż do napotkania znaku przejścia do nowej linii, który także zapisuje w wynikowej tablicy (funkcja gets() tego nie robi). Jeżeli brakuje miejsca w tablicy to funkcja przerywa czytanie, w ten sposób, aby sprawdzić czy została wczytana cała linia czy tylko jej część należy sprawdzić czy ostatnim znakiem nie jest znak przejścia do nowej linii. Jeżeli nastąpił jakiś błąd lub na wejściu nie ma już danych funkcja zwraca wartość NULL.

```
#include <stdio.h>

int main(void) {
  char buffer[128], whole_line = 1, *ch;
  while (fgets(buffer, sizeof buffer, stdin)) { /* 1 */
    if (whole_line) {                           /* 2 */
      putchar('>');
      if (buffer[0]!='>') {
        putchar(' ');
      }
    }
    fputs(buffer, stdout);                      /* 3 */
    for (ch = buffer; *ch && *ch!='\n'; ++ch);  /* 4 */
    whole_line = *ch == '\n';
  }
  if (!whole_line) {
    putchar('\n');
  }
  return 0;
}
```

Powyższy kod wczytuje dane ze standardowego wejścia — linia po linii — i dodaje na początku każdej linii znak większości, po którym dodaje spację jeżeli pierwszym znakiem na linii nie jest znak większości. W linijce 1 następuje odczytywanie linii. Jeżeli nie ma już więcej danych lub nastąpił błąd wejścia funkcja zwraca wartość NULL, która ma logiczną wartość 0 i wówczas pętla kończy działanie. W przeciwnym wypadku funkcja zwraca po prostu pierwszy argument, który ma wartość logiczną 1. W linijce 2 sprawdzamy, czy poprzednie wywołanie funkcji wczytało całą linię, czy tylko jej część — jeżeli całą to teraz jesteśmy na początku linii i należy dodać znak większości. W linii 3 najzwyczajniej w świecie wypisujemy linię. W linii 4 przeszukujemy tablicę znak po znaku, aż do momentu, gdy znajdziemy znak o kodzie 0 kończącym ciąg znaków albo znak przejścia do nowej linii. Ten drugi przypadek oznacza, że funkcja fgets() wczytała całą linię.

Więcej o funkcji fgets()

Funkcja getchar()

Jest to bardzo prosta funkcja, wczytująca 1 znak z klawiatury. W wielu przypadkach dane mogą być buforowane przez co wysyłane są do programu dopiero, gdy bufor zostaje przepełniony lub na wejściu jest znak przejścia do nowej linii. Z tego powodu po wpisaniu danego znaku należy nacisnąć klawisz enter, aczkolwiek trzeba pamiętać, że w następnym wywołaniu zostanie zwrócony znak przejścia do nowej linii. Gdy nastąpił błąd lub nie ma już więcej danych funkcja zwraca wartość EOF (która ma

jednak wartość logiczną 1 toteż zwykła pętla `while (getchar())` nie da oczekiwanego rezultatu):

```
#include <stdio.h>

int main(void)
{
  int c;
  while ((c = getchar())!=EOF) {
    if (c==' ') {
      c = '_';
    }
    putchar(c);
  }
  return 0;
}
```

Ten prosty program wczytuje dane znak po znaku i zamienia wszystkie spacje na znaki podkreślenia. Może wydać się dziwne, że zmienną c zdefiniowaliśmy jako trzymającą typ int, a nie char. Właśnie taki typ (tj. int) zwraca funkcja getchar() i jest to konieczne ponieważ wartość EOF wykracza poza zakres wartości typu char (gdyby tak nie było to nie byłoby możliwości rozróżnienia wartości EOF od poprawnie wczytanego znaku). Więcej o funkcji getchar()

Rozdział 11

Funkcje

W matematyce pod pojęciem funkcji rozumiemy twór, który pobiera pewną liczbę argumentów i zwraca wynik. Jeśli dla przykładu weźmiemy funkcję `sin(x)` to x będzie zmienną rzeczywistą, która określa kąt, a w rezultacie otrzymamy inną liczbę rzeczywistą — sinus tego kąta.

W C **funkcja** (czasami nazywana *podprogramem*, rzadziej *procedurą*) to wydzielona część programu, która przetwarza argumenty i ewentualnie zwraca wartość, która następnie może być wykorzystana jako argument w innych działaniach lub funkcjach. Funkcja może posiadać własne zmienne lokalne. W odróżnieniu od funkcji matematycznych, funkcje w C mogą zwracać dla tych samych argumentów różne wartości.

Po lekturze poprzednich części podręcznika zapewne mógłbyś podać kilka przykładów funkcji, z których korzystałeś. Były to np.

- funkcja `printf()`, drukująca tekst na ekranie, czy
- funkcja `main()`, czyli główna funkcja programu.

Główną motywacją tworzenia funkcji jest unikanie powtarzania kilka razy tego samego kodu. W poniższym fragmencie:

```
for(i=1; i <= 5; ++i) {
  printf("%d ", i*i);
}
for(i=1; i <= 5; ++i) {
  printf("%d ", i*i*i);
}
for(i=1; i <= 5; ++i) {
  printf("%d ", i*i);
}
```

widzimy, że pierwsza i trzecia pętla `for` są takie same. Zamiast kopiować fragment kodu kilka razy (co jest mało wygodne i może powodować błędy) lepszym rozwiązaniem mogłoby być wydzielenie tego fragmentu tak, by można bo było wywoływać kilka razy. Tak właśnie działają funkcje.

Innym, niemniej ważnym powodem używania funkcji jest rozbicie programu na fragmenty wg ich funkcjonalności. Oznacza to, że z jeden duży program dzieli się na mniejsze funkcje, które są "wyspecjalizowane" w wykonywaniu określonych czynności. Dzięki temu łatwiej jest zlokalizować błąd. Ponadto takie funkcje można potem przenieść do innych programów.

Tworzenie funkcji

Dobrze jest uczyć się na przykładach. Rozważmy następujący kod:

```
int iloczyn (int x, int y)
{
  int iloczyn_xy;
  iloczyn_xy = x*y;
  return iloczyn_xy;
}
```

`int iloczyn (int x, int y)` to nagłówek funkcji, który opisuje, jakie argumenty przyjmuje funkcja i jaką wartość zwraca (funkcja może przyjmować wiele argumentów, lecz może zwracać tylko jedną wartość)[1]. Na początku podajemy typ zwracanej wartości — u nas `int`. Następnie mamy nazwę funkcji i w nawiasach listę argumentów.

Ciało funkcji (czyli wszystkie wykonywane w niej operacje) umieszczamy w nawiasach klamrowych. Pierwszą instrukcją jest deklaracja zmiennej — jest to zmienna lokalna, czyli niewidoczna poza funkcją. Dalej przeprowadzamy odpowiednie działania i zwracamy rezultat za pomocą instrukcji `return`.

Ogólnie

Funkcję w języku C tworzy się następująco:

```
typ identyfikator (typ1 argument1, typ2 argument2, typn argumentn)
{
  /* instrukcje */
}
```

Oczywiście istnieje możliwość utworzenia funkcji, która nie posiada żadnych argumentów. Definiuje się ją tak samo, jak funkcję z argumentami z tą tylko różnicą, że między okrągłymi nawiasami nie znajduje się żaden argument lub pojedyncze słówko void — w definicji funkcji nie ma to znaczenia, jednak w deklaracji puste nawiasy oznaczają, że prototyp nie informuje jakie argumenty przyjmuje funkcja, dlatego bezpieczniej jest stosować słówko void.

Funkcje definiuje się poza główną funkcją programu (main). W języku C nie można tworzyć zagnieżdżonych funkcji (funkcji wewnątrz innych funkcji).

Procedury

Przyjęło się, że procedura od funkcji różni się tym, że ta pierwsza nie zwraca żadnej wartości. Zatem, aby stworzyć procedurę należy napisać:

```
void identyfikator (argument1, argument2, argumentn)
{
  /* instrukcje */
}
```

[1] Bardziej precyzyjnie można powiedzieć, że funkcja może zwrócić tylko jeden adres do jakiegoś obiektu w pamięci.

`void` (z ang. pusty, próżny) jest słowem kluczowym mającym kilka znaczeń, w tym przypadku oznacza "brak wartości".

Generalnie, w terminologii C pojęcie "procedura" nie jest używane, mówi się raczej "funkcja zwracająca void".

Jeśli nie podamy typu danych zwracanych przez funkcję kompilator domyślnie przyjmie typ int, choć już w standardzie C99 nieokreślenie wartości zwracanej jest błędem.

Stary sposób definiowania funkcji

Zanim powstał standard ANSI C, w liście parametrów nie podawało się typów argumentów, a jedynie ich nazwy. Również z tamtych czasów wywodzi się oznaczenie, iż puste nawiasy (w prototypie funkcji, nie w definicji) oznaczają, że funkcja przyjmuje nieokreśloną liczbę argumentów. Tego archaicznego sposobu definiowania funkcji nie należy już stosować, ale ponieważ w swojej przygodzie z językiem C Czytelnik może się na nią natknąć, a co więcej standard nadal (z powodu zgodności z wcześniejszymi wersjami) dopuszcza taką deklarację to należy tutaj o niej wspomnieć. Otóż wygląda ona następująco:

```
typ_zwracany nazwa_funkcji(argument1, argument2, argumentn)
  typ1 argumenty /*, ... */;
  typ2 argumenty /*, ... */;
  /* ... */
{
  /* instrukcje */
}
```

Na przykład wcześniejsza funkcja iloczyn wyglądałaby następująco:

```
int iloczyn(x, y)
  int x, y;
{
  int iloczyn_xy;
  iloczyn_xy = x*y;
  return iloczyn_xy;
}
```

Najpoważniejszą wadą takiego sposobu jest fakt, że w prototypie funkcji nie ma podanych typów argumentów, przez co kompilator nie jest w stanie sprawdzić poprawności wywołania funkcji. Naprawiono to (wprowadzając definicje takie jak je znamy obecnie) najpierw w języku C++, a potem rozwiązanie zapożyczono w standardzie ANSI C z 1989 roku.

Wywoływanie

Funkcje wywołuje się następująco:

```
identyfikator (argument1, argument2, argumentn);
```

Jeśli chcemy, aby przypisać zmiennej wartość, którą zwraca funkcja, należy napisać tak:

```
zmienna = funkcja (argument1, argument2, argumentn);
```

Programiści mający doświadczenia np. z językiem Pascal mogą popełniać błąd polegający na wywoływaniu funkcji bez nawiasów okrągłych, gdy nie przyjmuje ona żadnych argumentów.

Przykładowo, mamy funkcję:

```
void pisz_komunikat()
{
  printf("To jest komunikat\n");
}
```

Jeśli teraz ją wywołamy:

```
pisz_komunikat;   /* ŹLE    */
pisz_komunikat(); /* dobrze */
```

to pierwsze polecenie nie spowoduje wywołania funkcji. Dlaczego? Aby kompilator C zrozumiał, że chodzi nam o wywołanie funkcji, musimy po jej nazwie dodać nawiasy okrągłe, nawet, gdy funkcja nie ma argumentów. Użycie samej nazwy funkcji ma zupełnie inne znaczenie — oznacza pobranie jej adresu. W jakim celu? O tym będzie mowa w rozdziale Wskaźniki.

Przykład

A oto działający przykład, który demonstruje wiadomości podane powyżej:

```
#include <stdio.h>

int suma (int a, int b)
{
  return a+b;
}

int main ()
{
  int m = suma (4, 5);
  printf ("4+5=%d\n", m);
  return 0;
}
```

Zwracanie wartości

`return` to prawdopodobnie pierwsze słowo kluczowe języka C, z którym zetknąłeś się dotychczas. Służy ono do przerwania funkcji i zwrócenia wartości lub też przerwania funkcji bez zwracania wartości — dzieje się tak np. w procedurach. Użycie tej instrukcji jest bardzo proste i wygląda tak:

```
return zwracana_wartość;
```

lub dla procedur:

```
return;
```

Funkcja main()

Do tej pory we wszystkich programach istniała funkcja `main()`. Po co tak właściwie ona jest? Otóż jest to funkcja, która zostaje wywołana przez fragment kodu inicjującego pracę programu. Kod ten tworzony przez kompilator i nie mamy na niego wpływu. Istotne jest, że każdy program w języku C **musi zawierać** funkcję `main()`.

Istnieją dwa możliwe prototypy (nagłówki) omawianej funkcji: `int main(void);` lub `int main(int argc, char **argv);`[2]. Argument `argc` jest liczba nieujemną określającą, ile ciągów znaków przechowywanych jest w tablicy `argv`. Wyrażenie `argv[argc]` ma zawsze wartość NULL. Pierwszym elementem tablicy `argv` (o ile istnieje[3]) jest nazwa programu czy komenda, którą program został uruchomiony. Pozostałe przechowują argumenty podane przy uruchamianiu programu.

Zazwyczaj jeśli program uruchomimy poleceniem *program argument1 argument2* to `argc` będzie równe 3 (2 argumenty + nazwa programu) a `argv` będzie zawierać napisy *program*, *argument1*, *argument2* umieszczone w tablicy indeksowanej od 0 do 2.

Weźmy dla przykładu program, który wypisuje to, co otrzymuje w argumentach `argc` i `argv`:

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

int main(int argc, char **argv) {
  while (*argv) {
    puts(*argv++);
  }
  /* Ewentualnie można użyc:
  int i;
  for (i = 0; i<argc; ++i) {
    puts(argv[i]);
  }
  */
  return EXIT_SUCCESS;
}
```

Uruchomiony w systemie typu UNIX poleceniem `./test foo bar baz` powinien wypisać:

```
./test
foo
bar
baz
```

[2] Czasami można się spotkać z prototypem `int main(int argc, char **argv, char **env);`, który jest definiowany w standardzie POSIX, ale wykracza już poza standard C.

[3] Inne standardy mogą wymuszać istnienie tego elementu, jednak jeśli chodzi o standard języka C to nic nie stoi na przeszkodzie, aby argument argc miał wartość zero.

Na razie nie musisz rozumieć powyższych kodów i opisów, gdyż odwołują się do pojęć takich jak tablica oraz wskaźnik, które opisane zostaną w dalszej części podręcznika.

Co ciekawe, funkcja main nie różni się zanadto od innych funkcji i tak jak inne może wołać sama siebie (patrz rekurencja niżej), przykładowo powyższy program można zapisać tak[4]:

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

int main(int argc, char **argv) {
  if (*argv) {
    puts(*argv);
    return main(argc-1, argv+1);
  } else {
    return EXIT_SUCCESS;
  }
}
```

Ostatnią rzeczą dotyczącą funkcji main jest **zwracana** przez nią **wartość**. Już przy omawianiu pierwszego programu wspomniane zostało, że jedynymi wartościami, które znaczą zawsze to samo we wszystkich implementacjach języka są 0, EXIT_SUCCESS i EXIT_FAILURE[5] zdefiniowane w pliku nagłówkowym stdlib.h. Wartość 0 i EXIT_SUCCESS oznaczają poprawne zakończenie programu (co wcale nie oznacza, że makro EXIT_SUCCESS ma wartość zero), natomiast EXIT_FAILURE zakończenie błędne. Wszystkie inne wartości są zależne od implementacji.

Dalsze informacje

Poniżej przekażemy ci parę bardziej zaawansowanych informacji o funkcjach w C, jeśli nie masz ochoty wgłębiać się w szczegóły, możesz spokojnie pominąć tę część i wrócić tu później.

Jak zwrócić kilka wartości?

Jeśli chcesz zwrócić z funkcji kilka wartości, musisz zrobić to w trochę inny sposób. Generalnie możliwe są dwa podejścia: jedno to "upakowanie" zwracanych wartości – można stworzyć tak zwaną *strukturę*, która będzie przechowywała kilka zmiennych (jest to opisane w rozdziale Typy złożone). Prostszym sposobem jest zwracanie jednej z wartości w normalny sposób a pozostałych jako parametrów. Za chwilę dowiesz się, jak to zrobić; jeśli chcesz zobaczyć przykład, możesz przyjrzeć się funkcji scanf() z biblioteki standardowej.

[4] Jeżeli ktoś lubi ekstrawagancki kod ciało funkcji main można zapisać jako `return *argv ? puts(*argv), main(argc-1, argv+1) : EXIT_SUCCESS;`, ale nie radzimy stosować tak skomplikowanych i, bądź co bądź, mało czytelnych konstrukcji.

[5] Uwaga! Makra EXIT_SUCCESS i EXIT_FAILURE te służą tylko i wyłącznie jako wartości do zwracania przez funkcję main(). Nigdzie indziej nie mają one zastosowania.

Przekazywanie parametrów

Gdy wywołujemy funkcję, wartość argumentów, z którymi ją wywołujemy, jest kopiowana do funkcji. Kopiowana — to znaczy, że nie możemy normalnie zmienić wartości zewnętrznych dla funkcji zmiennych. Formalnie mówi się, że w C **argumenty** są **przekazywane przez wartość**, czyli wewnątrz funkcji operujemy tylko na ich kopiach.

Możliwe jest modyfikowanie zmiennych przekazywanych do funkcji jako parametry — ale do tego w C potrzebne są wskaźniki.

Funkcje rekurencyjne

Język C ma możliwość tworzenia tzw. **funkcji rekurencyjnych**. Jest to funkcja, która w swojej własnej definicji (ciele) wywołuje samą siebie. Najbardziej klasycznym przykładem może tu być silnia. Napiszmy sobie zatem naszą funkcję rekurencyjną, która oblicza silnię:

```
int silnia (int liczba)
{
  int sil;
  if (liczba<0) return 0; /* wywołanie jest bezsensowne, */
                          /* zwracamy 0 jako kod błędu   */
  if (liczba==0 || liczba==1) return 1;
  sil = liczba*silnia(liczba-1);
  return sil;
}
```

Musimy być ostrożni przy funkcjach rekurencyjnych, gdyż łatwo za ich pomocą utworzyć funkcję, która będzie sama siebie wywoływała w nieskończoność, a co za tym idzie będzie zawieszała program. Tutaj pierwszymi instrukcjami `if` ustalamy "warunki stopu", gdzie kończy się wywoływanie funkcji przez samą siebie, a następnie określamy, jak funkcja będzie wywoływać samą siebie (odjęcie jedynki od argumentu, co do którego wiemy, że jest dodatni, gwarantuje, że dojdziemy do warunku stopu w skończonej liczbie kroków).

Warto też zauważyć, że funkcje rekurencyjne czasami mogą być znacznie wolniejsze niż podejście nierekurencyjne (iteracyjne, przy użyciu pętli). Flagowym przykładem może tu być funkcja obliczająca wyrazy ciągu Fibonacciego:

```
#include <stdio.h>

unsigned count;

unsigned fib_rec(unsigned n) {
  ++count;
  return n<2 ? n : (fib_rec(n-2) + fib_rec(n-1));
}

unsigned fib_it (unsigned n) {
  unsigned a = 0, b = 0, c = 1;
  ++count;
  if (!n) return 0;
```

```
  while (--n) {
    ++count;
    a = b;
    b = c;
    c = a + b;
  }
  return c;
}

int main(void) {
  unsigned n, result;
  printf("Ktory element ciagu Fibonacciego obliczyc? ");
  while (scanf("%d", &n)==1) {
    count = 0;
    result = fib_rec(n);
    printf("fib_ret(%3u) = %6u  (wywolan: %5u)\n", n, result, count);

    count = 0;
    result = fib_it (n);
    printf("fib_it (%3u) = %6u  (wywolan: %5u)\n", n, result, count);
  }
  return 0;
}
```

W tym przypadku funkcja rekurencyjna, choć łatwiejsza w napisaniu, jest bardzo nieefektywna.

Deklarowanie funkcji

Czasami możemy chcieć przed napisaniem funkcji poinformować kompilator, że dana funkcja istnieje. Niekiedy kompilator może zaprotestować, jeśli użyjemy funkcji przed określeniem, jaka to funkcja, na przykład:

```
int a()
{
  return b(0);
}

int b(int p)
{
  if( p == 0 )
    return 1;
  else
    return a();
}

int main()
{
  return b(1);
}
```

W tym przypadku nie jesteśmy w stanie zamienić a i b miejscami, bo obie funkcje korzystają z siebie nawzajem. Rozwiązaniem jest wcześniejsze zadeklarowanie funkcji. Deklaracja funkcji (zwana czasem *prototypem*) to po prostu przekopiowana pierwsza linijka funkcji (przed otwierającym nawiasem klamrowym) z dodatkowo dodanym średnikiem na końcu. W naszym przykładzie wystarczy na samym początku wstawić:

```
int b(int p);
```

W deklaracji można pominąć nazwy parametrów funkcji:

```
int b(int);
```

Bardzo częstym zwyczajem jest wypisanie przed funkcją main samych prototypów funkcji, by ich definicje umieścić po definicji funkcji main, np.:

```
int a(void);
int b(int p);

int main()
{
  return b(1);
}

int a()
{
  return b(0);
}

int b(int p)
{
  if( p == 0 )
    return 1;
  else
    return a();
}
```

Z poprzednich rozdziałów pamiętasz, że na początku programu dołączaliśmy tzw. **pliki nagłówkowe**. Zawierają one właśnie prototypy funkcji i ułatwiają pisanie dużych programów. Dalsze informacje o plikach nagłówkowych zawarte są w rozdziale Tworzenie bibliotek.

Zmienna liczba parametrów

Zauważyłeś zapewne, że używając funkcji printf() lub scanf() po argumencie zawierającym tekst z odpowiednimi modyfikatorami mogłeś podać praktycznie nieograniczoną liczbę argumentów. Zapewne deklaracja obu funkcji zadziwi Cię jeszcze bardziej:

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

Jak widzisz w deklaracji zostały użyte 3 kropki. Otóż język C ma możliwość przekazywania nieograniczonej liczby argumentów do funkcji (tzn. jedynym ograniczeniem

jest rozmiar stosu programu). Cała zabawa polega na tym, aby umieć dostać się do odpowiedniego argumentu oraz poznać jego typ (używając funkcji printf, mogliśmy wpisać jako argument dowolny typ danych). Do tego celu możemy użyć wszystkich ciekawostek, zawartych w pliku nagłówkowym stdarg.h.

Załóżmy, że chcemy napisać prostą funkcję, która dajmy na to, mnoży wszystkie swoje argumenty (zakładamy, że argumenty są typu int). Przyjmujemy przy tym, że ostatni argument będzie 0. Będzie ona wyglądała tak:

```
#include <stdarg.h>

int mnoz (int pierwszy, ...)
{
  va_list arg;
  int iloczyn = 1, t;
  va_start (arg, pierwszy);
  for (t = pierwszy; t; t = va_arg(arg, int)) {
    iloczyn *= t;
  }
  va_end (arg);
  return iloczyn;
}
```

va_list oznacza specjalny typ danych, w którym przechowywane będą argumenty, przekazane do funkcji. Makropolecenie va_arg odczytuje kolejne argumenty i przekształca je do odpowiedniego typu danych. Na zakończenie używane jest makro va_end — jest ono obowiązkowe!

Oczywiście, tak samo jak w przypadku funkcji printf() czy scanf(), argumenty nie muszą być takich samych typów. Rozważmy dla przykładu funkcję, podobną do printf(), ale znacznie uproszczoną:

```
#include <stdarg.h>

void wypisz(const char *format, ...) {
  va_list arg;
  va_start (arg, format);
  for (; *format; ++format) {
    switch (*format) {
    case 'i': printf("%d" , va_arg(arg, int)); break;
    case 'I': printf("%u" , va_arg(arg, unsigned)); break;
    case 'l': printf("%ld", va_arg(arg, int)); break;
    case 'L': printf("%lu", va_arg(arg, unsigned long)); break;
    case 'f': printf("%f" , va_arg(arg, double)); break;
    case 'x': printf("%x" , va_arg(arg, unsigned)); break;
    case 'X': printf("%X" , va_arg(arg, unsigned)); break;
    case 's': printf("%s" , va_arg(arg, const char *)); break;
    default : putc(*format);
    }
  }
  va_end (arg);
}
```

Przyjmuje ona jako argument ciąg znaków, w których niektóre instruują funkcję, by pobrała argument i go wypisała. Nie przejmuj się jeżeli nie rozumiesz wyrażeń `*format` i `++format`. Istotne jest to, że pętla sprawdza po kolei wszystkie znaki formatu.

Ezoteryka C

C ma wiele niuansów, o których wielu programistów nie wie lub łatwo o nich zapomina:

- jeśli nie podamy typu wartości zwracanej w funkcji, zostanie przyjęty typ int (według najnowszego standardu C99 nie podanie typu wartości jest zwracane jako błąd);
- jeśli nie podamy żadnych parametrów funkcji, to funkcja będzie używała zmiennej ilości parametrów (inaczej niż w C++, gdzie przyjęte zostanie, że funkcja nie przyjmuje argumentów). Aby wymusić pustą listę argumentów, należy napisać `int funkcja(void)` (dotyczy to jedynie prototypów czy deklaracji funkcji);
- jeśli nie użyjemy w funkcji instrukcji `return`, wartość zwracana będzie przypadkowa (dostaniemy śmieci z pamięci).

Kompilator C++ użyty do kompilacji kodu C najczęściej zaprotestuje i ostrzeże nas, jeśli użyjemy powyższych konstrukcji. Natomiast czysty kompilator C z domyślnymi ustawieniami nie napisze nic i bez mrugnięcia okiem skompiluje taki kod.

Zobacz też

- C++/Funkcje inline — funkcje rozwijane w miejscu wywoływania (dostępne też w standardzie C99).
- C++/Przeciążanie funkcji

Rozdział 12

Preprocesor

Wstęp

W języku C wszystkie linijki, zaczynające się od symbolu "#" nie podlegają bezpośrednio procesowi kompilacji. Są to natomiast instrukcje **preprocesora** — elementu kompilatora, który analizuje plik źródłowy w poszukiwaniu wszystkich wyrażeń, zaczynających się od "#". Na podstawie tych instrukcji generuje on kod w "czystym" języku C, który następnie jest kompilowany przez kompilator. Ponieważ za pomocą preprocesora można niemal "sterować" kompilatorem daje on niezwykłe możliwości, które nie były dotąd znane w innych językach programowania. Aby przekonać się, jak wygląda kod, przetworzony przez preprocesor użyj (w kompilatorze gcc) przełącznika "-E":

```
gcc test.c -E -o test.txt
```

W pliku test.txt zostanie umieszczony cały kod w postaci, która zdatna jest do przetworzenia przez kompilator.

Dyrektywy preprocesora

Dyrektywy preprocesora są to wyrażenia, które występują zaraz za symbolem "#" i to właśnie za ich pomocą możemy używać preprocesora. Dyrektywa zaczyna się od znaku # i kończy się wraz z końcem linii. Aby przenieść dalszą część dyrektywy do następnej linii, należy zakończyć linię znakiem "\":

```
#define add(a,b) \
  a+b
```

Wymienimy teraz kilka ważniejszych dyrektyw.

#include

Najpopularniejsza dyrektywa, wstawiająca w swoje miejsce treść pliku podanego w nawiasach ostrych lub cudzysłowie. Składnia:

Przykład 1.

```
#include <plik_naglowkowy_do_dolaczenia>
```

Za pomocą #include możemy dołączyć dowolny plik — niekoniecznie plik nagłówkowy.

Przykład 2.

```
#include "plik_naglowkowy_do_dolaczenia"
```

Jeżeli nazwa pliku nagłówkowego będzie ujęta w nawiasy ostre (przykład 1), to kompilator poszuka go wśród własnych plików nagłówkowych (które najczęściej się znajdują w podkatalogu "includes" w katalogu kompilatora). Jeśli jednak nazwa ta będzie ujęta w podwójne cudzysłowy(przykład 2), to kompilator poszuka jej w katalogu, w którym znajduje się kompilowany plik (można zmienić to zachowanie w opcjach niektórych kompilatorów). Przy użyciu tej dyrektywy można także wskazać dokładne położenie plików nagłówkowych poprzez wpisanie bezwzględnej lub względnej ścieżki dostępu do tego pliku nagłówkowego.

Przykład 3 — ścieżka bezwzględna do pliku nagłówkowego w Linuksie i w Windowsie

Opis: W miejsce jednej i drugiej linijki zostanie wczytany plik umieszczony w danej lokalizacji

```
#include "/usr/include/plik_nagłówkowy.h"
#include "C:\\borland\includes\plik_nagłówkowy.h"
```

Przykład 4 — ścieżka względna do pliku nagłówkowego

Opis: W miejsce linijki zostanie wczytany plik umieszczony w katalogu "katalog1", a ten katalog jest w katalogu z plikiem źródłowym. Inaczej mówiąc, jeśli plik źródłowy jest w katalogu "/home/user/dokumenty/zrodla", to plik nagłówkowy jest umieszczony w katalogu "/home/user/dokumenty/zrodla/katalog1"

```
#include "katalog1/plik_naglowkowy.h"
```

Przykład 5 — ścieżka względna do pliku nagłówkowego

Opis: Jeśli plik źródłowy jest umieszczony w katalogu "/home/user/dokumenty-/zrodla", to plik nagłówkowy znajduje się w katalogu "/home/user/dokumenty/katalog1/katalog2/"

```
#include "../katalog1/katalog2/plik_naglowkowy.h"
```

Więcej informacji możesz uzyskać w rozdziale Biblioteki.

#define

Linia pozwalająca zdefiniować stałą, funkcję lub słowo kluczowe, które będzie potem podmienione w kodzie programu na odpowiednią wartość lub może zostać użyte w instrukcjach warunkowych dla preprocesora. Składnia:

```
#define NAZWA_STALEJ WARTOSC
```

lub

```
#define NAZWA_STALEJ
```

Przykład:

#define LICZBA 8 — spowoduje ,że każde wystąpienie słowa LICZBA w kodzie zostanie zastąpione ósemką.

#define SUMA(a,b) (a+b) — spowoduje, ze każde wystąpienie wywołania "funkcji" SUMA zostanie zastąpione przez sumę argumentów

#undef

Ta instrukcja odwołuje definicję wykonaną instrukcją *#define*.

```
#undef STALA
```

instrukcje warunkowe

Preprocesor zawiera również instrukcje warunkowe, pozwalające na wybór tego co ma zostać skompilowane w zależności od tego, czy stała jest zdefiniowana lub jaką ma wartość:

#if #elif #else #endif

Te instrukcje uzależniają kompilacje od warunków. Ich działanie jest podobne do instrukcji warunkowych w samym języku C. I tak:

#if wprowadza warunek, który jeśli nie jest prawdziwy powoduje pominięcie kompilowania kodu, aż do napotkania jednej z poniższych instrukcji.

#else spowoduje skompilowanie kodu jeżeli warunek za *#if* jest nieprawdziwy, aż do napotkania któregoś z poniższych instrukcji.

#elif wprowadza nowy warunek, który będzie sprawdzony jeżeli poprzedni był nieprawdziwy. Stanowi połączenie instrukcji *#if* i *#else*.

#endif zamyka blok ostatniej instrukcji warunkowej.

Przykład:

```
#if INSTRUKCJE == 2
  printf ("Podaj liczbę z przedziału 10 do 0\n"); /*1*/
#elif INSTRUKCJE == 1
  printf ("Podaj liczbę: "); /*2*/
#else
  printf ("Podaj parametr: "); /*3*/
#endif
scanf ("%d\n", &liczba);/*4*/
```

- wiersz nr 1 zostanie skompilowany jeżeli stała INSTRUKCJE będzie równa 2
- wiersz nr 2 zostanie skompilowany, gdy INSTRUKCJE będzie równa 1
- wiersz nr 3 zostanie skompilowany w pozostałych wypadkach
- wiersz nr 4 będzie kompilowany zawsze

#ifdef #ifndef #else #endif

Te instrukcje warunkują kompilację od tego, czy odpowiednia stała została zdefiniowana.

#ifdef spowoduje, że kompilator skompiluje poniższy kod tylko gdy została zdefiniowana odpowiednia stała.

#ifndef ma odwrotne działanie do *#ifdef*, a mianowicie brak definicji odpowiedniej stałej umożliwia kompilacje poniższego kodu.

#else,#endif mają identyczne zastosowanie jak te z powyższej grupy

Przykład:

```
#define INFO /*definicja stałej INFO*/
#ifdef INFO
  printf ("Twórcą tego programu jest Jan Kowalski\n");/*1*/
#endif
#ifndef INFO
  printf ("Twórcą tego programu jest znany programista\n");/*2*/
#endif
```

To czy dowiemy się kto jest twórcą tego programu zależy czy instrukcja definiująca stałą INFO będzie istnieć. W powyższym przypadku na ekranie powinno się wyświetlić

```
Twórcą tego programu jest Jan Kowalski
```

#error

Powoduje przerwanie kompilacji i wyświetlenie tekstu, który znajduje się za tą instrukcją. Przydatne gdy chcemy zabezpieczyć się przed zdefiniowaniem nieodpowiednich stałych.

Przykład:

```
#if BLAD == 1
#error "Poważny błąd kompilacji"
#endif
```

Co jeżeli zdefiniujemy stałą BLAD z wartością 1? Spowoduje to wyświetlenie w trakcie kompilacji komunikatu podobnego do poniższego:

```
Fatal error program.c 6: Error directive: "Poważny błąd kompilacji"
in function main()
*** 1 errors in Compile ***
```

wraz z przerwaniem kompilacji.

#warning

Wyświetla tekst, zawarty w cudzysłowach, jako ostrzeżenie. Jest często używany do sygnalizacji programiście, że dana część programu jest przestarzała lub może sprawiać problemy.

Przykład:

```
#warning "To jest bardzo prosty program"
```

Spowoduje to takie oto zachowanie kompilatora:

```
test.c:3:2: warning: \#warning ‘‘To jest bardzo prosty program’’
```

Użycie dyrektywy #warning nie przerywa procesu kompilacji i służy tylko do wyświetlania komunikatów dla programisty w czasie kompilacji programu.

#line

Powoduję wyzerowanie licznika linii kompilatora, który jest używany przy wyświetlaniu opisu błędów kompilacji. Pozwala to na szybkie znalezienie możliwej przyczyny błędu w rozbudowanym programie.

Przykład:

```
printf ("Podaj wartość funkcji");
#line
printf ("W przedziale od 10 do 0\n); /* tutaj jest błąd - brak cudzysłowu zamykającego */
```

Jeżeli teraz nastąpi próba skompilowania tego kodu to kompilator poinformuje, że wystąpił błąd składni w lini **1**, a nie np. **258**.

Makra

Preprocesor języka C umożliwia też tworzenie makr, czyli automatycznie wykonywanych czynności. Makra deklaruje się za pomocą dyrektywy #define:

```
#define MAKRO(arg1, arg2, ...) (wyrażenie)
```

W momencie wystąpienia MAKRA w tekście, preprocesor automatycznie zamieni makro na wyrażenie. Makra mogą być pewnego rodzaju alternatywami dla funkcji, ale powinno się ich używać tylko w specjalnych przypadkach. Ponieważ makro sprowadza się do prostego zastąpienia przez preprocesor wywołania makra przez jego tekst, jest bardzo podatne na trudne do zlokalizowania błędy (kompilator będzie podawał błędy w miejscach, w których nic nie widzimy — bo preprocesor wstawił tam tekst). Makra są szybsze (nie następuje wywołanie funkcji, które zawsze zajmuje trochę czasu[1]), ale też mniej bezpieczne i elastyczne niż funkcje.

Przeanalizujmy teraz fragment kodu:

```
#include <stdio.h>
#define KWADRAT(x) ((x)*(x))

int main ()
{
  printf ("2 do kwadratu wynosi %d\n", KWADRAT(2));
  return 0;
}
```

Preprocesor w miejsce wyrażenia `KWADRAT(2)` wstawił `((2)*(2))`. Zastanówmy się, co stałoby się, gdybyśmy napisali `KWADRAT(2)`. Preprocesor po prostu wstawi napis do kodu, co da wyrażenie `((2)*(2))`, które jest nieprawidłowe. Kompilator zgłosi błąd, ale programista widzi tylko w kodzie użycie makra a nie prawdziwą przyczynę błędu. Widać tu, że bezpieczniejsze jest użycie funkcji, które dają możliwość wyspecyfikowania typów argumentów.

Nawet jeżeli program się skompiluje to makro może dawać nieoczekiwany wynik. Jest tak w przypadku poniższego kodu:

```
int x = 1;
int y = KWADRAT(++x);
```

[1] Tak naprawdę wg standardu C99 istnieje możliwość napisania funkcji, której kod także będzie wstawiany w miejscu wywołania. Odbywa się to dzięki inline.

Dzieje się tak dlatego, że makra rozwijane są przez preprocesor i kompilator widzi kod:

```
int x = 1;
int y = ((++x)*(++x));
```

Również poniższe makra są błędne pomimo, że opisany problem w nich nie występuje:

```
#define SUMA(a, b) a + b
#define ILOCZYN(a, b) a * b
```

Dają one nieoczekiwane wyniki dla wywołań:

```
SUMA(2, 2) * 2; /* 6 zamiast 8 */
ILOCZYN(2 + 2, 2 + 2); /* 8 zamiast 16 */
```

Z tego powodu istotne jest użycie nawiasów:

```
#define SUMA(a, b) ((a) + (b))
#define ILOCZYN(a, b) ((a) * (b))
```

oraz

Dość ciekawe możliwości ma w makrach znak "#". Zamienia on na napis stojący za nim identyfikator.

```
#include <stdio.h>
#define wypisz(x) printf("%s=%i\n", #x, x);

int main()
{
  int i=1;
  char a=5;
  wypisz(i);
  wypisz(a);
  return 0;
}
```

Program wypisze:

```
i=1
a=5
```

Czyli `wypisz(a)` jest rozwijane w `printf("%s=%i\n", "a", a);`.
Natomiast znaki "##" łączą dwie nazwy w jedną. Przykład:

```
#include <stdio.h>
#define abc(x) int zmienna ## x

int main()
{
  abc(nasza); /* dzięki temu zadeklarujemy zmienną o nazwie zmiennanasza */
  zmiennanasza = 2;
  return 0;
}
```

Więcej o dobrych zwyczajach w tworzeniu makr można się dowiedzieć w rozdziale Powszechne praktyki.

Predefiniowane makra

Pisząc duży program czasami spotykamy się z błędami, które ciężko znaleźć w kodzie. Aby ułatwić programiście życie, standard ANSI wprowadził **predefiniowane makra**, czyli pewne stałe, które umożliwiają wstawienie do kodu informacji np. o dokładnej dacie i czasie kompilacji oraz o pliku w którym znajduje się dany kod. Oto one:

- __DATE__ — data w momencie kompilacji
- __TIME__ — czas w momencie kompilacji
- __FILE__ — łańcuch, który zawiera nazwę pliku, który aktualnie jest kompilowany przez kompilator
- __LINE__ — definiuje numer linijki

Spróbujmy użyć tych makr w praktyce

```
#include <stdio.h>

#define BUG printf("Wystąpił błąd w pliku: %s, w wierszu: %d\n", __FILE__, __LINE__)

int main ()
{
  BUG;
}
```

Efekt działania programu: `Wystąpił błąd w pliku: test.c, w wierszu: 7`

Rozdział 13

Biblioteka standardowa

Czym jest biblioteka?

Bibliotekę w języku C stanowi zbiór skompilowanych wcześniej funkcji, który można łączyć z programem. Biblioteki tworzy się, aby udostępnić zbiór pewnych "wyspecjalizowanych" funkcji do dyspozycji innych programów. Tworzenie bibliotek jest o tyle istotne, że takie podejście znacznie ułatwia tworzenie nowych programów. Łatwiej jest utworzyć program w oparciu o istniejące biblioteki, niż pisać program wraz ze wszystkimi potrzebnymi funkcjami[1].

Po co nam biblioteka standardowa?

W którymś z początkowych rozdziałów tego podręcznika napisane jest, że czysty język C nie może zbyt wiele. Tak naprawdę, to język C sam w sobie praktycznie nie ma mechanizmów do obsługi np. wejścia-wyjścia. Dlatego też większość systemów operacyjnych posiada tzw. **bibliotekę standardową** zwaną też **biblioteką języka C**. To właśnie w niej zawarte są podstawowe funkcjonalności, dzięki którym twój program może np. napisać coś na ekranie.

Jak skonstruowana jest biblioteka standardowa?

Zapytacie się zapewne jak biblioteka standardowa realizuje te funkcje, skoro sam język C tego nie potrafi. Odpowiedź jest prosta — biblioteka standardowa nie jest napisana w samym języku C. Ponieważ C jest językiem tłumaczonym do kodu maszynowego, to w praktyce nie ma żadnych przeszkód, żeby np. połączyć go z językiem niskiego poziomu, jakim jest np. asembler. Dlatego biblioteka C z jednej strony udostępnia gotowe funkcje w języku C, a z drugiej za pomocą niskopoziomowych mechanizmów[2] komunikuje się z systemem operacyjnym, który wykonuje odpowiednie czynności.

[1] Początkujący programista zapewne nie byłby w stanie napisać nawet funkcji printf.
[2] Takich, jak np. wywoływanie przerwań programowych.

Gdzie są funkcje z biblioteki standardowej?

Pisząc program w języku C używamy różnego rodzaju funkcji, takich jak np. printf. Nie jesteśmy jednak ich autorami, mało tego nie widzimy nawet deklaracji tych funkcji w naszym programie. Pamiętacie program "Hello world"? Zaczynał on się od takiej oto linijki:

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

linijka ta oznacza: "w tym miejscu wstaw zawartość pliku stdio.h". Nawiasy "<" i ">" oznaczają, że plik stdio.h znajduje się w standardowym katalogu z plikami nagłówkowymi. Wszystkie pliki z rozszerzeniem h są właśnie plikami nagłówkowymi. Wróćmy teraz do tematu biblioteki standardowej. Każdy system operacyjny ma za zadanie wykonywać pewne funkcje na rzecz programów. Wszystkie te funkcje zawarte są właśnie w bibliotece standardowej. W systemach z rodziny UNIX nazywa się ją LibC (biblioteka języka C). To tam właśnie znajduje się funkcja printf, scanf, puts i inne.

Oprócz podstawowych funkcji wejścia-wyjścia, biblioteka standardowa udostępnia też możliwość wykonywania funkcji matematycznych, komunikacji przez sieć oraz wykonywania wielu innych rzeczy.

Jeśli biblioteka nie jest potrzebna...

Czasami korzystanie z funkcji bibliotecznych oraz standardowych plików nagłówkowych jest niepożądane np. wtedy, gdy programista pisze swój własny system operacyjny oraz bibliotekę do niego. Aby wyłączyć używanie biblioteki C w opcjach kompilatora GCC możemy dodać następujące argumenty:

```
-nostdinc -fno-builtin
```

Opis funkcji biblioteki standardowej

Podręcznik C na Wikibooks zawiera opis dużej części biblioteki standardowej C:

- Indeks alfabetyczny
- Indeks tematyczny

W systemach uniksowych możesz uzyskać pomoc dzięki narzędziu man, przykładowo pisząc:

```
man printf
```

Uwagi

Programy w języku C++ mogą dokładnie w ten sam sposób korzystać z biblioteki standardowej, ale zalecane jest, by robić to raczej w trochę odmienny sposób, właściwy dla C++. Szczegóły w podręczniku C++.

Rozdział 14

Czytanie i pisanie do plików

Pojęcie pliku

Na początku dobrze by było, abyś dowiedział się, czym jest plik. Odpowiedni artykuł dostępny jest w Wikipedii. Najprościej mówiąc, plik to pewne dane zapisane na dysku.

Identyfikacja pliku

Każdy z nas, korzystając na co dzień z komputera przyzwyczaił się do tego, że plik ma określoną nazwę. Jednak w pisaniu programu posługiwanie się całą nazwą niosło by ze sobą co najmniej dwa problemy:

- pamięciożerność — przechowywanie całego (czasami nawet 255-bajtowego łańcucha) zajmuje niepotrzebnie pamięć
- ryzyko błędów (owe błędy szerzej omówione zostały w rozdziale Napisy)

Aby uprościć korzystanie z plików programiści wpadli na pomysł, aby identyfikatorem pliku stała się liczba. Dzięki temu kod programu stał się czytelniejszy oraz wyeliminowano konieczność ciągłego korzystania z łańcuchów. Jednak sam plik nadal jest identyfikowany po swojej nazwie. Aby "przetworzyć" nazwę pliku na odpowiednią liczbę korzystamy z funkcji open lub fopen. Różnica wyjaśniona jest poniżej.

Podstawowa obsługa plików

Istnieją dwie metody obsługi czytania i pisania do plików: wysoko- i niskopoziomowa. Nazwy funkcji z pierwszej grupy zaczynają się od litery "f" (np. fopen(), fread(), fclose()), a identyfikatorem pliku jest wskaźnik na strukturę typu FILE. Owa struktura to pewna grupa zmiennych, która przechowuje dane o danym pliku — jak na przykład aktualną pozycję w nim. Szczegółami nie musisz się przejmować, funkcje biblioteki standardowej same zajmują się wykorzystaniem struktury FILE, programista może więc zapomnieć, czym tak naprawdę jest struktura FILE i traktować taką zmienną jako "uchwyt", identyfikator pliku. Druga grupa to funkcje typu read(), open(), write() i close(). Podstawowym identyfikatorem pliku jest liczba całkowita, która jednoznacznie

identyfikuje dany plik w systemie operacyjnym. Liczba ta w systemach typu UNIX jest nazywana **deskryptorem** pliku.

Należy pamiętać, że nie wolno nam używać funkcji z obu tych grup jednocześnie w stosunku do jednego, otwartego pliku, tzn. nie można najpierw otworzyć pliku za pomocą fopen(), a następnie odczytywać danych z tego samego pliku za pomocą read().

Czym różnią się oba podejścia do obsługi plików? Otóż metoda wysokopoziomowa ma swój własny bufor, w którym znajdują się dane po odczytaniu z dysku a przed wysłaniem ich do programu użytkownika. W przypadku funkcji niskopoziomowych dane kopiowane są bezpośrednio z pliku do pamięci programu. W praktyce używanie funkcji wysokopoziomowych jest prostsze a przy czytaniu danych małymi porcjami również często szybsze i właśnie ten model zostanie tutaj zaprezentowany.

Dane znakowe

Skupimy się teraz na najprostszym z możliwych zagadnień — zapisie i odczycie pojedynczych znaków oraz całych łańcuchów.

Napiszmy zatem nasz pierwszy program, który stworzy plik "test.txt" i umieści w nim tekst "Hello world":

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

int main ()
{
  FILE *fp; /* używamy metody wysokopoziomowej */
            /* musimy mieć zatem identyfikator pliku, uwaga na gwiazdkę! */
  char tekst[] = "Hello world";
  if ((fp=fopen("test.txt", "w"))==NULL) {
    printf ("Nie mogę otworzyć pliku test.txt do zapisu!\n");
    exit(1);
    }
  fprintf (fp, "%s", tekst); /* zapisz nasz łańcuch w pliku */
  fclose (fp); /* zamknij plik */
  return 0;
}
```

Teraz omówimy najważniejsze elementy programu. Jak już było wspomniane wyżej, do identyfikacji pliku używa się wskaźnika na strukturę `FILE` (czyli `FILE *`). Funkcja **fopen** zwraca ów wskaźnik w przypadku poprawnego otwarcia pliku, bądź też NULL, gdy plik nie może zostać otwarty. Pierwszy argument funkcji to nazwa pliku, natomiast drugi to **tryb dostępu** — **w** oznacza "write" (pisanie); zwrócony "uchwyt" do pliku będzie mógł być wykorzystany jedynie w funkcjach zapisujących dane. I odwrotnie, gdy otworzymy plik podając tryb **r** ("read", czytanie), będzie można z niego jedynie czytać dane. Funkcja fopen została dokładniej opisana w odpowiedniej części rozdziału o bibliotece standardowej.

Po zakończeniu korzystania z pliku należy plik zamknąć. Robi się to za pomocą funkcji fclose. Jeśli zapomnimy o zamknięciu pliku, wszystkie dokonane w nim zmiany zostaną utracone!

Pliki a strumienie

Można zauważyć, że do zapisu do pliku używamy funkcji `fprintf`, która wygląda bardzo podobnie do `printf` — jedyną różnicą jest to, że w `fprintf` musimy jako pierwszy argument podać identyfikator pliku. Nie jest to przypadek — obie funkcje tak naprawdę robią tak samo. Używana do wczytywania danych z klawiatury funkcja `scanf` też ma swój odpowiednik wśród funkcji operujących na plikach — jak nietrudno zgadnąć, nosi ona nazwę `fscanf`.

W rzeczywistości język C traktuje tak samo klawiaturę i plik — są to źródła danych, podobnie jak ekran i plik, do których możne dane kierować. Jest to myślenie typowe dla systemów typu UNIX, jednak dla użytkowników przyzwyczajonych do systemu Windows albo języków typu Pascal może być to co najmniej dziwne. Nie da się ukryć, że między klawiaturą i plikiem na dysku zachodzą podstawowe różnice i dostęp do nich odbywa się inaczej — jednak funkcje języka C pozwalają nam o tym zapomnieć i same zajmują się szczegółami technicznymi. Z punktu widzenia programisty, urządzenia te sprowadzają się do nadanego im identyfikatora. Uogólnione pliki nazywa się w C **strumieniami**.

Każdy program w momencie uruchomienia "otrzymuje" od razu trzy otwarte strumienie:

- **stdin** (wejście)
- **stdout** (wyjście)
- **stderr** (wyjście błędów)

(aby z nich korzystać należy dołączyć plik nagłówkowy stdio.h)

Pierwszy z tych plików umożliwia odczytywanie danych wpisywanych przez użytkownika, natomiast pozostałe dwa służą do wyprowadzania informacji dla użytkownika oraz powiadamiania o błędach.

Warto tutaj zauważyć, że konstrukcja:

```
fprintf (stdout, "Hej, ja działam!") ;
```

jest równoważna konstrukcji

```
printf ("Hej, ja działam!");
```

Podobnie jest z funkcją scanf():

```
fscanf (stdin, "%d", &zmienna);
```

działa tak samo jak

```
scanf("%d", &zmienna);
```

Obsługa błędów

Jeśli nastąpił błąd, możemy się dowiedzieć o jego przyczynie na podstawie zmiennej errno zadeklarowanej w pliku nagłówkowym errno.h. Możliwe jest też wydrukowanie komunikatu o błędzie za pomocą funkcji perror. Na przykład używając:

```
fp = fopen ("tego pliku nie ma", "r");
if( fp == NULL )
  {
  perror("błąd otwarcia pliku");
  exit(-10);
  }
```

dostaniemy komunikat:

```
błąd otwarcia pliku: No such file or directory
```

Zaawansowane operacje

Pora na kolejny, tym razem bardziej złożony przykład. Oto krótki program, który swoje wejście zapisuje do pliku o nazwie podanej w linii poleceń:

```
#include <stdio.h>
#include <stdlib.h>
/* program udający bardzo prymitywną wersję programu tee(1) */

int main (int argc, char *argv[])
{
   FILE *fp;
   int c;
   if (argc < 2) {
      fprintf (stderr, "Uzycie: %s nazwa_pliku\n", argv[0]);
      exit (-1);
   }
   fp = fopen (argv[1], "w");
   if (!fp) {
      fprintf (stderr, "Nie moge otworzyc pliku %s\n", argv[1]);
      exit (-1);
   }
   printf("Wcisnij Ctrl+D+Enter lub Ctrl+Z+Enter aby zakonczyc\n");
   while ( (c = fgetc(stdin)) != EOF) {
      fputc (c, stdout);
      fputc (c, fp);
   }
   fclose(fp);
   return 0;
}
```

Tym razem skorzystaliśmy już z dużo większego repertuaru funkcji. Między innymi można zauważyć tutaj funkcję fputc(), która umieszcza pojedynczy znak w pliku. Ponadto w wyżej zaprezentowanym programie została użyta stała EOF, która reprezentuje koniec pliku (ang. End Of File). Powyższy program otwiera plik, którego nazwa przekazywana jest jako pierwszy argument programu, a następnie kopiuje dane z wejścia programu (stdin) na wyjście (stdout) oraz do utworzonego pliku (identyfikowanego za pomocą fp). Program robi to dotąd, aż naciśniemy kombinację klawiszy Ctrl+D(w systemach Unixowych) lub Ctrl+Z(w Windows), która wyśle do programu informację, że skończyliśmy wpisywać dane. Program wyjdzie wtedy z pętli i zamknie utworzony plik.

Rozmiar pliku

Dzięki standardowym funkcjom języka C możemy m.in. określić długość pliku. Do tego celu służą funkcje fsetpos, fgetpos oraz fseek. Ponieważ przy każdym odczycie-/zapisie z/do pliku wskaźnik niejako "przesuwa" się o liczbę przeczytanych/zapisanych bajtów. Możemy jednak ustawić wskaźnik w dowolnie wybranym miejscu. Do tego właśnie służą wyżej wymienione funkcje. Aby odczytać rozmiar pliku powinniśmy ustawić nasz wskaźnik na koniec pliku, po czym odczytać ile bajtów od początku pliku się znajdujemy. Wiem, brzmi to strasznie, ale działa wyjątkowo prosto i skutecznie. Użyjemy do tego tylko dwóch funkcji: fseek oraz fgetpos. Pierwsza służy do ustawiania wskaźnika na odpowiedniej pozycji w pliku, a druga do odczytywania na którym bajcie pliku znajduje się wskaźnik. Kod, który określa rozmiar pliku znajduje się tutaj:

```
#include <stdio.h>

int main (int argc, char **argv)
{
  FILE *fp = NULL;
  fpos_t dlugosc;
  if (argc != 2) {
    printf ("Użycie: %s <nazwa pliku>\n", argv[0]);
    return 1;
    }
  if ((fp=fopen(argv[1], "rb"))==NULL) {
    printf ("Błąd otwarcia pliku: %s!\n", argv[1]);
    return 1;
    }
  fseek (fp, 0, SEEK_END); /* ustawiamy wskaźnik na koniec pliku */
  fgetpos (fp, &dlugosc);
  printf ("Rozmiar pliku: %d\n", dlugosc);
  fclose (fp);
  return 0;
}
```

Znajomość rozmiaru pliku przydaje się w wielu różnych sytuacjach, więc dobrze przeanalizuj przykład!

Przykład — pliki graficzny

Najprostszym przykładem rastrowego pliku graficznego jest plik PPM. Poniższy program pokazuje jak utworzyć plik w katalogu roboczym programu. Do zapisu :

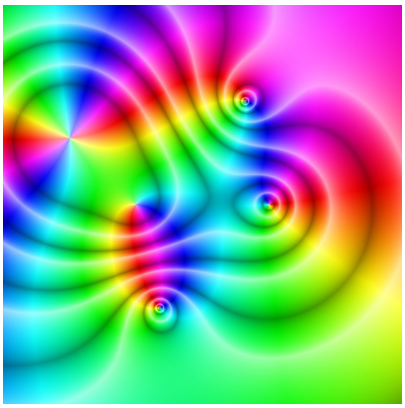
- nagłówka pliku używana jest funkcja fprintf,
- tablicy do pliku używana jest funkcja fwrite.

```
#include <stdio.h>
 int main() {
        const int dimx = 800;
        const int dimy = 800;
        int i, j;
```

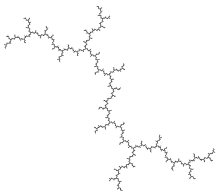
```
        FILE * fp = fopen("first.ppm", "wb"); /* b - tryb binarny */
        fprintf(fp, "P6\n%d %d\n255\n", dimx, dimy);
        for(j=0; j<dimy; ++j){
          for(i=0; i<dimx; ++i){
                        static unsigned char color[3];
                        color[0]=i % 255; /* red */
                        color[1]=j % 255; /* green */
                        color[2]=(i*j) % 255; /* blue */
                        fwrite(color,1,3,fp);
                }
        }
        fclose(fp);
        return 0;
}
```

W powyższym przykładzie dostęp do danych jest **sekwencyjny**. Jeśli chcemy mieć **swobodny** dostęp do danych to :

- korzystać z funkcji: fsetpos, fgetpos oraz fseek,
- utworzyć tablicę (dla dużych plików dynamiczną), zapisać do niej wszystkie dane a następnie zapisać całą tablicę do pliku. Ten sposób jest prostszy i szybszy. Należy zwrócić uwagę, że do obliczania rozmiaru całej tablicy nie możemy użyć funkcji sizeof.



(a) Przykład użycia tej techniki, sekwencyjny dostęp do danych (kod źródłowy)



(b) Przykład użycia tej techniki, swobodny dostęp do danych (kod źródłowy)

Co z katalogami?

Faktycznie, zapomnieliśmy o nich. Jednak wynika to z tego, że specyfikacja ANSI C nie uwzględnia obsługi katalogów.

Rozdział 15

Ćwiczenia dla początkujących

Ćwiczenia

Wszystkie, zamieszczone tutaj ćwiczenia mają na celu pomóc Ci w sprawdzeniu Twojej wiedzy oraz umożliwieniu Tobie wykorzystania nowo nabytych wiadomości w praktyce. Pamiętaj także, że ten podręcznik ma służyć także innym, więc nie zamieszczaj tutaj Twoich rozwiązań. Zachowaj je dla siebie.

Ćwiczenie 1

Napisz program, który wyświetli na ekranie twoje imię i nazwisko.

Ćwiczenie 2

Napisz program, który poprosi o podanie dwóch liczb rzeczywistych i wyświetli wynik mnożenia obu zmiennych.

Ćwiczenie 3

Napisz program, który pobierze jako argumenty z linii komend nazwy dwóch plików i przekopiuje zawartość pierwszego pliku do drugiego (tworząc lub zamazując drugi).

Ćwiczenie 4

Napisz program, który utworzy nowy plik (o dowolnie wybranej przez Ciebie nazwie) i zapisze tam:

1. Twoje imię
2. wiek
3. miasto, w którym mieszkasz

Przykładowy plik powinien wyglądać tak:

```
Stanisław
30
Kraków
```

Ćwiczenie 5

Napisz program generujący tabliczkę mnożenia 10 x 10 i wyświetlający ją na ekranie.

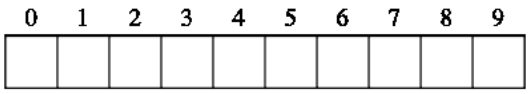
Ćwiczenie 6

Napisz program znajdujący pierwiastki trójmianu kwadratowego $ax^2+bx+c=0$, dla zadanych parametrów a, b, c.

Rozdział 16

Tablice

W rozdziale Zmienne w C dowiedziałeś się, jak przechowywać pojedyncze liczby oraz znaki. Czasami zdarza się jednak, że potrzebujemy przechować kilka, kilkanaście albo i więcej zmiennych jednego typu. Nie tworzymy wtedy np. dwudziestu osobnych zmiennych. W takich przypadkach z pomocą przychodzi nam **tablica**.



Rysunek 16.1: tablica 10-elementowa

Tablica to ciąg zmiennych jednego typu. Ciąg taki posiada jedną nazwę a do jego poszczególnych elementów odnosimy się przez numer (indeks).

Wstęp

Sposoby deklaracji tablic

Tablicę deklaruje się w następujący sposób:

```
typ nazwa_tablicy[rozmiar];
```

gdzie rozmiar oznacza ile zmiennych danego typu możemy zmieścić w tablicy. Zatem aby np. zadeklarować tablicę, mieszczącą 20 liczb całkowitych możemy napisać tak:

```
int tablica[20];
```

Podobnie jak przy deklaracji zmiennych, także tablicy możemy nadać wartości początkowe przy jej deklaracji. Odbywa się to przez umieszczenie wartości kolejnych elementów oddzielonych przecinkami wewnątrz nawiasów klamrowych:

```
int tablica[3] = {1,2,3};
```

Może to się wydać dziwne, ale po ostatnim elemencie tablicy może występować przecinek. Ponadto, jeżeli poda się tylko część wartości, w pozostałe wpisywane są zera:

```
int tablica[20] = {1,};
```

Niekoniecznie trzeba podawać rozmiar tablicy, np.:

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

W takim przypadku kompilator sam ustali rozmiar tablicy (w tym przypadku — 5 elementów).

Rozpatrzmy następujący kod:

```
#include <stdio.h>
#define ROZMIAR 3
int main()
{
  int tab[ROZMIAR] = {3,6,8};
  int i;
  printf ("Druk tablicy tab:\n");

  for (i=0; i<ROZMIAR; ++i) {
    printf ("Element numer %d = %d\n", i, tab[i]);
  }
  return 0;
}
```

Wynik:

```
Druk tablicy tab:
Element numer 0 = 3
Element numer 1 = 6
Element numer 2 = 8
```

Jak widać, wszystko się zgadza. W powyżej zamieszczonym przykładzie użyliśmy stałej do podania rozmiaru tablicy. Jest to o tyle pożądany zwyczaj, że w razie konieczności zmiany rozmiaru tablicy zmieniana jest tylko jedna linijka kodu przy stałej, a nie kilkadziesiąt innych linijek, rozsianych po kodzie całego programu.

W pierwotnym standardzie języka C rozmiar tablicy nie mógł być określany przez zmienną lub nawet stałą zadeklarowaną przy użyciu słowa kluczowego const. Dopiero w późniejszej wersji standardu (tzw. C99) dopuszczono taką możliwość. Dlatego do deklarowania rozmiaru tablic często używa się dyrektywy preprocesora #define. Powinni na to zwrócić uwagę zwłaszcza programiści C++, gdyż tam zawsze możliwe były oba sposoby.

Innym sposobem jest użycie operatora sizeof do poznania wielkości tablicy. Poniższy kod robi to samo co przedstawiony:

```
#include <stdio.h>
int main()
{
  int tab[3] = {3,6,8};
```

```
  int i;
  printf ("Druk tablicy tab:\n");

  for (i=0; i<(sizeof tab / sizeof *tab); ++i) {
    printf ("Element numer %d = %d\n", i, tab[i]);
  }
  return 0;
}
```

Należy pamiętać, że działa on tylko dla tablic, a nie wskaźników (jak później się dowiesz wskaźnik też można w pewnym stopniu traktować jak tablicę).

Odczyt/zapis wartości do tablicy

Z tablicami posługujemy się tak samo jak ze zwykłymi zmiennymi. Różnica polega jedynie na podaniu **indeksu** tablicy. Określa on jednoznacznie, z którego elementu (wartości) chcemy skorzystać. Indeksem jest liczba naturalna począwszy od zera. To oznacza, że pierwszy element tablicy ma indeks równy 0, drugi 1, trzeci 2, itd.

Osoby, które wcześniej programowały w językach, takich jak Pascal, Basic czy Fortran muszą przyzwyczaić się do tego, że w języku C indeks numeruje się od 0.

Spróbujmy przedstawić to na działającym przykładzie. Przeanalizuj następujący kod:

```
int tablica[5] = {0};
int i = 0;
tablica[2] = 3;
tablica[3] = 7;
for (i=0;i!=5;++i) {
  printf ("tablica[%d]=%d\n", i, tablica[i]);
}
```

Jak widać, na początku deklarujemy 5-elementową tablicę, którą od razu zerujemy. Następnie pod trzeci i czwarty element podstawiamy liczby 3 i 7. Pętla ma za zadanie wyprowadzić wynik naszych działań.

Tablice znaków

Tablice znaków tj. typu char oraz unsigned char posiadają dwie ogólnie przyjęte nazwy, zależnie od ich przeznaczenia:

- bufory — gdy wykorzystujemy je do przechowywania ogólnie pojętych danych, gdy traktujemy je jako po prostu "ciągi bajtów" (typ char ma rozmiar 1 bajta, więc jest elastyczny do przechowywania np. danych wczytanych z pliku przed ich przetworzeniem).
- napisy — gdy zawarte w nich dane traktujemy jako ciągi liter; jest im poświęcony osobny rozdział Napisy.

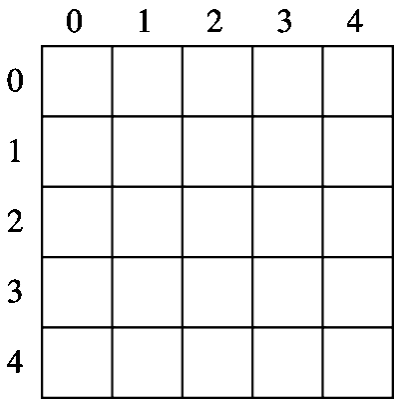
Tablice wielowymiarowe

Rozważmy teraz konieczność przechowania w pamięci komputera całej macierzy o wymiarach 10 x 10. Można by tego dokonać tworząc 10 osobnych tablic jednowymiarowych, reprezentujących poszczególne wiersze macierzy. Jednak język C dostarcza nam dużo wygodniejszej metody, która w dodatku jest bardzo łatwa w użyciu. Są to **tablice wielowymiarowe**, lub inaczej "tablice tablic". Tablice wielowymiarowe definiujemy podając przy zmiennej kilka wymiarów, np.:

```
float macierz[10][10];
```

Tak samo wygląda dostęp do poszczególnych elementów tablicy:

```
macierz[2][3] = 1.2;
```



Rysunek 16.2: tablica dwuwymiarowa (5x5)

Jak widać ten sposób jest dużo wygodniejszy (i zapewne dużo bardziej "naturalny") niż deklarowanie 10 osobnych tablic jednowymiarowych. Aby zainicjować tablicę wielowymiarową należy zastosować zagłębianie klamer, np.:

```
float macierz[3][4] = {
  { 1.6, 4.5, 2.4, 5.6 },  /* pierwszy wiersz */
  { 5.7, 4.3, 3.6, 4.3 },  /* drugi wiersz */
  { 8.8, 7.5, 4.3, 8.6 }   /* trzeci wiersz */
};
```

Dodatkowo, pierwszego wymiaru nie musimy określać (podobnie jak dla tablic jednowymiarowych) i wówczas kompilator sam ustali odpowiednią wielkość, np.:

```
float macierz[][4] = {
  { 1.6, 4.5, 2.4, 5.6 },  /* pierwszy wiersz */
  { 5.7, 4.3, 3.6, 4.3 },  /* drugi wiersz */
  { 8.8, 7.5, 4.3, 8.6 },  /* trzeci wiersz */
  { 6.3, 2.7, 5.7, 2.7 },  /* czwarty wiersz */
};
```

Innym, bardziej elastycznym sposobem deklarowania tablic wielowymiarowych jest użycie wskaźników. Opisane to zostało w następnym rozdziale.

Ograniczenia tablic

Pomimo swej wygody tablice mają ograniczony, z góry zdefiniowany rozmiar, którego nie można zmienić w trakcie działania programu. Dlatego też w niektórych zastosowaniach tablice zostały wyparte przez dynamiczną alokację pamięci. Opisane to zostało w następnym rozdziale.

Przy używaniu tablic trzeba być szczególnie ostrożnym przy konstruowaniu pętli, ponieważ ani kompilator, ani skompilowany program nie będą w stanie wychwycić przekroczenia przez indeks rozmiaru tablicy [1]. Efektem będzie odczyt lub zapis pamięci, znajdującej się poza tablicą.

Wystarczy pomylić się o jedno miejsce (tzw. błąd off by one) by spowodować, że działanie programu zostanie nagle przerwane przez system operacyjny:

```
int foo[100];
int i;

for (i=0; i<=100; ++i) /* powinno być i<100 */
  foo[i] = 0;
```

Ciekawostki

W pierwszej edycji konkursu IOCCC zwyciężył program napisany w C, który wyglądał dość nietypowo:

```
short main[] = {
  277, 04735, -4129, 25, 0, 477, 1019, 0xbef, 0, 12800,
  -113, 21119, 0x52d7, -1006, -7151, 0, 0x4bc, 020004,
  14880, 10541, 2056, 04010, 4548, 3044, -6716, 0x9,
  4407, 6, 5568, 1, -30460, 0, 0x9, 5570, 512, -30419,
  0x7e82, 0760, 6, 0, 4, 02400, 15, 0, 4, 1280, 4, 0,
  4, 0, 0, 0, 0x8, 0, 4, 0, ',', 0, 12, 0, 4, 0, '#',
  0, 020, 0, 4, 0, 30, 0, 026, 0, 0x6176, 120, 25712,
  'p', 072163, 'r', 29303, 29801, 'e'
};
```

Co ciekawe — program ten bez przeszkód wykonywał się na komputerach VAX-11 oraz PDP-11. Cały program to po prostu tablica z zawartym wewnątrz kodem maszynowym! Tak naprawdę jest to wykorzystanie pewnych właściwości programu, który ostatecznie produkuje kod maszynowy. Linker (to o nim mowa) nie rozróżnia na dobrą sprawę nazw funkcji od nazw zmiennych, więc bez problemu ustawił punkt wejścia programu na tablicę wartości, w których zapisany był kod maszynowy. Tak przygotowany program został bez problemu wykonany przez komputer.

[1] W zasadzie kompilatory mają możliwość dodania takiego sprawdzania, ale nie robi się tego, gdyż znacznie spowolniłoby to działanie programu. Takie postępowanie jest jednak pożądane w okresie testowania programu.

Rozdział 17

Wskaźniki

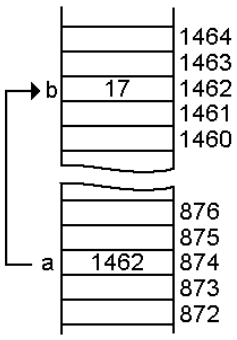
Zobacz w Wikipedii: Zmienna wskaźnikowa

Zmienne w komputerze są przechowywane w pamięci. To wie każdy programista, a dobry programista potrafi kontrolować zachowanie komputera w przydzielaniu i obsługi pamięci dla zmiennych. W tym celu pomocne są **wskaźniki**.

Co to jest wskaźnik?

Dla ułatwienia przyjęto poniżej, że bajt ma 8 bitów, typ int składa się z dwóch bajtów (16 bitów), typ long składa się z czterech bajtów (32 bitów) oraz liczby zapisane są w formacie big endian (tzn. bardziej znaczący bajt na początku), co niekoniecznie musi być prawdą na Twoim komputerze.

Wskaźnik (ang. pointer) to specjalny rodzaj zmiennej, w której zapisany jest adres w pamięci komputera, tzn. wskaźnik **wskazuje** miejsce, gdzie zapisana jest jakaś informacja. Oczywiście nic nie stoi na przeszkodzie aby wskazywaną daną był inny wskaźnik do kolejnego miejsca w pamięci.



Rysunek 17.1: Wskaźnik a wskazujący na zmienną b. Zauważmy, że b przechowuje liczbę, podczas gdy a przechowuje adres b w pamięci (1462)

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: kilka rewersów może dotyczyć tej samej książki, numer w rewersie możemy skreślić i użyć go do zamówienia innej książki, jeśli wpiszemy nieprawidłowy numer katalogowy to możemy dostać nie tą książkę, którą chcemy, albo też nie dostać nic.

Warto też poznać w tym miejscu definicję **adresu pamięci**. Możemy powiedzieć, że adres to pewna liczba całkowita, jednoznacznie definiująca położenie pewnego obiektu

(czyli np. znaku czy liczby) w pamięci komputera. Dokładniejszą definicję możesz znaleźć w Wikipedii.

Operowanie na wskaźnikach

By stworzyć wskaźnik do zmiennej i móc się nim posługiwać należy przypisać mu odpowiednią wartość (adres obiektu, na jaki ma wskazywać). Skąd mamy znać ten adres? Wystarczy zapytać nasz komputer, jaki adres przydzielił zmiennej, którą np. wcześniej gdzieś stworzyliśmy. Robi się to za pomocą operatora **&** (operatora pobrania adresu). Przeanalizuj następujący kod[1]:

```
#include <stdio.h>

int main (void)
{
  int liczba = 80;
  printf("Zmienna znajduje sie pod adresem: %p, i przechowuje wartosc: %d\n",
         (void*)&liczba, liczba);
  return 0;
}
```

Program ten wypisuje adres pamięci, pod którym znajduje się zmienna oraz wartość jaką kryje zmienna przechowywana pod owym adresem.

Aby móc zapisać gdzieś taki adres należy zadeklarować zmienną wskaźnikową. Robi się to poprzez dodanie * (gwiazdki) po typie na jaki zmienna ma wskazywać, np.:

```
int *wskaznik1;
char *wskaznik2;
float*wskaznik3;
```

Niektórzy programiści mogą nieco błędnie interpretować wskaźnik do typu jako nowy typ i uważać, że jeśli napiszą:

```
int* a,b,c;
```

to otrzymają trzy wskaźniki do liczby całkowitej. Tymczasem wskaźnikiem będzie tylko zmienna a, natomiast b i c będą po prostu liczbami. Powodem jest to, że "gwiazdka" odnosi się **do zmiennej** a nie do typu. W tym przypadku trzy wskaźniki otrzymamy pisząc:

```
int *a,*b,*c;
```

Aby uniknąć pomyłek, lepiej jest pisać gwiazdkę tuż przy zmiennej:

```
 int *a,b,c;
```

albo jeszcze lepiej nie mieszać deklaracji wskaźników i zmiennych:

```
 int *a;
 int b,c;
```

[1] Warto zwrócić uwagę na rzutowanie do typu *wskaźnik na void*. Rzutowanie to jest wymagane przez funkcję **printf**, gdyż ta oczekuje, że argumentem dla formatu `%p` będzie właśnie *wskaźnik na void*, gdy tymczasem w naszym przykładzie wyrażenie `&liczba` jest typu *wskaźnik na int*.

Aby dobrać się do wartości wskazywanej przez zmienną należy użyć unarnego operatora ***** (gwiazdka), zwanego **operatorem wyłuskania**:

```
#include <stdio.h>

int main (void)
{
  int liczba = 80;
  int *wskaznik = &liczba;
  printf("Wartosc zmiennej: %d; jej adres: %p.\n", liczba, (void*)&liczba);
  printf("Adres zapisany we wskazniku: %p, wskazywana wartosc: %d.\n",
         (void*)wskaznik, *wskaznik);

  *wskaznik = 42;
  printf("Wartosc zmiennej: %d, wartosc wskazywana przez wskaznik: %d\n",
         liczba, *wskaznik);

  liczba = 0x42;
  printf("Wartosc zmiennej: %d, wartosc wskazywana przez wskaznik: %d\n",
         liczba, *wskaznik);

  return 0;
}
```

O co chodzi z tym typem, na który ma wskazywać? Czemu to takie ważne?

Jest to ważne z kilku powodów.

Różne typy zajmują w pamięci różną wielkość. Przykładowo, jeżeli w zmiennej typu **unsigned int** zapiszemy liczbę **65 530**, to w pamięci będzie istnieć jako:

```
+--------+--------+
|komórka1|komórka2|
+--------+--------+
|11111111|11111010| = (unsigned int) 65530
+--------+--------+
```

Wskaźnik do takiej zmiennej (jak i do dowolnej innej) będzie wskazywać na pierwszą komórkę, w której ta zmienna ma swoją wartość.

Jeżeli teraz stworzymy drugi wskaźnik do tego adresu, tym razem typu **unsigned char***, to wskaźnik przejmie ten adres prawidłowo[2], lecz gdy spróbujemy odczytać wartość na jaką wskazuje ten wskaźnik to zostanie odczytana tylko pierwsza komórka i wynik będzie równy **255**:

[2]Tak naprawdę nie zawsze można przypisywać wartości jednych wskaźników do innych. Standard C gwarantuje jedynie, że można przypisać wskaźnikowi typu void* wartość dowolnego wskaźnika, a następnie przypisać tą wartość do wskaźnika pierwotnego typu oraz, że dowolny wskaźnik można przypisać do wskaźnika typu char*.

```
+--------+
|komórka1|
+--------+
|11111111| = (unsigned char) 255
+--------+
```

Gdybyśmy natomiast stworzyli inny wskaźnik do tego adresu tym razem typu **unsigned long*** to przy próbie odczytu odczytane zostaną dwa bajty z wartością zapisaną w zmiennej **unsigned int** oraz dodatkowe dwa bajty z niewiadomą zawartością i wówczas wynik będzie równy **65530 * 65536 + *losowa wartość*** :

```
+--------+--------+--------+--------+
|komórka1|komórka2|komórka3|komórka4|
+--------+--------+--------+--------+
|11111111|11111010|????????|????????|
+--------+--------+--------+--------+
```

Ponadto, zapis czy odczyt poza przydzielonym obszarem pamięci może prowadzić do nieprzyjemnych skutków takich jak zmiana wartości innych zmiennych czy wręcz natychmiastowe przerwanie programu. Jako przykład można podać ten (błędny) program[3]:

```
#include <stdio.h>

int main(void)
{
  unsigned char tab[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  unsigned short *ptr= (unsigned short*)&tab[2];
  unsigned i;

  *ptr = 0xffff;
  for (i = 0; i < 9; ++i) {
    printf("%d ", tab[i]);
  }
  printf("%d\n", tab[9]);
  return 0;
}
```

Nie można również zapominać, że na niektórych architekturach dane wielobajtowe muszą być odpowiednio wyrównane w pamięci. Np. zmienna dwubajtowa może się znajdować jedynie pod parzystymi adresami. Wówczas, gdybyśmy chcieli adres zmiennej jednobajtowej przypisać wskaźnikowi na zmienną dwubajtową mogłoby dojść do nieprzewidzianych błędów wynikających z próby odczytu niewyrównanej danej.

Zaskakujące może się okazać, że różne wskaźniki mogą mieć różny rozmiar. Np. *wskaźnik na char* może być większy od *wskaźnika na int*, ale również na odwrót. Co więcej, wskaźniki różnych typów mogą się różnić reprezentacją adresów. Dla przykładu *wskaźnik na char* może przechowywać adres do bajtu natomiast *wskaźnik na int* ten adres podzielony przez 2.

[3]Może się okazać, że błąd nie będzie widoczny na Twoim komputerze.

Podsumowując, różne wskaźniki to różne typy i nie należy beztrosko rzutować wyrażeń pomiędzy różnymi typami wskaźnikowymi, bo grozi to nieprzewidywalnymi błędami.

Do czego służy typ void*?

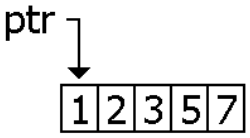
Czasami zdarza się, że nie wiemy, na jaki typ wskazuje dany wskaźnik. W takich przypadkach stosujemy typ void*. Sam void nie znaczy nic, natomiast void* oznacza "wskaźnik na obiekt w pamięci niewiadomego typu". Taki wskaźnik możemy potem odnieść do konkretnego typu danych (w języku C++ wymagane jest do tego rzutowania). Na przykład, funkcja malloc zwraca właśnie wskaźnik za pomocą void*.

Arytmetyka wskaźników

W języku C do wskaźników można dodawać lub odejmować liczby całkowite. Istotne jest jednak, że dodanie do wskaźnika liczby dwa nie spowoduje przesunięcia się w pamięci komputera o dwa bajty. Tak naprawdę przesuniemy się o 2*rozmiar zmiennej. Jest to bardzo ważna informacja! Początkujący programiści popełniają często dużo błędów, związanych z nieprawidłową arytmetyką wskaźników.

Zobaczmy na przykład:

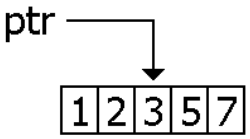
```
int *ptr;
int a[] = {1, 2, 3, 5, 7};
ptr = &a[0];
```



Rysunek 17.2: Wskaźnik wskazuje na pierwszą komórkę pamięci

Otrzymujemy następującą sytuację:
Gdy wykonamy

```
ptr += 2;
```



Rysunek 17.3: Przesunięcie wskaźnika na kolejne komórki

wskaźnik ustawi się na trzecim elemencie tablicy.

Wskaźniki można również od siebie odejmować, czego wynikiem jest *odległość* dwóch wskazywanych wartości. Odległość zwracana jest jako liczba obiektów danego typu, a nie liczba bajtów. Np.:

```
int a[] = {1, 2, 3, 5, 7};
int *ptr = &a[2];
int diff = ptr - a;  /* diff ma wartość 2 (a nie 2*sizeof(int)) */
```

Wynikiem może być oczywiście liczba ujemna. Operacja jest przydatna do obliczania wielkości tablicy (długości łańcucha znaków) jeżeli mamy wskaźnik na jej pierwszy i ostatni element.

Operacje arytmetyczne na wskaźnikach mają pewne ograniczenia. Przede wszystkim nie można (tzn. standard tego nie definiuje) skonstruować wskaźnika wskazującego gdzieś poza zadeklarowaną tablicę, chyba, że jest to obiekt zaraz za ostatnim (*one past last*), np.:

```
int a[] = {1, 2, 3, 5, 7};
int *ptr;
ptr = a + 10; /* niezdefiniowane */
ptr = a - 10; /* niezdefiniowane */
ptr = a + 5;  /* zdefiniowane (element za ostatnim) */
*ptr = 10;    /* to już nie! */
```

Nie można[4] również odejmować od siebie wskaźników wskazujących na obiekty znajdujące się w różnych tablicach, np.:

```
int a[] = {1, 2, 3}, b[] = {5, 7};
int *ptr1 = a, *ptr2 = b;
int diff = a - b; /* niezdefiniowane */
```

Tablice a wskaźniki

Trzeba wiedzieć, że tablice to też rodzaj zmiennej wskaźnikowej. Taki wskaźnik wskazuje na miejsce w pamięci, gdzie przechowywany jest jej pierwszy element. Następne elementy znajdują się bezpośrednio w następnych komórkach pamięci, w odstępie zgodnym z wielkością odpowiedniego typu zmiennej.

Na przykład tablica:

```
int tab[] = {100,200,300};
```

występuje w pamięci w sześciu komórkach [5]:

```
+--------+--------+--------+--------+--------+--------+
|wartosc1|        |wartosc2|        |wartosc3|        |
+--------+--------+--------+--------+--------+--------+
|00000000|01100100|00000000|11001000|00000001|00101100|
+--------+--------+--------+--------+--------+--------+
```

[4] To znaczy standard nie definiuje co się wtedy stanie, aczkolwiek na większości architektur odejmowanie dowolnych dwóch wskaźników ma zdefiniowane zachowanie. Pisząc przenośne programy nie można jednak na tym polegać, zwłaszcza, że odejmowanie wskaźników wskazujących na elementy różnych tablic zazwyczaj nie ma sensu.

[5] Ponownie przyjmując, że bajt ma 8 bitów, int dwa bajty i liczby zapisywane są w formacie little endian

Stąd do trzeciej wartości można się dostać tak (komórki w tablicy numeruje się od zera):

```
zmienna = tab[2];
```

albo wykorzystując metodę wskaźnikową:

```
zmienna = *(tab + 2);
```

Z definicji obie te metody są równoważne.

Z definicji (z wyjątkiem użycia operatora sizeof) wartością zmiennej lub wyrażenia typu tablicowego jest wskaźnik na jej pierwszy element (`tab == &tab0`).

Co więcej, można pójść w drugą stronę i potraktować wskaźnik jak tablicę:

```
int *wskaznik;
wskaznik = &tab[1]; /* lub wskaznik = tab + 1; */
zmienna = wskaznik[1]; /* przypisze 300 */
```

Jako ciekawostkę podamy, iż w języku C można odnosić się do elementów tablicy jeszcze w inny sposób:

```
printf ("%d\n", 1[tab]);
```

Skąd ta dziwna notacja? Uzasadnienie jest proste:

```
tab[1] = *(tab + 1) = *(1 + tab) = 1[tab]
```

Podobną składnię stosuje m.in. asembler GNU.

Gdy argument jest wskaźnikiem...

Czasami zdarza się, że argumentem (lub argumentami) funkcji są wskaźniki. W przypadku "normalnych" zmiennych nasza funkcja działa tylko na lokalnych kopiach tychże argumentów, natomiast nie zmienia zmiennych, które zostały podane jako argument. Natomiast w przypadku wskaźnika, każda operacja na wartości wskazywanej powoduje zmianę wartości zmiennej zewnętrznej. Spróbujmy rozpatrzeć poniższy przykład:

```
#include <stdio.h>

void func (int *zmienna)
{
  *zmienna = 5;
}

int main ()
{
  int z=3;
  printf ("z=%d\n", z); /* wypisze 3 */
  func(&z);
  printf ("z=%d\n", z); /* wypisze 5 */
}
```

Widzimy, że funkcje w języku C nie tylko potrafią zwracać określoną wartość, lecz także zmieniać dane, podane im jako argumenty. Ten sposób przekazywania argumentów do funkcji jest nazywany *przekazywaniem przez wskaźnik* (w przeciwieństwie do normalnego *przekazywania przez wartość*).

Zwróćmy uwagę na wywołanie `func(&z);`. Należy pamiętać, by do funkcji przekazać adres zmiennej a nie samą zmienną. Jeśli byśmy napisali `func(z);` to funkcja starałaby się zmienić komórkę pamięci o numerze 3. Kompilator powinien ostrzec w takim przypadku o konwersji z typu int do wskaźnika, ale często kompiluje taki program pozostając na ostrzeżeniu.

Nie gra roli czy przy deklaracji funkcji jako argument funkcji podamy wskaźnik czy tablicę (z podanym rozmiarem lub nie), np. poniższe deklaracje są identyczne:

```
void func(int ptr[]);
void func(int *ptr);
```

Można przyjąć konwencję, że deklaracja określa czy funkcji przekazujemy wskaźnik do pojedynczy argument czy do sekwencji, ale równie dobrze można za każdym razem stosować gwiazdkę.

Pułapki wskaźników

Ważne jest, aby przy posługiwaniu się wskaźnikami nigdy nie próbować odwoływać się do komórki wskazywanej przez wskaźnik o wartości NULL lub niezainicjowany wskaźnik! Przykładem nieprawidłowego kodu, może być np.:

```
int *wsk;
printf ("zawartosc komorki: %d\n", *(wsk)); /* Błąd */
wsk = 0; /* 0 w kontekście wskaźników oznacza wskaźnik NULL */
printf ("zawartosc komorki: %d\n", *(wsk)); /* Błąd */
```

Należy również uważać, aby nie odwoływać się do komórek poza przydzieloną pamięcią, np.:

```
int tab[] = { 0, 1, 2 };
tab[3] = 3;  /* Błąd */
```

Pamiętaj też, że możesz być rozczarowany używając operatora sizeof, podając zmienną wskaźnikową. Uzyskana wielkość będzie wielkością wskaźnika, a nie wielkością typu użytego podczas deklarowania naszego wskaźnika. Wielkość ta będzie zawsze miała taki sam rozmiar dla każdego wskaźnika, w zależności od kompilatora, a także docelowej platformy. Zamiast tego używaj: *sizeof(*wskaźnik)*. Przykład:

```
 char *zmienna;
 int a = sizeof zmienna; /* a wynosi np. 4, tj. sizeof(char*) */
 a = sizeof(char*);      /* robimy to samo, co wyżej */
 a = sizeof *zmienna;    /* zmienna a ma teraz przypisany rozmiar
                            pojedyńczego znaku, tj. 1 */
 a = sizeof(char);       /* robimy to samo, co wyżej */
```

Na co wskazuje NULL?

Analizując kody źródłowe programów często można spotkać taki oto zapis:

```
void *wskaznik = NULL; /* lub = 0 */
```

Wiesz już, że nie możemy odwołać się pod komórkę pamięci wskazywaną przez wskaźnik NULL. Po co zatem przypisywać wskaźnikowi 0? Odpowiedź może być zaskakująca: właśnie po to, aby uniknąć błędów! Wydaje się to zabawne, ale większość (jeśli nie wszystkie) funkcje, które zwracają wskaźnik w przypadku błędu zwrócą właśnie NULL, czyli zero. Tutaj rodzi się kolejna wskazówka: jeśli w danej zmiennej przechowujemy wskaźnik, zwrócony wcześniej przez jakąś funkcję zawsze sprawdzajmy, czy nie jest on równy 0 (NULL). Wtedy mamy pewność, że funkcja zadziałała poprawnie.

Dokładniej, NULL nie jest słowem kluczowym, lecz stałą (makrem) zadeklarowaną przez dyrektywy preprocesora. Deklaracja taka może być albo wartością 0 albo też wartością 0 zrzutowaną na void* (`((void *)0)`), ale też jakimś słowem kluczowym deklarowanym przez kompilator.

Warto zauważyć, że pomimo przypisywania wskaźnikowi zera, nie oznacza to, że wskaźnik NULL jest reprezentowany przez same zerowe bity. Co więcej, wskaźniki NULL różnych typów mogą mieć różną wartość! Z tego powodu poniższy kod jest niepoprawny:

```
int **tablica_wskaznikow = calloc(100, sizeof *tablica_wskaznikow);
```

Zakłada on, że w reprezentacji wskaźnika NULL występują same zera. Poprawnym zainicjowaniem dynamicznej tablicy wskaźników wartościami NULL jest (pomijamy sprawdzanie wartości zwróconej przez malloc()):

```
int **tablica_wskaznikow = malloc(100 * sizeof *tablica_wskaznikow);
int i = 0;
while (i<100)
  tablica_wskaznikow[i++] = 0;
```

Stałe wskaźniki

Tak, jak istnieją zwykłe stałe, tak samo możemy mieć stałe wskaźniki — jednak są ich dwa rodzaje. Wskaźniki na stałą wartość:

```
const int *a; /* lub równoważnie */
int const *a;
```

oraz stałe wskaźniki:

```
int * const b;
```

Pierwszy to wskaźnik, którym nie można zmienić wskazywanej wartości. Drugi to wskaźnik, którego nie można przestawić na inny adres. Dodatkowo, można zadeklarować stały wskaźnik, którym nie można zmienić wartości wskazywanej zmiennej, i również można zrobić to na dwa sposoby:

```
const int * const c; /* alternatywnie */
int const * const c;
```

```
int i=0;
const int *a=&i;
int * const b=&i;
int const * const c=&i;
*a = 1;  /* kompilator zaprotestuje */
*b = 2;  /* ok */
*c = 3   /* kompilator zaprotestuje */
a = b;   /* ok */
b = a;   /* kompilator zaprotestuje */
c = a;   /* kompilator zaprotestuje */
```

Wskaźniki na stałą wartość są przydatne między innymi w sytuacji gdy mamy duży obiekt (na przykład strukturę z kilkoma polami). Jeśli przypiszemy taką zmienną do innej zmiennej, kopiowanie może potrwać dużo czasu, a oprócz tego zostanie zajęte dużo pamięci. Przekazanie takiej struktury do funkcji albo zwrócenie jej jako wartość funkcji wiąże się z takim samym narzutem. W takim wypadku dobrze jest użyć wskaźnika na stałą wartość.

```
void funkcja(const duza_struktura *ds)
{
   /* czytamy z ds i wykonujemy obliczenia */
}

funkcja(&dane); /* mamy pewność, że zmienna dane nie zostanie zmieniona */
```

Dynamiczna alokacja pamięci

Mając styczność z tablicami można się zastanowić, czy nie dałoby się mieć tablic, których rozmiar dostosowuje się do naszych potrzeb a nie jest na stałe zaszyty w kodzie programu. Chcąc pomieścić więcej danych możemy po prostu zwiększyć rozmiar tablicy — ale gdy do przechowania będzie mniej elementów okaże się, że marnujemy pamięć. Język C umożliwia dzięki wskaźnikom i dynamicznej alokacji pamięci tworzenie tablic takiej wielkości, jakiej akurat potrzebujemy.

O co chodzi

Czym jest dynamiczna alokacja pamięci? Normalnie zmienne programu przechowywane są na tzw. stosie (ang. *stack*) — powstają, gdy program wchodzi do bloku, w którym zmienne są zadeklarowane a zwalniane w momencie, kiedy program opuszcza ten blok. Jeśli deklarujemy tak tablice, to ich rozmiar musi być znany w momencie kompilacji — żeby kompilator wygenerował kod rezerwujący odpowiednią ilość pamięci. Dostępny jest jednak drugi rodzaj rezerwacji (czyli alokacji) pamięci. Jest to alokacja na stercie (ang. *heap*). Sterta to obszar pamięci wspólny dla całego programu, przechowywane są w nim zmienne, których czas życia nie jest związany z poszczególnymi blokami. Musimy sami rezerwować dla nich miejsce i to miejsce zwalniać, ale dzięki temu możemy to zrobić w dowolnym momencie działania programu.

Należy pamiętać, że rezerwowanie i zwalnianie pamięci na stercie zajmuje więcej czasu niż analogiczne działania na stosie. Dodatkowo, zmienna zajmuje na stercie więcej miejsca niż na stosie — sterta utrzymuje specjalną strukturę, w której trzymane są wolne partie (może to być np. *lista*). Tak więc używajmy dynamicznej alokacji tam, gdzie jest potrzebna — dla danych, których rozmiaru nie jesteśmy w stanie przewidzieć na etapie kompilacji lub ich żywotność ma być niezwiązana z blokiem, w którym zostały zaalokowane.

Obsługa pamięci

Podstawową funkcją do rezerwacji pamięci jest funkcja malloc. Jest to niezbyt skomplikowana funkcja — podając jej rozmiar (w bajtach) potrzebnej pamięci, dostajemy wskaźnik do zaalokowanego obszaru.

Załóżmy, że chcemy stworzyć tablicę liczb typu float:

```
int rozmiar;
float *tablica;

rozmiar = 3;
tablica = malloc(rozmiar * sizeof *tablica);
tablica[0] = 0.1;
```

Przeanalizujmy teraz po kolei, co dzieje się w powyższym fragmencie. Najpierw deklarujemy zmienne — rozmiar tablicy i wskaźnik, który będzie wskazywał obszar w pamięci, gdzie będzie trzymana tablica. Do zmiennej rozmiar możemy w trakcie działania programu przypisać cokolwiek — wczytać ją z pliku, z klawiatury, obliczyć, wylosować — nie jest to istotne. `rozmiar * sizeof *tablica` oblicza potrzebną wielkość tablicy. Dla każdej zmiennej float potrzebujemy tyle bajtów, ile zajmuje ten typ danych. Ponieważ może się to różnić na rozmaitych maszynach, istnieje operator **sizeof**, zwracający dla danego wyrażenia rozmiar jego typu w bajtach.

W wielu książkach (również K&Rv2) i w Internecie stosuje się inny schemat użycia funkcji malloc a mianowicie: `tablica = (float*)malloc(rozmiar * sizeof(float))`. Takie użycie należy traktować jako błędne, gdyż nie sprzyja ono poprawnemu wykrywaniu błędów.

Rozważmy sytuację, gdy programista zapomni dodać plik nagłówkowy stdlib.h, wówczas kompilator (z braku deklaracji funkcji malloc) przyjmie, że zwraca ona typ int zatem do zmiennej `tablica` (która jest wskaźnikiem) będzie przypisywana liczba całkowita, co od razu spowoduje błąd kompilacji (a przynajmniej ostrzeżenie), dzięki czemu będzie można szybko poprawić kod programu. Rzutowanie jest konieczne tylko w języku C++, gdzie konwersja z `void*` na inne typy wskaźnikowe nie jest domyślna, ale język ten oferuje nowe sposoby alokacji pamięci.

Teraz rozważmy sytuację, gdy zdecydujemy się zwiększyć dokładność obliczeń i zamiast typu float użyć typu double. Będziemy musieli wyszukać wszystkie wywołania funkcji malloc, calloc i realloc odnoszące się do naszej tablicy i zmieniać wszędzie `sizeof(float)` na `sizeof(double)`. Aby temu zapobiec lepiej od razu użyć `sizeof *tablica` (lub jeśli ktoś woli z nawiasami: `sizeof(*tablica)`), wówczas zmiana typu zmiennej `tablica` na `double*` zostanie od razu uwzględniona przy alokacji pamięci.

Dodatkowo, należy sprawdzić, czy funkcja malloc nie zwróciła wartości NULL — dzieje się tak, gdy zabrakło pamięci. Ale uwaga: może się tak stać również jeżeli jako argument funkcji podano zero.

Jeśli dany obszar pamięci nie będzie już nam więcej potrzebny powinniśmy go zwolnić, aby system operacyjny mógł go przydzielić innym potrzebującym procesom. Do zwolnienia obszaru pamięci używamy funkcji `free()`, która przyjmuje tylko jeden argument — wskaźnik, który otrzymaliśmy w wyniku działania funkcji `malloc()`.

```
free (addr);
```

Należy pamiętać o zwalnianiu pamięci — inaczej dojdzie do tzw. wycieku pamięci — program będzie rezerwował nową pamięć ale nie zwracał jej z powrotem i w końcu pamięci może mu zabraknąć.

Należy też uważać, by nie zwalniać dwa razy tego samego miejsca. Po wywołaniu free wskaźnik nie zmienia wartości, pamięć wskazywana przez niego może też nie od razu ulec zmianie. Czasem możemy więc korzystać ze wskaźnika (zwłaszcza czytać) po wywołaniu free nie orientując się, że robimy coś źle — i w pewnym momencie dostać komunikat o nieprawidłowym dostępie do pamięci. Z tego powodu zaraz po wywołaniu funkcji free można przypisać wskaźnikowi wartość 0.

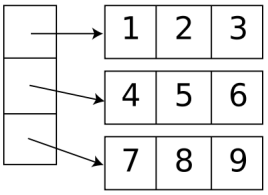
Czasami możemy potrzebować zmienić rozmiar już przydzielonego bloku pamięci. Tu z pomocą przychodzi funkcja realloc:

```
tablica = realloc(tablica, 2*rozmiar*sizeof *tablica);
```

Funkcja ta zwraca wskaźnik do bloku pamięci o pożądanej wielkości (lub NULL gdy zabrakło pamięci). Uwaga — może to być inny wskaźnik. Jeśli zażądamy zwiększenia rozmiaru a za zaalokowanym aktualnie obszarem nie będzie wystarczająco dużo wolnego miejsca, funkcja znajdzie nowe miejsce i przekopiuje tam starą zawartość. Jak widać, wywołanie tej funkcji może być więc kosztowne pod względem czasu.

Ostatnią funkcją jest funkcja calloc(). Przyjmuje ona dwa argumenty: liczbę elementów tablicy oraz wielkość pojedynczego elementu. Podstawową różnicą pomiędzy funkcjami malloc() i calloc() jest to, że ta druga zeruje wartość przydzielonej pamięci (do wszystkich bajtów wpisuje wartość 0).

Tablice wielowymiarowe



Rysunek 17.4: tablica dwuwymiarowa — w rzeczywistości tablica ze wskaźnikami do tablic

W rozdziale Tablice pokazaliśmy, jak tworzyć tablice wielowymiarowe, gdy ich rozmiar jest znany w czasie kompilacji. Teraz zaprezentujemy, jak to wykonać za pomocą wskaźników i to w sytuacji, gdy rozmiar może się zmieniać. Załóżmy, że chcemy stworzyć tabliczkę mnożenia:

```
int rozmiar;
int i;
int **tabliczka;

printf("Podaj rozmiar tabliczki mnozenia: ");
scanf("%i", &rozmiar); /* dla prostoty nie będziemy sprawdzali,
                          czy użytkownik wpisał sensowną wartość */

tabliczka = malloc(rozmiar * sizeof *tabliczka);          /* 1 */
for (i = 0; i<rozmiar; ++i) {                             /* 2 */
   tabliczka[i] = malloc(rozmiar * sizeof **tabliczka);   /* 3 */
}                                                         /* 4 */

for (i = 0; i<rozmiar; ++i) {
  int j;
  for (j = 0; j<rozmiar; ++j) {
    tabliczka[i][j] = (i+1)*(j+1);
  }
}
```

Najpierw musimy przydzielić pamięć — najpierw dla "tablicy tablic" (1) a potem dla każdej z podtablic osobno (2-4). Ponieważ tablica jest typu int* to nasza tablica tablic będzie wskaźnikiem na int* czyli int**. Podobnie osobno, ale w odwrotnej kolejności będziemy zwalniać tablicę wielowymiarową:

```
for (i = 0; i<rozmiar; ++i) {
  free(tabliczka[i]);
}
free(tabliczka);
```

Należy nie pomylić kolejności: po wykonaniu `free(tabliczka)` nie będziemy mieli prawa odwoływać się do `tabliczka[i]` (bo wcześniej dokonaliśmy zwolnienia tego obszaru pamięci).

Można także zastosować bardziej oszczędny sposób alokowania tablicy wielowymiarowej, a mianowicie:

```
#define ROZMIAR 10
int i;
int **tabliczka = malloc(ROZMIAR * sizeof *tabliczka);
*tabliczka = malloc(ROZMIAR * ROZMIAR * sizeof **tabliczka);
for (i = 1; i<ROZMIAR; ++i) {
   tabliczka[i] = tabliczka[0] + (i * ROZMIAR);
}

for (i = 0; i<ROZMIAR; ++i) {
  int j;
  for (j = 0; j<ROZMIAR; ++j) {
    tabliczka[i][j] = (i+1)*(j+1);
  }
}

free(*tabliczka);
free(tabliczka);
```

Powyższy kod działa w ten sposób, że zamiast dla poszczególnych wierszy alokować osobno pamięć alokuje pamięć dla wszystkich elementów tablicy i dopiero później przypisuje wskazania poszczególnych wskaźników-wierszy na kolejne bloki po ROZMIAR elementów.

Sposób ten jest bardziej oszczędny z dwóch powodów: Po pierwsze wykonywanych jest mniej operacji przydzielania pamięci (bo tylko dwie). Po drugie za każdym razem, gdy alokuje się pamięć trochę miejsca się marnuje, gdyż funkcja malloc musi w stogu przechowywać różne dodatkowe informacje na temat każdej zaalokowanej przestrzeni. Ponadto, czasami alokacja odbywa się blokami i gdy zażąda się niepełny blok to reszta bloku jest tracona.

Zauważmy, że w ten sposób możemy uzyskać nie tylko normalną, "kwadratową" tablicę (dla dwóch wymiarów). Możliwe jest np. uzyskanie tablicy trójkątnej:

```
0123
012
01
0
```

lub tablicy o dowolnym innym rozkładzie długości wierszy, np.:

```
const size_t wymiary[] = { 2, 4, 6, 8, 1, 3, 5, 7, 9 };
int i;
int **tablica = malloc((sizeof wymiary / sizeof *wymiary) * sizeof *tablica);
for (i = 0; i<10; ++i) {
  tablica[i] = malloc(wymiary[i] * sizeof **tablica);
}
```

Gdy nabierzesz wprawy w używaniu wskaźników oraz innych funkcji malloc i realloc nauczysz się wykonywać różne inne operacje takie jak dodawanie kolejnych wierszy, usuwanie wierszy, zmiana rozmiaru wierszy, zamiana wierszy miejscami itp.

Wskaźniki na funkcje

Dotychczas zajmowaliśmy się sytuacją, gdy wskaźnik wskazywał na jakąś zmienną. Jednak nie tylko zmienna ma swój adres w pamięci. Oprócz zmiennej także i funkcja musi mieć swoje

określone miejsce w pamięci. A ponieważ funkcja ma swój adres[6], to nie ma przeszkód, aby i na nią wskazywał jakiś wskaźnik.

Deklaracja wskaźnika na funkcję

Tak naprawdę kod maszynowy utworzony po skompilowaniu programu odnosi się właśnie do adresu funkcji. Wskaźnik na funkcję różni się od innych rodzajów wskaźników. Jedną z głównych różnic jest jego deklaracja. Zwykle wygląda ona tak:

```
typ_zwracanej_wartości (*nazwa_wskaźnika)(typ1 parametr1, typ2 parametr2);
```

Oczywiście parametrów może być więcej (albo też w ogóle może ich nie być). Oto przykład wykorzystania wskaźnika na funkcję:

```
#include <stdio.h>

int suma (int a, int b)
{
  return a+b;
}

int main ()
{
  int (*wsk_suma)(int a, int b);
  wsk_suma = suma;
  printf("4+5=%d\n", wsk_suma(4,5));
  return 0;
}
```

Zwróćmy uwagę na dwie rzeczy:

1. przypisując nazwę funkcji bez nawiasów do wskaźnika automatycznie informujemy kompilator, że chodzi nam o **adres** funkcji
2. wskaźnika używamy tak, jak normalnej funkcji, na którą on wskazuje

Do czego można użyć wskaźników na funkcje?

Język C jest językiem strukturalnym, jednak dzięki wskaźnikom istnieje w nim możliwość "zaszczepienia" pewnych obiektowych właściwości. Wskaźnik na funkcję może być np. elementem struktury — wtedy mamy bardzo prymitywną namiastkę klasy, którą dobrze znają programiści, piszący w języku C++. Ponadto dzięki wskaźnikom możemy tworzyć mechanizmy działające na zasadzie funkcji zwrotnej[7]. Dobrym przykładem może być np. tworzenie sterowników, gdzie musimy poinformować różne podsystemy, jakie funkcje w naszym kodzie służą do wykonywania określonych czynności. Przykład:

```
struct urzadzenie {
  int (*otworz)(void);
  void (*zamknij)(void);
};

int moje_urzadzenie_otworz (void)
{
```

[6] Tak naprawdę kod maszynowy utworzony po skompilowaniu programu odnosi się właśnie do adresu funkcji.

[7] Funkcje zwrotne znalazły zastosowanie głównie w programowaniu GUI

```
  /* kod...*/
}

void moje_urzadzenie_zamknij (void)
{
  /* kod... */
}

int rejestruj_urzadzenie(struct urzadzenie &u) {
  /* kod... */
}

int init (void)
{
  struct urzadzenie moje_urzadzenie;
  moje_urzadzenie.otworz = moje_urzadzenie_otworz;
  moje_urzadzenie.zamknij = moje_urzadzenie_zamknij;
  rejestruj_urzadzenie(&moje_urzadzenie);
}
```

W ten sposób w pamięci każda *klasa* musi przechowywać wszystkie wskaźniki do wszystkich *metod*. Innym rozwiązaniem może być stworzenie statycznej struktury ze wskaźnikami do funkcji i wówczas w strukturze będzie przechowywany jedynie wskaźnik do tej struktury, np.:

```
struct urzadzenie_metody {
  int (*otworz)(void);
  void (*zamknij)(void);
};

struct urzadzenie {
  const struct urzadzenie_metody *m;
}

int moje_urzadzenie_otworz (void)
{
  /* kod...*/
}

void moje_urzadzenie_zamknij (void)
{
  /* kod... */
}

static const struct urzadzenie_metody
  moje_urzadzenie_metody = {
  moje_urzadzenie_otworz,
  moje_urzadzenie_zamknij
};

int rejestruj_urzadzenie(struct urzadzenie &u) {
  /* kod... */
}

int init (void)
```

```
{
  struct urzadzenie moje_urzadzenie;
  moje_urzadzenie.m = &moje_urzadzenie_metody;
  rejestruj_urzadzenie(&moje_urzadzenie);
}
```

Możliwe deklaracje wskaźników

Tutaj znajduje się krótkie kompendium jak definiować wskaźniki oraz co oznaczają poszczególne definicje:

`i;`	zmienna całkowita (typu int) `i`
`*p;`	wskaźnik `p` wskazujący na zmienną całkowitą
`a[];`	tablica `a` liczb całkowitych typu int
`f();`	funkcja `f` zwracająca liczbę całkowitą typu int
`**pp;`	wskaźnik `pp` na wskaźnik wskazujący na liczbę całkowitą typu int
`(*pa)[];`	wskaźnik `pa` wskazujący na tablicę liczb całkowitych typu int
`(*pf)();`	wskaźnik `pf` na funkcję zwracającą liczbę całkowitą typu int
`*ap[];`	tablica `ap` wskaźników na liczby całkowite typu int
`*fp();`	funkcja `fp`, która zwraca wskaźnik na zmienną typu int
`***ppp;`	wskaźnik `ppp` wskazujący na wskaźnik wskazujący na wskaźnik wskazujący na liczbę typu int
`(**ppa)[];`	wskaźnik `ppa` na wskaźnik wskazujący na tablicę liczb całkowitych typu int
`(**ppf)();`	wskaźnik `ppf` wskazujący na wskaźnik funkcji zwracającej dane typu int
`*(*pap)[];`	wskaźnik `pap` wskazujący na tablicę wskaźników na typ int
`*(*pfp)();`	wskaźnik `pfp` na funkcję zwracającą wskaźnik na typ int
`**app[];`	tablica wskaźników `app` wskazujących na typ int
`(*apa[])[];`	tablica wskaźników `apa` wskazujących wskaźniki na typ int
`(*apf[])();`	tablica wskaźników `apf` na funkcję, które zwracają wskaźniki na typ int
`***fpp();`	funkcja `fpp`, która zwraca wskaźnik na wskaźnik na wskaźnik, który wskazuje typ int
`(*fpa())[];`	funkcja `fpa`, która zwraca wskaźnik na tablicę liczb typu int
`(*fpf())();`	funkcja `fpf`, która zwraca wskaźnik na funkcję, która zwraca dane typu int

Popularne błędy

Jednym z najczęstszych błędów, oprócz prób wykonania operacji na wskaźniku NULL, są odwołania się do obszaru pamięci po jego zwolnieniu. Po wykonaniu funkcji `free()` nie możemy już wykonywać żadnych odwołań do zwolnionego obszaru. Innym rodzajem błędów są:

1. odwołania do adresów pamięci, które są poza obszarem przydzielonym funkcją `malloc()`
2. brak sprawdzania, czy dany wskaźnik nie ma wartości NULL
3. wycieki pamięci, czyli nie zwalnianie całej, przydzielonej wcześniej pamięci

Ciekawostki

- w rozdziale Zmienne pisaliśmy o stałych. Normalnie nie mamy możliwości zmiany ich wartości, ale z użyciem wskaźników staje się to możliwe:

```
const int CONST=0;
int *c=&CONST;
*c = 1;
printf("%i\n",CONST); /* wypisuje 1 */
```

Konstrukcja taka może jednak wywołać ostrzeżenie kompilatora bądź nawet jego błąd — wtedy może pomóc jawne rzutowanie z `const int*` na `int*`.

- język C++ oferuje mechanizm podobny do wskaźników, ale nieco wygodniejszy – referencje
- język C++ dostarcza też innego sposobu dynamicznej alokacji i zwalniania pamięci — przez operatory new i delete
- w rozdziale Typy złożone znajduje się opis implementacji listy za pomocą wskaźników. Przykład ten może być bardzo przydatny przy zrozumieniu po co istnieją wskaźniki, jak się nimi posługiwać oraz jak dobrze zarządzać pamięcią.

Rozdział 18

Napisy

W dzisiejszych czasach komputer przestał być narzędziem tylko i wyłącznie do przetwarzania danych. Od programów komputerowych zaczęto wymagać czegoś nowego — program w wyniku swojego działania nie ma zwracać danych, rozumianych tylko przez autora programu, lecz powinien być na tyle komunikatywny, aby przeciętny użytkownik komputera mógł bez problemu tenże komputer obsłużyć. Do przechowywania tychże komunikatów służą tzw. "łańcuchy" (ang. string) czyli ciągi znaków.

Język C nie jest wygodnym narzędziem do manipulacji napisami. Jak się wkrótce przekonamy, zestaw funkcji umożliwiających operacje na napisach w bibliotece standardowej C jest raczej skromny. Dodatkowo, problemem jest sposób, w jaki łańcuchy przechowywane są w pamięci.

Napisy w języku C mogą być przyczyną wielu trudnych do wykrycia błędów w programach. Warto dobrze zrozumieć, jak należy operować na łańcuchach znaków i zachować szczególną ostrożność w tych miejscach, gdzie napisów używamy.

Łańcuchy znaków w języku C

Napis jest zapisywany w kodzie programu jako ciąg znaków zawarty pomiędzy dwoma cudzysłowami.

```
printf ("Napis w języku C");
```

W pamięci taki łańcuch jest następującym po sobie ciągiem znaków (char), który kończy się znakiem "null" (czyli po prostu liczbą zero), zapisywanym jako '\0'.

Jeśli mamy napis, do poszczególnych znaków odwołujemy się jak w tablicy:

```
char *tekst = "Jakiś tam tekst";
printf("%c\n", "przykład"[0]); /* wypisze p - znaki w napisach są numerowane od zera */
printf("%c\n", tekst[2]);      /* wypisze k */
```

Ponieważ napis w pamięci kończy się zerem umieszczonym tuż za jego zawartością, odwołanie się do znaku o indeksie równym długości napisu zwróci zero:

```
printf("%d", "test"[4]);       /* wypisze 0 */
```

Napisy możemy wczytywać z klawiatury i wypisywać na ekran przy pomocy dobrze znanych funkcji scanf, printf i pokrewnych. Formatem używanym dla napisów jest %s.

```
printf("%s", tekst);
```

Większość funkcji działających na napisach znajduje się w pliku nagłówkowym string.h.

Jeśli łańcuch jest zbyt długi, można zapisać go w kilku linijkach, ale wtedy przechodząc do następnej linii musimy na końcu postawić znak "\".

```
printf("Ten napis zajmuje \
więcej niż jedną linię");
```

Instrukcja taka wydrukuje:

```
Ten napis zajmuje więcej niż jedną linię
```

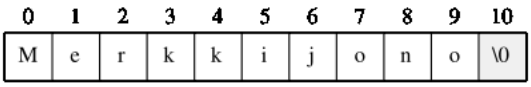
Możemy zauważyć, że napis, który w programie zajął więcej niż jedną linię, na ekranie zajął tylko jedną. Jest tak, ponieważ "\" informuje kompilator, że łańcuch będzie kontynuowany w następnej linii kodu — nie ma wpływu na prezentację łańcucha. Aby wydrukować napis w kilku liniach należy wstawić do niego `\n` ("n" pochodzi tu od "new line", czyli "nowa linia").

```
printf("Ten napis\nna ekranie\nzajmie więcej niż jedną linię.");
```

W wyniku otrzymamy:

```
Ten napis
na ekranie
zajmie więcej niż jedną linię.
```

Jak komputer przechowuje w pamięci łańcuch?



Rysunek 18.1: Napis "Merkkijono" przechowywany w pamięci

Zmienna, która przechowuje łańcuch znaków, jest tak naprawdę wskaźnikiem do ciągu znaków (bajtów) w pamięci. Możemy też myśleć o napisie jako o tablicy znaków (jak wyjaśnialiśmy wcześniej, tablice to też wskaźniki).

□ Możemy wygodnie zadeklarować napis:

```
char *tekst  = "Jakiś tam tekst"; /* Umieszcza napis w obszarze danych programu */
                                  /* i przypisuje adres */
char tekst[] = "Jakiś tam tekst"; /* Umieszcza napis w tablicy */
char tekst[] = {'J','a','k','i','s',' ','t','a','m',' ','t','e','k','s','t','\0'};
               /* Tekst to taka tablica jak każda inna */
```

Kompilator automatycznie przydziela wtedy odpowiednią ilość pamięci (tyle bajtów, ile jest liter plus jeden dla kończącego nulla). Jeśli natomiast wiemy, że dany łańcuch powinien przechowywać określoną ilość znaków (nawet, jeśli w deklaracji tego łańcucha podajemy mniej znaków) deklarujemy go w taki sam sposób, jak tablicę jednowymiarową:

```
char tekst[80] = "Ten tekst musi być krótszy niż 80 znaków";
```

Należy cały czas pamiętać, że napis jest tak naprawdę tablicą. Jeśli zarezerwowaliśmy dla napisu 80 znaków, to przypisanie do niego dłuższego napisu spowoduje **pisanie po pamięci**.

Uwaga! Deklaracja `char *tekst = cokolwiek;` oraz `char tekst = cokolwiek;` pomimo, że wyglądają bardzo podobnie bardzo się od siebie różnią. W przypadku pierwszej deklaracji próba zmodyfikowania napisu (np. `tekst[0] = 'C';`) może mieć nieprzyjemne skutki. Dzieje się tak dlatego, że `char *tekst = cokolwiek;` deklaruje wskaźnik na **stały** obszar pamięci [1].

Pisanie po pamięci może czasami skończyć się błędem dostępu do pamięci ("segmentation fault" w systemach UNIX) i zamknięciem programu, jednak może zdarzyć się jeszcze gorsza ewentualność — możemy zmienić w ten sposób przypadkowo wartość innych zmiennych. Program zacznie wtedy zachowywać się nieprzewidywalnie — zmienne a nawet stałe, co do których zakładaliśmy, że ich wartość będzie ściśle ustalona, mogą przyjąć taką wartość, jaka absolutnie nie powinna mieć miejsca. Warto więc stosować zabezpieczenia typu makra assert.

Kluczowy jest też kończący napis znak null. W zasadzie wszystkie funkcje operujące na napisach opierają właśnie na nim. Na przykład, strlen szuka rozmiaru napisu idąc od początku i zliczając znaki, aż nie natrafi na znak o kodzie zero. Jeśli nasz napis nie kończy się znakiem null, funkcja będzie szła dalej po pamięci. Na szczęście, wszystkie operacje podstawienia typu tekst = "Tekst" powodują zakończenie napisu nullem (o ile jest na niego miejsce) [2].

Znaki specjalne

Jak zapewne zauważyłeś w poprzednim przykładzie, w łańcuchu ostatnim znakiem jest znak o wartości zero ('\0'). Jednak łańcuchy mogą zawierać inne znaki specjalne(sekwencje sterujące), np.:

- '\a' - alarm (sygnał akustyczny terminala)
- '\b' - backspace (usuwa poprzedzający znak)
- '\f' - wysuniecie strony (np. w drukarce)
- '\r' - powrót kursora (karetki) do początku wiersza
- '\n' - znak nowego wiersza
- '\"' - cudzysłów
- '\'' - apostrof
- '\\' - ukośnik wsteczny (backslash)
- '\t' - tabulacja pozioma
- '\v' - tabulacja pionowa
- '\?' - znak zapytania (pytajnik)
- '\ooo' - liczba zapisana w systemie oktalnym (ósemkowym), gdzie 'ooo' należy zastąpić trzycyfrową liczbą w tym systemie
- '\xhh' - liczba zapisana w systemie heksadecymalnym (szesnastkowym), gdzie 'hh' należy zastąpić dwucyfrową liczbą w tym systemie
- '\unnnn' - uniwersalna nazwa znaku, gdzie 'nnnn' należy zastąpić czterocyfrowym identyfikatorem znaku w systemie szesnatkowym. 'nnnn' odpowiada dłuższej formie w postaci '0000nnnn'
- '\unnnnnnnn' - uniwersalna nazwa znaku, gdzie 'nnnnnnnn' należy zastąpić ośmiocyfrowym identyfikatorem znaku w systemie szesnatkowym.

[1] Można się zatem zastanawiać czemu kompilator dopuszcza przypisanie do zwykłego wskaźnika wskazania na stały obszar, skoro kod `const int *foo; int *bar = foo;` generuje ostrzeżenie lub wręcz się nie kompiluje. Jest to pewna zaszłość historyczna wynikająca, z faktu, że słówko const zostało wprowadzone do języka, gdy już był on w powszechnym użyciu.

[2] Nie należy mylić znaku null (czyli znaku o kodzie zero) ze wskaźnikiem null (czy też NULL).

Warto zaznaczyć, że znak nowej linii ('\n') jest w różny sposób przechowywany w różnych systemach operacyjnych. Wiąże się to z pewnymi historycznymi uwarunkowaniami. W niektórych systemach używa się do tego jednego znaku o kodzie 0x0A (Line Feed — nowa linia). Do tej rodziny zaliczamy systemy z rodziny Unix: Linux, *BSD, Mac OS X inne. Drugą konwencją jest zapisywanie '\n' za pomocą dwóch znaków: LF (Line Feed) + CR (Carriage return — powrót karetki). Znak CR reprezentowany jest przez wartość 0x0D. Kombinacji tych dwóch znaków używają m.in.: CP/M, DOS, OS/2, Microsoft Windows. Trzecia grupa systemów używa do tego celu samego znaku CR. Są to systemy działające na komputerach Commodore, Apple II oraz Mac OS do wersji 9. W związku z tym plik utworzony w systemie Linux może wyglądać dziwnie pod systemem Windows.

Operacje na łańcuchach

Porównywanie łańcuchów

Napisy to tak naprawdę wskaźniki. Tak więc używając zwykłego operatora porównania ==, otrzymamy wynik porównania adresów a nie tekstów.

Do porównywania dwóch ciągów znaków należy użyć funkcji strcmp zadeklarowanej w pliku nagłówkowym string.h. Jako argument przyjmuje ona dwa napisy i zwraca wartość ujemną jeżeli napis pierwszy jest *mniejszy* od drugiego, 0 jeżeli napisy są równe lub wartość dodatnią jeżeli napis pierwszy jest *większy* od drugiego. Ciągi znaków porównywalne są leksykalnie kody znaków, czyli np. (przyjmując kodowanie ASCII) `a` jest mniejsze od `b`, ale jest większe od `B`. Np.:

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

int main(void) {
  char str1[100], str2[100];
  int cmp;

  puts("Podaj dwa ciagi znakow: ");
  fgets(str1, sizeof str1, stdin);
  fgets(str2, sizeof str2, stdin);

  cmp = strcmp(str1, str2);
  if (cmp<0) {
    puts("Pierwszy napis jest mniejszy.");
  } else if (cmp>0) {
    puts("Pierwszy napis jest wiekszy.");
  } else {
    puts("Napisy sa takie same.");
  }

  return 0;
}
```

Czasami możemy chcieć porównać tylko fragment napisu, np. sprawdzić czy zaczyna się od jakiegoś ciągu. W takich sytuacjach pomocna jest funkcja strncmp. W porównaniu do strcmp() przyjmuje ona jeszcze jeden argument oznaczający maksymalną liczbę znaków do porównania:

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

```
int main(void) {
  char str[100];
  int cmp;

  fputs("Podaj ciag znakow: ", stdout);
  fgets(str, sizeof str, stdin);

  if (!strncmp(str, "foo", 3)) {
    puts("Podany ciag zaczyna sie od 'foo'.");
  }

  return 0;
}
```

Kopiowanie napisów

Do kopiowania ciągów znaków służy funkcja strcpy, która kopiuje drugi napis w miejsce pierwszego. Musimy pamiętać, by w pierwszym łańcuchu było wystarczająco dużo miejsca.

```
char napis[100];
strcpy(napis, "Ala ma kota.");
```

Znacznie bezpieczniej jest używać funkcji strncpy, która kopiuje co najwyżej tyle bajtów ile podano jako trzeci parametr. Uwaga! Jeżeli drugi napis jest za długi funkcja **nie** kopiuje znaku null na koniec pierwszego napisu, dlatego zawsze trzeba to robić ręcznie:

```
char napis[100];
strncpy(napis, "Ala ma kota.", sizeof napis - 1);
napis[sizeof napis - 1] = 0;
```

Łączenie napisów

Do łączenia napisów służy funkcja strcat, która kopiuje drugi napis do pierwszego. Ponownie jak w przypadku strcpy musimy zagwarantować, by w pierwszym łańcuchu było wystarczająco dużo miejsca.

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

int main(void) {
  char napis1[80] = "hello ";
  char *napis2 = "world";
  strcat(napis1, napis2);
  puts(napis1);
  return 0;
}
```

I ponownie jak w przypadku strcpy istnieje funkcja strncat, która skopiuje co najwyżej tyle bajtów ile podano jako trzeci argument i *dodatkowo* dopisze znak null. Przykładowo powyższy kod bezpieczniej zapisać jako:

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

```
int main(void) {
  char napis1[80] = "hello ";
  char *napis2 = "world";
  strncat(napis1, napis2, sizeof napis1 - 1);
  puts(napis1);
  return 0;
}
```

Osoby, które programowały w językach skryptowych muszą bardzo uważać na łączenie i kopiowanie napisów. Kompilator języka C nie wykryje nadpisania pamięci za zmienną łańcuchową i nie przydzieli dodatkowego obszaru pamięci. Może się zdarzyć, że program pomimo nadpisywania pamięci za łańcuchem będzie nadal działał, co bardzo utrudni wykrywanie tego typu błędów!

Bezpieczeństwo kodu a łańcuchy

Przepełnienie bufora

O co właściwie chodzi z tymi funkcjami strncpy i strncat? Otóż, niewinnie wyglądające łańcuchy mogą okazać się zabójcze dla bezpieczeństwa programu, a przez to nawet dla systemu, w którym ten program działa. Może brzmi to strasznie, lecz jest to prawda. Może pojawić się tutaj pytanie: "w jaki sposób łańcuch może zaszkodzić programowi?". Otóż może i to całkiem łatwo. Przeanalizujmy następujący kod:

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

int main(int argc, char **argv) {
  char haslo_poprawne = 0;
  char haslo[16];

  if (argc!=2) {
    fprintf(stderr, "uzycie: %s haslo", argv[0]);
    return EXIT_FAILURE;
  }

  strcpy(haslo, argv[1]); /* tutaj następuje przepełnienie bufora */
  if (!strcmp(haslo, "poprawne")) {
    haslo_poprawne = 1;
  }

  if (!haslo_poprawne) {
    fputs("Podales bledne haslo.\n", stderr);
    return EXIT_FAILURE;
  }

  puts("Witaj, wprowadziles poprawne haslo.");
  return EXIT_SUCCESS;
}
```

Jest to bardzo prosty program, który wykonuje jakąś akcję, jeżeli podane jako pierwszy argument hasło jest poprawne. Sprawdźmy czy działa:

```
$ ./a.out niepoprawne
Podales bledne haslo.
$ ./a.out poprawne
Witaj, wprowadziles poprawne haslo.
```

Jednak okazuje się, że z powodu użycia funkcji strcpy włamywacz nie musi znać hasła, aby program uznał, że zna hasło, np.:

```
$ ./a.out 1111111111111111111111111111111
Witaj, wprowadziles poprawne haslo.
```

Co się stało? Podaliśmy ciąg jedynek dłuższy niż miejsce przewidziane na hasło. Funkcja `strcpy()` kopiując znaki z `argv1` do tablicy (bufora) `haslo` przekroczyła przewidziane dla niego miejsce i szła dalej — gdzie znajdowała się zmienna `haslo_poprawne`. `strcpy()` kopiowała znaki już tam, gdzie znajdowały się inne dane — między innymi wpisała jedynkę do `haslo_poprawne`.

Podany przykład może się różnie zachowywać w zależności od kompilatora, jakim został skompilowany, i systemu, na jakim działa, ale ogólnie mamy do czynienia z poważnym niebezpieczeństwem.

Taką sytuację nazywamy przepełnieniem bufora. Może umożliwić dostęp do komputera osobom nieuprzywilejowanym. Należy wystrzegać się tego typu konstrukcji, a w miejsce niebezpiecznej funkcji strcpy stosować bardziej bezpieczną strncpy.

Oto bezpieczna wersja poprzedniego programu:

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

int main(int argc, char **argv) {
  char haslo_poprawne = 0;
  char haslo[16];

  if (argc!=2) {
    fprintf(stderr, "uzycie: %s haslo", argv[0]);
    return EXIT_FAILURE;
  }

  strncpy(haslo, argv[1], sizeof haslo - 1);
  haslo[sizeof haslo - 1] = 0;
  if (!strcmp(haslo, "poprawne")) {
    haslo_poprawne = 1;
  }

  if (!haslo_poprawne) {
    fputs("Podales bledne haslo.\n", stderr);
    return EXIT_FAILURE;
  }

  puts("Witaj, wprowadziles poprawne haslo.");
  return EXIT_SUCCESS;
}
```

Bezpiecznymi alternatywami do strcpy i strcat są też funkcje strlcpy oraz strlcat opracowana przez projekt OpenBSD i dostępna do ściągnięcia: strlcpy, strlcat. strlcpy() działa podobnie do strncpy: `strlcpy (buf, argv[1], sizeof buf);`, jednak jest szybsza (nie wypełnia pustego miejsca zerami) i zawsze kończy napis nullem (czego nie gwarantuje strncpy). `strlcat(dst, src, size)` działa natomiast jak `strncat(dst, src, size-1)`.

Do innych niebezpiecznych funkcji należy np. gets zamiast której należy używać fgets.

Zawsze możemy też alokować napisy dynamicznie:

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

int main(int argc, char **argv) {
  char haslo_poprawne = 0;
  char *haslo;

  if (argc!=2) {
    fprintf(stderr, "uzycie: %s haslo", argv[0]);
    return EXIT_FAILURE;
  }

  haslo = malloc(strlen(argv[1]) + 1); /* +1 dla znaku null */
  if (!haslo) {
    fputs("Za malo pamieci.\n", stderr);
    return EXIT_FAILURE;
  }

  strcpy(haslo, argv[1]);
  if (!strcmp(haslo, "poprawne")) {
    haslo_poprawne = 1;
  }

  if (!haslo_poprawne) {
    fputs("Podales bledne haslo.\n", stderr);
    return EXIT_FAILURE;
  }

  puts("Witaj, wprowadziles poprawne haslo.");
  free(haslo)
  return EXIT_SUCCESS;
}
```

Nadużycia z udziałem ciągów formatujących

Jednak to nie koniec kłopotów z napisami. Wielu programistów, nieświadomych zagrożenia często używa tego typu konstrukcji:

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  printf (argv[1]);
}
```

Z punktu widzenia bezpieczeństwa jest to bardzo poważny błąd programu, który może nieść ze sobą katastrofalne skutki! Prawidłowo napisany kod powinien wyglądać następująco:

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  printf ("%s", argv[1]);
}
```

lub:

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  fputs (argv[1], stdout);
}
```

Źródło problemu leży w konstrukcji funkcji printf. Przyjmuje ona bowiem za pierwszy parametr łańcuch, który następnie przetwarza. Jeśli w pierwszym parametrze wstawimy jakąś zmienną, to funkcja printf potraktuje ją jako ciąg znaków razem ze znakami formatującymi. Zatem ważne, aby wcześnie wyrobić sobie nawyk stosowania funkcji printf z co najmniej dwoma parametrami, nawet w przypadku wyświetlenia samego tekstu.

Konwersje

Czasami zdarza się, że łańcuch można interpretować nie tylko jako ciąg znaków, lecz np. jako liczbę. Jednak, aby dało się taką liczbę przetworzyć musimy skopiować ją do pewnej zmiennej. Aby ułatwić programistom tego typu zamiany powstał zestaw funkcji bibliotecznych. Należą do nich:

- atol, strtol — zamienia łańcuch na liczbę całkowitą typu long
- atoi — zamienia łańcuch na liczbę całkowitą typu int
- atoll, strtoll — zamienia łańcuch na liczbę całkowitą typu long long (64 bity); dodatkowo istnieje przestarzała funkcja atoq będąca rozszerzeniem GNU,
- atof, strtod — przekształca łańcuch na liczbę typu double

Ogólnie rzecz ujmując funkcje z serii ato* nie pozwalają na wykrycie błędów przy konwersji i dlatego, gdy jest to potrzebne, należy stosować funkcje strto*.

Czasami przydaje się też konwersja w drugą stronę, tzn. z liczby na łańcuch. Do tego celu może posłużyć funkcja sprintf lub snprintf. sprintf jest bardzo podobna do printf, tyle, że wyniki jej prac zwracane są do pewnego łańcucha, a nie wyświetlane np. na ekranie monitora. Należy jednak uważać przy jej użyciu (patrz — Bezpieczeństwo kodu a łańcuchy). snprintf (zdefiniowana w nowszym standardzie) dodatkowo przyjmuje jako argument wielkość bufora docelowego.

Operacje na znakach

Warto też powiedzieć w tym miejscu o operacjach na samych znakach. Spójrzmy na poniższy program:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int main()
{
  int znak;
```

```
  while ((znak = getchar())!=EOF) {
    if( islower(znak) ) {
      znak = toupper(znak);
    } else if( isupper](znak) ) {
      znak = tolower(znak);
    }
    putchar(znak);
  }
  return 0;
}
```

Program ten zmienia we wczytywanym tekście wielkie litery na małe i odwrotnie. Wykorzystujemy funkcje operujące na znakach z pliku nagłówkowego ctype.h. isupper sprawdza, czy znak jest wielką literą, natomiast toupper zmienia znak (o ile jest literą) na wielką literę. Analogicznie jest dla funkcji islower i tolower.

Jako ćwiczenie, możesz tak zmodyfikować program, żeby odczytywał dane z pliku podanego jako argument lub wprowadzonego z klawiatury.

Częste błędy

- pisanie do niezaalokowanego miejsca

```
char *tekst;
scanf("%s", tekst);
```

- zapominanie o kończącym napis nullu

```
char test[4] = "test"; /* nie zmieścił się null kończący napis */
```

- nieprawidłowe porównywanie łańcuchów

```
char tekst1[] = "jakis tekst";
char tekst2[] = "jakis tekst";
if( tekst1 == tekst2 ) { /* tu zawsze będzie fałsz bo == porównuje adresy, należy użyć strcmp() */
   ...
}
```

Unicode

Zobacz w Wikipedii: Unicode

W dzisiejszych czasach brak obsługi wielu języków praktycznie marginalizowałoby język. Dlatego też C99 wprowadza możliwość zapisu znaków wg norm Unicode.

Jaki typ?

Do przechowywania znaków zakodowanych w Unicode powinno się korzystać z typu **wchar_t**. Jego domyślny rozmiar jest zależny od użytego kompilatora, lecz w większości zaktualizowanych kompilatorów powinny to być 2 bajty. Typ ten jest częścią języka C++, natomiast w C znajduje się w pliku nagłówkowym stddef.h.

Alternatywą jest wykorzystanie gotowych bibliotek dla Unicode (większość jest dostępnych jedynie dla C++, nie współpracuje z C), które często mają zdefiniowane własne typy, jednak zmuszeni jesteśmy wtedy do przejścia ze znanych nam już funkcji jak np. strcpy, strcmp na funkcje dostarczane przez bibliotekę, co jest dość niewygodne. My zajmiemy się pierwszym wyjściem.

Jaki rozmiar i jakie kodowanie?

Unicode określa jedynie jakiej liczbie odpowiada jaki znak, nie mówi zaś nic o sposobie dekodowania (tzn. jaka sekwencja znaków odpowiada jakiemu znaku/znakom). Jako że Unicode obejmuje 918 tys. znaków, zmienna zdolna pomieścić go w całości musi mieć przynajmniej 3 bajty. Niestety procesory nie funkcjonują na zmiennych o tym rozmiarze, pracują jedynie na zmiennych o wielkościach: 1, 2, 4 oraz 8 bajtów (kolejne potęgi liczby 2). Dlatego też jeśli wciąż uparcie chcemy być dokładni i zastosować przejrzyste kodowanie musimy skorzystać ze zmiennej 4-bajtowej (32 bity). Tak do sprawy podeszli twórcy kodowania Unicode nazwanego UTF-32/UCS-4. Ten typ kodowania po prostu przydziela każdemu znakowi Unicode kolejne liczby. Jest to najbardziej intuicyjny i wygodny typ kodowania, ale jak widać ciągi znaków zakodowane w nim są bardzo obszerne, co zajmuje dostępną pamięć, spowalnia działanie programu oraz drastycznie pogarsza wydajność podczas transferu przez sieć. Poza UTF-32 istnieje jeszcze wiele innych kodowań. Najpopularniejsze z nich to:

Zobacz w Wikipedii: UTF-32

- UTF-8 — od 1 do 6 bajtów (dla znaków poniżej 65536 do 3 bajtów) na znak przez co jest skrajnie niewygodny, gdy chcemy przeprowadzać jakiekolwiek operacje na tekście bez korzystania z gotowych funkcji
- UTF-16 — 2 lub 4 bajty na znak; ręczne modyfikacje łańcucha są bardziej skomplikowane niż przy UTF-32
- UCS-2 — 2 bajty na znak przez co znaki z numerami powyżej 65 535 nie są uwzględnione; równie wygodny w użytkowaniu co UTF-32.

Ręczne operacje na ciągach zakodowanych w UTF-8 i UTF-16 są utrudnione, ponieważ w przeciwieństwie do UTF-32, gdzie można określić, iż powiedzmy 2. znak ciągu zajmuje bajty od 4. do 7. (gdyż z góry wiemy, że 1. znak zajął bajty od 0. do 3.), w tych kodowaniach musimy najpierw określić rozmiar 1. znaku. Ponadto, gdy korzystamy z nich nie działają wtedy funkcje udostępniane przez biblioteki C do operowania na ciągach znaków.

Priorytet	**Proponowane kodowania**
mały rozmiar	UTF-8
łatwa i wydajna edycja	UTF-32 lub UCS-2
przenośność	UTF-8[3]
ogólna szybkość	UCS-2 lub UTF-8

Co należy zrobić, by zacząć korzystać z kodowania UCS-2 (domyślne kodowanie dla C):

- powinniśmy korzystać z typu wchar_t (ang. “wide character”), jednak jeśli chcemy udostępniać kod źródłowy programu do kompilacji na innych platformach, powinniśmy ustawić odpowiednie parametry dla kompilatorów, by rozmiar był identyczny niezależnie od platformy.
- korzystamy z odpowiedników funkcji operujących na typie char pracujących na wchar_t (z reguły składnia jest identyczna z tą różnicą, że w nazwach funkcji zastępujemy “str” na “wcs” np. strcpy — wcscpy; strcmp — wcscmp)
- jeśli przyzwyczajeni jesteśmy do korzystania z klasy string, powinniśmy zamiast niej korzystać z wstring, która posiada zbliżoną składnię, ale pracuje na typie wchar_t.

Co należy zrobić, by zacząć korzystać z Unicode:

- gdy korzystamy z kodowań innych niż UTF-16 i UCS-2, powinniśmy zdefiniować własny typ
- w wykorzystywanych przez nas bibliotekach podajemy typ wykorzystanego kodowania.
- gdy chcemy ręcznie modyfikować ciąg **musimy** przeczytać specyfikację danego kodowania; są one wyczerpująco opisane na siostrzanym projekcie Wikibooks — Wikipedii.

Przykład użycia kodowania UCS-2:

```
#include <stddef.h> /* jeśli używamy C++, możemy opuścić tę linijkę */
#include <stdio.h>
#include <string.h>

int main() {
  wchar_t* wcs1 = L"Ala ma kota.";
  wchar_t* wcs2 = L"Kot ma Ale.";
  wchar_t calosc[25];

  wcscpy(calosc, wcs1);
  *(calosc + wcslen(wcs1)) = L' ';
  wcscpy(calosc + wcslen(wcs1) + 1, wcs2);

  printf("lancuch wyjsciowy: %ls\n", calosc);
  return 0;
}
```

Rozdział 19

Typy złożone

typedef

Jest to słowo kluczowe, które służy do definiowania typów pochodnych np.:

```
typedef stara_nazwa  nowa_nazwa;
typedef int mojInt;
typedef int* WskNaInt;
```

od tej pory mozna używać typów mojInt i WskNaInt.

Typ wyliczeniowy

Służy do tworzenia zmiennych, które powinny przechowywać tylko pewne z góry ustalone wartości:

```
enum Nazwa {WARTOSC_1, WARTOSC_2, WARTOSC_N };
```

Na przykład można w ten sposób stworzyć zmienną przechowującą kierunek:

```
enum Kierunek {W_GORE, W_DOL, W_LEWO, W_PRAWO};

enum Kierunek kierunek = W_GORE;
```

którą można na przykład wykorzystać w instrukcji switch

```
switch(kierunek)
{
  case W_GORE:
    printf("w górę\n");
    break;
  case W_DOL:
    printf("w dół\n");
    break;
  default:
    printf("gdzieś w bok\n");
}
```

Tradycyjnie przechowywane wielkości zapisuje się wielkimi literami (W_GORE, W_DOL).

Tak naprawdę C przechowuje wartości typu wyliczeniowego jako liczby całkowite, o czym można się łatwo przekonać:

```
kierunek = W_DOL;
printf("%i\n", kierunek); /* wypisze 1 */
```

Kolejne wartości to po prostu liczby naturalne: domyślnie pierwsza to zero, druga jeden itp. Możemy przy deklarowaniu typu wyliczeniowego zmienić domyślne przyporządkowanie:

```
enum Kierunek { W_GORE, W_DOL = 8, W_LEWO, W_PRAWO };
printf("%i %i\n", W_DOL, W_LEWO); /* wypisze 8 9 */
```

Co więcej liczby mogą się powtarzać i wcale nie muszą być ustawione w kolejności rosnącej:

```
enum Kierunek { W_GORE = 5, W_DOL = 5, W_LEWO = 2, W_PRAWO = 1 };
printf("%i %i\n", W_DOL, W_LEWO); /* wypisze 5 2 */
```

Traktowanie przez kompilator typu wyliczeniowego jako liczby pozwala na wydajną ich obsługę, ale stwarza niebezpieczeństwa — można przypisywać pod typ wyliczeniowy liczby, nawet nie mające odpowiednika w wartościach, a kompilator może o tym nawet nie ostrzec:

```
kierunek = 40;
```

Unie

Są to twory deklarowane w następujący sposób:

```
union Nazwa {
  typ1 nazwa1;
  typ2 nazwa2;
  /* ... */
};
```

Na przykład:

```
union LiczbaLubZnak {
  int calkowita;
  char znak;
  double rzeczywista;
};
```

Pola w unii **nakładają** się na siebie w ten sposób, że w danej chwili można w niej przechowywać wartość tylko jednego typu. Unia zajmuje w pamięci tyle miejsca, ile zajmuje największa z jej składowych. W powyższym przypadku unia będzie miała prawdopodobnie rozmiar typu double czyli często 64 bity, a całkowita i znak będą wskazywały odpowiednio na pierwsze cztery bajty lub na pierwszy bajt unii (choć nie musi tak być zawsze).

Do konkretnych wartości pól unii odwołujemy się przy pomocy **operatorem wyboru składnika** — kropki:

```
union LiczbaLubZnak liczba;
liczba.calkowita = 10;
printf("%d\n", liczba.calkowita);
```

Zazwyczaj użycie unii ma na celu zmniejszenie zapotrzebowania na pamięć, gdy naraz będzie wykorzystywane tylko jedno pole i jest często łączone z użyciem struktur.

Życiowy przykład użycia — zamieniamy kolejnosc bajtow w p.p1 — np. w kodzie oprogramowania sieciowego Big Endian -> Little Endian, gdypotrzeba zmiany numeru portu podanego w “nie sieciowej” kolejnosci bajtów:

```
char t;
union ZamianaKolejnosci {
  unsigned short p1;
  unsigned char p2[2];
};
union ZamianaKolejnosci p;
p.p1=GetPortNr(); /* w wyniku czego z np. klawiatury w p.p1 bedzie 0x3412 */
/* Teraz zamienmy kolejnosc bajtow */
t=p.p2[1];
p.p2[1]=p.p2[0];
p.p2[0]=t;
/*a teraz mozemy wyslac pakiet UDP np. w systemie Ethernut (port RTOS dla urzadzen wbudowanych) */
r=NutUdpSendTo(sock, ip, p.p1, "ALA MA KOTA", strlen("ALA MA KOTA"));
```

Na koniec w zmiennej p.p1 mamy 0x1234.

Struktury

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

Struktury definiuje się podobnie jak unie, ale zamiast słowa kluczowego **union** używa się **struct**, np.:

```
struct Struktura {
  int pole1;
  int pole2;
  char pole3;
};
```

Zmienną posiadającą strukturę tworzy się podając jako jej typ nazwę struktury.

```
struct Struktura zmiennaS;
```

Dostęp do poszczególnych pól, tak samo jak w przypadku unii, uzyskuje się przy pomocy operatora kropki:

```
zmiennaS.pole1 = 60;   /* przypisanie liczb do pól */
zmiennaS.pole2 = 2;
zmiennaS.pole3 = 'a';  /* a teraz znaku */
```

Wspólne własności typów wyliczeniowych, unii i struktur

Warto w zwrócić uwagę, że język C++ przy deklaracji zmiennych typów wyliczeniowych, unii lub struktur nie wymaga przed nazwą typu odpowiedniego słowa kluczowego. Na przykład poniższy kod jest poprawnym programem C++:

```
enum   Enum   { A, B, C };
union  Union  { int a; float b; };
struct Struct { int a; float b; };
int main() {
  Enum   e;
  Union  u;
  Struct s;
```

```
  e = A;
  u.a = 0;
  s.a = 0;
  return e + u.a + s.a;
}
```

Nie jest to jednak poprawny kod C i należy o tym pamiętać szczególnie jeżeli uczysz się języka C korzystając z kompilatora C++.

Należy również pamiętać, że po klamrze zamykającej definicje **musi** następować średnik. Brak tego średnika jest częstym błędem powodującym czasami niezrozumiałe komunikaty błędów. Jedynym wyjątkiem jest natychmiastowa definicja zmiennych danego typu, na przykład:

```
struct Struktura {
  int pole;
} s1, s2, s3;
```

Definicja typów wyliczeniowych, unii i struktur jest lokalna do bloku. To znaczy, możemy zdefiniować strukturę wewnątrz jednej z funkcji (czy wręcz wewnątrz jakiegoś bloku funkcji) i tylko tam będzie można używać tego typu.

Częstym idiomem w C jest użycie `typedef` od razu z definicją typu, by uniknąć pisania `enum`, `union` czy `struct` przy deklaracji zmiennych danego typu.

```
typedef struct struktura {
  int pole;
} Struktura;
Struktura s1;
struct struktura s2;
```

W tym przypadku zmienne s1 i s2 są tego samego typu. Możemy też zrezygnować z nazywania samej struktury:

```
typedef struct {
  int pole;
} Struktura;
Struktura s1;
```

Wskaźnik na unię i strukturę

Podobnie, jak na każdą inną zmienna, wskaźnik może wskazywać także na unię lub strukturę. Oto przykład:

```
typedef struct {
  int p1, p2;
} Struktura;

int main ()
{
  Struktura s = { 0, 0 };
  Struktura *wsk = &s;
  wsk->p1 = 2;
  wsk->p2 = 3;
  return 0;
}
```

Zapis `wsk->p1` jest (z definicji) równoważny `(*wsk).p1`, ale bardziej przejrzysty i powszechnie stosowany. Wyrażenie `wsk.p1` spowoduje błąd kompilacji (strukturą jest `*wsk` a nie `wsk`).

Zobacz też

- Powszechne praktyki — konstruktory i destruktory

Pola bitowe

Struktury mają pewne dodatkowe możliwości w stosunku do zmiennych. Mowa tutaj o rozmiarze elementu struktury. W przeciwieństwie do zmiennej może on mieć nawet 1 bit!. Aby móc zdefiniować taką zmienną musimy użyć tzw. **pola bitowego**. Wygląda ono tak:

```
struct moja {
  unsigned int a1:4, /* 4 bity */
               a2:8, /* 8 bitów (często 1 bajt) */
               a3:1, /* 1 bit */
               a4:3; /* 3 bity */
};
```

Wszystkie pola tej struktury mają w sumie rozmiar 16 bitów, jednak możemy odwoływać się do nich w taki sam sposób, jak do innych elementów struktury. W ten sposób efektywniej wykorzystujemy pamięć, jednak istnieją pewne zjawiska, których musimy być świadomi przy stosowaniu pól bitowych. Więcej na ten temat w rozdziale przenośność programów.

Pola bitowe znalazły zastosowanie głównie w implementacjach protokołów sieciowych.

Studium przypadku — implementacja listy wskaźnikowej

Zobacz w Wikipedii: Lista

Rozważmy teraz coś, co każdy z nas może spotkać w codziennym życiu. Każdy z nas widział kiedyś jakiś przykład listy (czy to zakupów, czy też listę wierzycieli). Język C też oferuje listy, jednak w programowaniu listy będą służyły do czegoś innego. Wyobraźmy sobie sytuację, w której jesteśmy autorami genialnego programu, który znajduje kolejne liczby pierwsze. Oczywiście każdą kolejną liczbę pierwszą może wyświetlać na ekran, jednak z matematyki wiemy, że dana liczba jest liczbą pierwszą, jeśli nie dzieli się przez żadną liczbę pierwszą ją poprzedzającą, mniejszą od pierwiastka z badanej liczby. Uff, mniej więcej chodzi o to, że moglibyśmy wykorzystać znalezione wcześniej liczby do przyspieszenia działania naszego programu. Jednak nasze liczby trzeba jakoś mądrze przechować w pamięci. Tablice mają ograniczenie — musimy z góry znać ich rozmiar. Jeśli zapełnilibyśmy tablicę, to przy znalezieniu każdej kolejnej liczby musielibyśmy:

1. przydzielać nowy obszar pamięci o rozmiarze poprzedniego rozmiaru + rozmiar zmiennej, przechowującej nowo znalezioną liczbę
2. kopiować zawartość starego obszaru do nowego
3. zwalniać stary, nieużywany obszar pamięci
4. w ostatnim elemencie nowej tablicy zapisać znalezioną liczbę.

Cóż, trochę tutaj roboty jest, a kopiowanie całej zawartości jednego obszaru w drugi jest czasochłonne. W takim przypadku możemy użyć listy. Tworząc listę możemy w prosty sposób przechować nowo znalezione liczby. Przy użyciu listy nasze postępowanie ograniczy się do:

1. przydzielenia obszaru pamięci, aby przechować wartość obliczeń
2. dodać do listy nowy element

Prawda, że proste? Dodatkowo, lista zajmuje w pamięci tylko tyle pamięci, ile potrzeba na aktualną liczbę elementów. Pusta tablica zajmuje natomiast tyle samo miejsca co pełna tablica.

Implementacja listy

W języku C aby stworzyć listę musimy użyć struktur. Dlaczego? Ponieważ musimy przechować co najmniej dwie wartości:

1. pewną zmienną (np. liczbę pierwszą z przykładu)
2. wskaźnik na kolejny element listy

Przyjmijmy, że szukając liczb pierwszych nie przekroczymy możliwości typu unsigned long:

```
typedef struct element {
  struct element *next; /* wskaźnik na kolejny element listy */
  unsigned long val; /* przechowywana wartość */
} el_listy;
```

Zacznijmy zatem pisać nasz eksperymentalny program, do wyszukiwania liczb pierwszych. Pierwszą liczbą pierwszą jest liczba 2 Pierwszym elementem naszej listy będzie zatem struktura, która będzie przechowywała liczbę 2. Na co będzie wskazywało pole next? Ponieważ na początku działania programu będziemy mieć tylko jeden element listy, pole next powinno wskazywać na NULL. Umówmy się zatem, że pole next ostatniego elementu listy będzie wskazywało NULL — po tym poznamy, że lista się skończyła.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct element {
  struct element *next;
  unsigned long val;
} el_listy;

el_listy *first; /* pierwszy element listy */

int main ()
{
  unsigned long i = 3; /* szukamy liczb pierwszych w zakresie od 3 do 1000 */
  const unsigned long END = 1000;
  first = malloc (sizeof(el_listy));
  first->val = 2;
  first->next = NULL;
  for (;i<=END;++i) {
    /* tutaj powinien znajdować się kod, który sprawdza podzielność sprawdzanej liczby przez
       poprzednio znalezione liczby pierwsze oraz dodaje liczbę do listy w przypadku stwierdzenia,
       że jest ona liczbą pierwszą. */
    }
  wypisz_liste(first);
 return 0;
}
```

Na początek zajmiemy się wypisywaniem listy. W tym celu będziemy musieli "odwiedzić" każdy element listy. Elementy listy są połączone polem next, aby przeglądnąć listę użyjemy następującego algorytmu:

1. Ustaw wskaźnik roboczy na pierwszym elemencie listy
2. Jeśli wskaźnik ma wartość NULL, przerwij
3. Wypisz element wskazywany przez wskaźnik
4. Przesuń wskaźnik na element, który jest wskazywany przez pole next

5. Wróć do punktu 2

```
void wypisz_liste(el_listy *lista)
{
  el_listy *wsk=lista;           /* 1 */
  while( wsk != NULL )           /* 2 */
    {
    printf ("%lu\n", wsk->val); /* 3 */
    wsk = wsk->next;            /* 4 */
    }                              /* 5 */
}
```

Zastanówmy się teraz, jak powinien wyglądać kod, który dodaje do listy następny element. Taka funkcja powinna:

1. znaleźć ostatni element (tj. element, którego pole next == NULL)
2. przydzielić odpowiedni obszar pamięci
3. skopiować w pole val w nowo przydzielonym obszarze znalezioną liczbę pierwszą
4. nadać polu next ostatniego elementu listy wartość NULL
5. w pole next ostatniego elementu listy wpisać adres nowo przydzielonego obszaru

Napiszmy zatem odpowiednią funkcję:

```
void dodaj_do_listy (el_listy *lista, unsigned long liczba)
{
  el_listy *wsk, *nowy;
  wsk = lista;
  while (wsk->next != NULL)          /* 1 */
    {
    wsk = wsk->next; /* przesuwamy wsk aż znajdziemy ostatni element */
    }
  nowy = malloc (sizeof(el_listy));  /* 2 */
  nowy->val = liczba;                /* 3 */
  nowy->next = NULL;                 /* 4 */
  wsk->next = nowy;                  /* 5 */
}
```

I... to już właściwie koniec naszej funkcji (warto zwrócić uwagę, że funkcja w tej wersji zakłada, że na liście jest już przynajmniej jeden element). Wstaw ją do kodu przed funkcją main. Został nam jeszcze jeden problem: w pętli for musimy dodać kod, który odpowiednio będzie "badał" liczby oraz w przypadku stwierdzenia pierwszeństwa liczby, będzie dodawał ją do listy. Ten kod powinien wyglądać mniej więcej tak:

```
int jest_pierwsza(el_listy *lista, int liczba)
{
  el_listy *wsk;
  wsk = first;
  while (wsk != NULL) {
    if ((liczba % wsk->val)==0) return 0; /* jeśli reszta z dzielenia
        liczby przez którąkolwiek z poprzednio znalezionych
        liczb pierwszych jest równa zero, to znaczy, że liczba ta
        nie jest liczbą pierwszą */
    wsk = wsk->next;
    }
  /* natomiast jeśli sprawdzimy wszystkie poprzednio znalezione liczby
```

```
        i żadna z nich nie będzie dzieliła liczby i,
        możemy liczbę i dodać do listy liczb pierwszych */
  return 1;
}
...
for (;i<=END;++i) {
  if (jest_pierwsza(first, i))
    dodaj_do_listy (first,i);
    }
```

Podsumujmy teraz efekty naszej pracy. Oto cały kod naszego programu:

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

typedef struct element {
  struct element *next;
  unsigned long val;
} el_listy;

el_listy *first;

void dodaj_do_listy (el_listy *lista, unsigned long liczba)
{
  el_listy *wsk, *nowy;
  wsk = lista;
  while (wsk->next != NULL)
    {
    wsk = wsk->next; /* przesuwamy wsk aż znajdziemy ostatni element */
    }
  nowy = malloc (sizeof(el_listy));
  nowy->val = liczba;
  nowy->next = NULL;
  wsk->next = nowy; /* podczepiamy nowy element do ostatniego z listy */
}

void wypisz_liste(el_listy *lista)
{
  el_listy *wsk=lista;
  while( wsk != NULL )
    {
    printf ("%lu\n", wsk->val);
    wsk = wsk->next;
    }
}

int jest_pierwsza(el_listy *lista, int liczba)
{
  el_listy *wsk;
  wsk = first;
  while (wsk != NULL) {
    if ((liczba%wsk->val)==0) return 0;
       wsk = wsk->next;
    }
    return 1;
```

```
}

int main ()
{
  unsigned long i = 3; /* szukamy liczb pierwszych w zakresie od 3 do 1000 */
  const unsigned long END = 1000;
  first = malloc (sizeof(el_listy));
  first->val = 2;
  first->next = NULL;
  for (;i!=END;++i) {
    if (jest_pierwsza(first, i))
      dodaj_do_listy (first, i);
      }
  wypisz_liste(first);
  return 0;
}
```

Możemy jeszcze pomyśleć, jak można by wykonać usuwanie elementu z listy. Najprościej byłoby zrobić:

```
wsk->next = wsk->next->next
```

ale wtedy element, na który wskazywał wcześniej `wsk->next` przestaje być dostępny i zaśmieca pamięć. Trzeba go usunąć. Zauważmy, że aby usunąć element potrzebujemy wskaźnika do **elementu go poprzedzającego** (po to, by nie rozerwać listy). Popatrzmy na poniższą funkcję:

```
void usun_z_listy(el_listy *lista, int element)
{
  el_listy *wsk=lista;
  while (wsk->next != NULL)
    {
    if (wsk->next->val == element) /* musimy mieć wskaźnik do elementu poprzedzającego */
      {
      el_listy *usuwany=wsk->next; /* zapamiętujemy usuwany element */
      wsk->next = usuwany->next;   /* przestawiamy wskaźnik next by omijał usuwany element */
      free(usuwany);               /* usuwamy z pamięci */
      } else
        {
        wsk = wsk->next;           /* idziemy dalej tylko wtedy kiedy nie usuwaliśmy */
        }                          /* bo nie chcemy zostawić duplikatów */
      }
}
```

Funkcja ta jest tak napisana, by usuwała z listy wszystkie wystąpienia danego elementu (w naszym programie nie ma to miejsca, ale lista jest zrobiona tak, że może trzymać dowolne liczby). Zauważmy, że wskaźnik `wsk` jest przesuwany tylko wtedy, gdy nie kasowaliśmy. Gdybyśmy zawsze go przesuwali, przegapilibyśmy element gdyby występował kilka razy pod rząd.

Funkcja ta działa poprawnie tylko wtedy, gdy nie chcemy usuwać pierwszego elementu. Można to poprawić — dodając instrukcję warunkową do funkcji lub dodając do listy "głowę" — pierwszy element nie przechowujący niczego, ale upraszczający operacje na liście. Zostawiamy to do samodzielnej pracy.

Cały powyższy przykład omawiał tylko jeden przypadek listy — listę jednokierunkową. Jednak istnieją jeszcze inne typy list, np. lista jednokierunkowa cykliczna, lista dwukierunkowa oraz dwukierunkowa cykliczna. Różnią się one od siebie tylko tym, że:

- w przypadku list dwukierunkowych — w strukturze el_listy znajduje się jeszcze pole, które wskazuje na element poprzedni
- w przypadku list cyklicznych — ostatni element wskazuje na pierwszy (nie rozróżnia się wtedy elementu pierwszego, ani ostatniego)

Rozdział 20

Biblioteki

Czym jest biblioteka

Biblioteka jest to zbiór funkcji, które zostały wydzielone po to, aby dało się z nich korzystać w wielu programach. Ułatwia to programowanie — nie musimy np. sami tworzyć funkcji printf. Każda biblioteka posiada swoje pliki nagłówkowe, które zawierają deklaracje funkcji bibliotecznych oraz często zawarte są w nich komentarze, jak używać danej funkcji. W tej części podręcznika nauczymy się tworzyć nasze własne biblioteki.

Jak zbudowana jest biblioteka

Każda biblioteka składa się z co najmniej dwóch części:

- pliku nagłówkowego z deklaracjami funkcji (plik z rozszerzeniem .h)
- pliku źródłowego, zawierającego ciała funkcji (plik z rozszerzeniem .c)

Budowa pliku nagłówkowego

Oto najprostszy możliwy plik nagłówkowy:

```
#ifndef PLIK_H
#define PLIK_H
/* tutaj są wpisane deklaracje funkcji */
#endif /* PLIK_H */
```

Zapewne zapytasz się na co komu instrukcje `#ifndef`, `#define` oraz `#endif`. Otóż często się zdarza, że w programie korzystamy z plików nagłówkowych, które dołączają się wzajemnie. Oznaczałoby to, że w kodzie programu kilka razy pojawiła by się zawartość tego samego pliku nagłówkowego. Instrukcja `#ifndef` i `#define` temu zapobiega. Dzięki temu kompilator nie musi kilkakrotnie kompilować tego samego kodu.

W plikach nagłówkowych często umieszcza się też definicje typów, z których korzysta biblioteka albo np. makr.

Budowa najprostszej biblioteki

Załóżmy, że nasza biblioteka będzie zawierała jedną funkcję, która wypisuje na ekran tekst "pl.Wikibooks". Utwórzmy zatem nasz plik nagłówkowy:

```
#ifndef WIKI_H
#define WIKI_H
void wiki (void);
#endif
```

Należy pamiętać, o podaniu void w liście argumentów funkcji nie przyjmujących argumentów. O ile przy definicji funkcji nie trzeba tego robić (jak to często czyniliśmy w przypadku funkcji main) o tyle w prototypie brak słówka void oznacza, że w prototypie nie ma informacji na temat tego jakie argumenty funkcja przyjmuje.

Plik nagłówkowy zapisujemy jako "wiki.h". Teraz napiszmy ciało tej funkcji:

```
#include "wiki.h"
#include <stdio.h>

void wiki (void)
{
  printf ("pl.Wikibooks\n");
}
```

Ważne jest dołączenie na początku pliku nagłówkowego. Dlaczego? Plik nagłówkowy zawiera deklaracje naszych funkcji — jeśli popełniliśmy błąd i deklaracja nie zgadza się z definicją, kompilator od razu nas o tym powiadomi. Oprócz tego plik nagłówkowy może zawierać definicje istotnych typów lub makr.

Zapiszmy naszą bibliotekę jako plik "wiki.c". Teraz należy ją skompilować. Robi się to trochę inaczej, niż normalny program. Należy po prostu do opcji kompilatora gcc dodać opcję "-c":

```
gcc wiki.c -c -o wiki.o
```

Rozszerzenie ".o" jest domyślnym rozszerzeniem dla bibliotek statycznych (typowych bibliotek łączonych z resztą programu na etapie kompilacji). Teraz możemy spokojnie skorzystać z naszej nowej biblioteki. Napiszmy nasz program:

```
#include "wiki.h"

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

Zapiszmy program jako "main.c" Teraz musimy odpowiednio skompilować nasz program:

```
gcc main.c wiki.o -o main
```

Uruchamiamy nasz program:

```
./main
pl.Wikibooks
```

Jak widać nasza pierwsza biblioteka działa.

Zauważmy, że kompilatorowi podajemy i pliki z kodem źródłowym (main.c) i pliki ze skompilowanymi bibliotekami (wiki.o) by uzyskać plik wykonywalny (main). Jeśli nie podalibyśmy plików z bibliotekami, main.c co prawda skompilowałby się, ale błąd zostałby zgłoszony przez linker — część kompilatora odpowiedzialna za wstawienie w miejsce wywołań funkcji ich adresów (takiego adresu linker nie mógłby znaleźć).

Zmiana dostępu do funkcji i zmiennych (static i extern)

Język C, w przeciwieństwie do swego młodszego krewnego — C++ nie posiada praktycznie żadnych mechanizmów ochrony kodu biblioteki przed modyfikacjami. C++ ma w swoim asortymencie m.in. sterowanie uprawnieniami różnych elementów klasy. Jednak programista, piszący program w C nie jest tak do końca bezradny. Autorzy C dali mu do ręki dwa narzędzia: **extern** oraz **static**. Pierwsze z tych słów kluczowych informuje kompilator, że dana funkcja lub zmienna istnieje, ale w innym miejscu, i zostanie dołączona do kodu programu w czasie łączenia go z biblioteką.

extern przydaje się, gdy zmienna lub funkcja jest zadeklarowana w bibliotece, ale nie jest udostępniona na zewnątrz (nie pojawia się w pliku nagłówkowym). Przykładowo:

```
/* biblioteka.h */
extern char zmienna_dzielona[];

/* biblioteka.c */
#include "biblioteka.h"

char zmienna_dzielona[] = "Zawartosc";

/* main.c */
#include <stdio.h>
#include "biblioteka.h"

int main()
{
  printf("%s\n", zmienna_dzielona);
  return 0;
}
```

Gdybyśmy tu nie zastosowali **extern**, kompilator (nie linker) zaprotestowałby, że nie zna zmiennej *zmienna_dzielona*. Próba dopisania deklaracji `char zmienna_dzielona;` stworzyłaby nową zmienną i utracilibyśmy dostęp do interesującej nas zawartości.

Odwrotne działanie ma słowo kluczowe **static** użyte w tym kontekście (użyte wewnątrz bloku tworzy zmienną statyczną, więcej informacji w rozdziale Zmienne). Może ono odnosić się zarówno do zmiennych jak i do funkcji globalnych. Powoduje, że dana zmienna lub funkcja jest **niedostępna** na zewnątrz biblioteki[1]. Możemy dzięki temu ukryć np. funkcje, które używane są przez samą bibliotekę, by nie dało się ich wykorzystać przez **extern**.

[1] Tak naprawdę całe "ukrycie" funkcji polega na zmianie niektórych danych w pliku z kodem binarnym danej biblioteki (pliku .o), przez co linker powoduje wygenerowanie komunikatu o błędzie w czasie łączenia biblioteki z programem.

Rozdział 21

Więcej o kompilowaniu

Ciekawe opcje kompilatora GCC

- E — powoduje wygenerowanie kodu programu ze zmianami, wprowadzonymi przez preprocesor
- S — zamiana kodu w języku C na kod asemblera (komenda: gcc -S plik.c spowoduje utworzenie pliku o nazwie plik.s, w którym znajdzie się kod asemblera)
- c — kompilacja bez łączenia z bibliotekami
- Ikatalog — ustawienie domyślnego katalogu z plikami nagłówkowymi na *katalog*
- lbiblioteka — wymusza łączenie programu z podaną biblioteką (np. -lGL)

Program make

Dość często może się zdarzyć, że nasz program składa się z kilku plików źródłowych. Jeśli tych plików jest mało (np. 3-5) możemy jeszcze próbować ręcznie kompilować każdy z nich. Jednak jeśli tych plików jest dużo, lub chcemy pokazać nasz program innym użytkownikom musimy stworzyć elegancki sposób kompilacji naszego programu. Właśnie po to, aby zautomatyzować proces kompilacji powstał program **make**. Program make analizuje pliki Makefile i na ich podstawie wykonuje określone czynności.

Budowa pliku Makefile

Uwaga: poniżej został umówiony Makefile dla GNU Make. Istnieją inne programy make i mogą używać innej składni. Na Wikibooks został też obszernie opisany program make firmy Borland.

Najważniejszym elementem pliku Makefile są **zależności** oraz **reguły** przetwarzania. Zależności polegają na tym, że np. jeśli nasz program ma być zbudowany z 4 plików, to najpierw należy skompilować każdy z tych 4 plików, a dopiero później połączyć je w jeden cały program. Zatem zależności określają kolejność wykonywanych czynności. Natomiast reguły określają **jak** skompilować dany plik. Zależności tworzy się tak:

```
co: od_czego
  reguły...
```

Dzięki temu program make zna już kolejność wykonywanych działań oraz czynności, jakie ma wykonać. Aby zbudować "co" należy wykonać polecenie: `make co`. Pierwsza reguła w pliku Makefile jest regułą domyślną. Jeśli wydamy polecenie `make` bez parametrów, zostanie

zbudowana właśnie reguła domyślna. Tak więc dobrze jest jako pierwszą regułę wstawić regułę budującą końcowy plik wykonywalny; zwyczajowo regułę tą nazywa się `all`.

Należy pamiętać, by sekcji "co" nie wcinać, natomiast "reguły" wcinać dwoma spacjami. Część "od_czego" może być pusta.

Plik Makefile umożliwia też definiowanie pewnych zmiennych. Nie trzeba tutaj się już troszczyć o typ zmiennej, wystarczy napisać:

```
nazwa_zmiennej = wartość
```

W ten sposób możemy zadeklarować dowolnie dużo zmiennych. Zmienne mogą być różne — nazwa kompilatora, jego parametry i wiele innych. Zmiennej używamy w następujący sposób: `$(nazwa_zmiennej)`.

Komentarze w pliku Makefile tworzymy zaczynając linię od znaku hash (#).

Przykładowy plik Makefile

Dość tej teorii, teraz zajmiemy się działającym przykładem. Załóżmy, że nasz przykładowy program nazywa się test oraz składa się z czterech plików: pierwszy.c drugi.c trzeci.c czwarty.c

Odpowiedni plik Makefile powinien wyglądać mniej więcej tak:

```
# Mój plik makefile - wpisz 'make all' aby skompilować cały program
CC = gcc

all: pierwszy.o drugi.o trzeci.o czwarty.o
  $(CC) pierwszy.o drugi.o trzeci.o czwarty.o -o test

pierwszy.o: pierwszy.c
  $(CC) pierwszy.c -c -o pierwszy.o

drugi.o: drugi.c
  $(CC) drugi.c -c -o drugi.o

trzeci.o: trzeci.c
  $(CC) trzeci.c -c -o trzeci.o

czwarty.o: czwarty.c
  $(CC) czwarty.c -c -o czwarty.o
```

Widzimy, że nasz program zależy od 4 plików z rozszerzeniem .o (pierwszy.o itd.), potem każdy z tych plików zależy od plików .c, które program make skompiluje w pierwszej kolejności, a następnie połączy w jeden program (test). Nazwę kompilatora zapisaliśmy jako zmienną, ponieważ powtarza się i zmienna jest sposobem, by zmienić ją wszędzie za jednym zamachem.

Zatem jak widać używanie pliku Makefile jest bardzo proste. Warto na koniec naszego przykładu dodać regułę, która wyczyści katalog z plików .o:

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

Ta reguła spowoduje usunięcie wszystkich plików .o oraz naszego programu jeśli napiszemy `make clean`.

Możemy też ukryć wykonywane komendy albo dopisać własny opis czynności:

```
clean:
  @echo Usuwam gotowe pliki
  @rm -f *.o test
```

Ten sam plik Makefile mógłby wyglądać inaczej:

```
CFLAGS = -g -O # tutaj można dodawać inne flagi kompilatora
LIBS = -lm # tutaj można dodawać biblioteki

OBJ =\
    pierwszy.o \
    drugi.o \
    trzeci.o \
    czwarty.o

all: main

clean:
       rm -f *.o test
.c.o:
       $(CC) -c $(INCLUDES) $(CFLAGS) $<

main: $(OBJ)
       $(CC) $(OBJ) $(LIBS) -o test
```

Tak naprawdę jest to dopiero bardzo podstawowe wprowadzenie do używania programu make, jednak jest ono wystarczające, byś zaczął z niego korzystać. Wyczerpujące omówienie całego programu niestety przekracza zakres tego podręcznika.

Optymalizacje

Kompilator GCC umożliwia generację kodu zoptymalizowanego dla konkretnej architektury. Służą do tego opcje **-march=** i **-mtune=**. Stopień optymalizacji ustalamy za pomocą opcji **-Ox**, gdzie x jest numerem stopnia optymalizacji (od 1 do 3). Możliwe jest też użycie opcji **-Os**, która powoduje generowanie kodu o jak najmniejszym rozmiarze. Aby skompilować dany plik z optymalizacjami dla procesora Athlon XP, należy napisać tak:

```
gcc program.c -o program -march=athlon-xp -O3
```

Z optymalizacjami należy uważać, gdyż często zdarza się, że kod skompilowany bez optymalizacji działa zupełnie inaczej, niż ten, który został skompilowany z optymalizacjami.

Wyrównywanie

Wyrównywanie jest pewnym zjawiskiem, na które w bardzo wielu podręcznikach, mówiących o C w ogóle się nie wspomina. Ten rozdział ma za zadanie wyjaśnienie tego zjawiska oraz uprzedzenie programisty o pewnych faktach, które w późniejszej jego "twórczości" mogą zminimalizować czas na znalezienie pewnych informacji, które mogą wpływać na to, że jego program nie będzie działał poprawnie.

Często zdarza się, że kompilator w ramach optymalizacji "wyrównuje" elementy struktury tak, aby procesor mógł łatwiej odczytać i przetworzyć dane. Przyjrzyjmy się bliżej następującemu fragmentowi kodu:

```
typedef struct {
  unsigned char wiek; /* 8 bitów */
  unsigned short dochod; /* 16 bitów */
  unsigned char plec; /* 8 bitów */
} nasza_str;
```

Aby procesor mógł łatwiej przetworzyć dane kompilator może dodać do tej struktury jedno, ośmiobitowe pole. Wtedy struktura będzie wyglądała tak:

```
typedef struct {
  unsigned char wiek; /*8 bitów */
  unsigned char fill[1]; /* 8 bitów */
  unsigned short dochod; /* 16 bitów */
  unsigned char plec; /* 8 bitów */
} nasza_str;
```

Wtedy rozmiar zmiennych przechowujących wiek, płeć, oraz dochód będzie wynosił 64 bity — będzie zatem potęgą liczby dwa i procesorowi dużo łatwiej będzie tak ułożoną strukturę przechowywać w pamięci cache. Jednak taka sytuacja nie zawsze jest pożądana. Aby jej zapobiec w kompilatorze GNU GCC możemy użyć takiej oto linijki:

```
__attribute__ ((packed))
```

Dzięki użyciu tego atrybutu, kompilator zostanie "zmuszony" do braku ingerencji w naszą strukturę. Jest jednak jeszcze jeden, być może bardziej elegancki sposób na obejście dopełniania. Zauważyłeś, że dopełnienie, dodane przez kompilator pojawiło się między polem o długości 8 bitów (plec) oraz polem o długości 32 bitów (dochod). Wyrównywanie polega na tym, że dana zmienna powinna być umieszczona pod adresem będącym wielokrotnością jej rozmiaru. Oznacza to, że jeśli np. mamy w strukturze na początku dwie zmienne, o rozmiarze jednego bajta, a potem jedną zmienną, o rozmiarze 4 bajtów, to pomiędzy polami o rozmiarze 2 bajtów, a polem czterobajtowym pojawi się dwubajtowe dopełnienie. Może Ci się wydawać, że jest to tylko niepotrzebne mącenie w głowie, jednak niektóre architektury (zwłaszcza typu RISC) mogą nie wykonać kodu, który nie został wyrównany. Dlatego, naszą strukturę powinniśmy zapisać mniej więcej tak:

```
typedef struct {
  unsigned short dochod; /* 16 bitów */
  unsigned char wiek; /* 8 bitów */
  unsigned char plec; /* 8 bitów */
} nasza_str;
```

W ten sposób wyrównana struktura nie będzie podlegała modyfikacjom przez kompilator oraz będzie przenośna pomiędzy różnymi kompilatorami.

Wyrównywanie działa także na pojedynczych zmiennych w programie, jednak ten problem nie powoduje tyle zamieszania, co ingerencja kompilatora w układ pól struktury. Wyrównywanie zmiennych polega tylko na tym, że kompilator umieszcza je pod adresami, które są wielokrotnością ich rozmiaru

Kompilacja skrośna

Mając w domu dwa komputery, o odmiennych architekturach (np. i386 oraz Sparc) możemy potrzebować stworzyć program dla jednej maszyny, mając do dyspozycji tylko drugi komputer. Nie musimy wtedy latać do znajomego, posiadającego odpowiedni sprzęt. Możemy skorzystać z tzw. **kompilacji skrośnej** (ang. cross-compile). Polega ona na tym, że program nie jest kompilowany pod procesor, na którym działa kompilator, lecz na inną, zdefiniowaną wcześniej maszynę. Efekt będzie taki sam, a skompilowany program możemy bez problemu uruchomić na drugim komputerze.

Inne narzędzia

Wśród przydatnych narzędzi, warto wymienić również program objdump (zarówno pod Unix jak i pod Windows) oraz readelf (tylko Unix). Objdump służy do deasemblacji i analizy skompilowanych programów. Readelf służy do analizy pliku wykonywalnego w formacie ELF (używanego w większości systemów z rodziny Unix). Więcej informacji możesz uzyskać, pisząc (w systemach Unix):

```
man 1 objdump
man 1 readelf
```

Rozdział 22

Zaawansowane operacje matematyczne

Biblioteka matematyczna

Aby móc korzystać z wszystkich dobrodziejstw funkcji matematycznych musimy na początku dołączyć plik math.h:

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

A w procesie kompilacji (dotyczy kompilatora GCC) musimy niekiedy dodać flagę "-lm":

```
gcc plik.c -o plik -lm
```

Funkcje matematyczne, które znajdują się w bibliotece standardowej możesz znaleźć tutaj. Przy korzystaniu z nich musisz wziąć pod uwagę m.in. to, że biblioteka matematyczna prowadzi kalkulację w oparciu o radiany a nie stopnie.

Stałe matematyczne

W pliku math.h zdefiniowane są pewne stałe, które mogą być przydatne do obliczeń. Są to m.in.:

- M_E — podstawa logarytmu naturalnego
- M_LOG2E — logarytm o podstawie 2 z liczby e
- M_LOG10E — logarytm o podstawie 10 z liczby e
- M_LN2 — logarytm naturalny z liczby 2
- M_LN10 — logarytm naturalny z liczby 10
- M_PI — liczba π
- M_PI_2 — liczba $\pi/2$
- M_PI_4 — liczba $\pi/4$
- M_1_PI — liczba $1/\pi$
- M_2_PI — liczba $2/\pi$

Prezentacja liczb rzeczywistych w pamięci komputera

Być może ten temat może wydać Ci się niepotrzebnym, lecz w wielu książkach nie ma w ogóle tego tematu. Dzięki niemu zrozumiesz, jak komputer radzi sobie z przecinkiem oraz dlaczego niektóre obliczenia dają niezbyt dokładne wyniki. Na początek trochę teorii: do przechowywania liczb rzeczywistych przeznaczone są 3 typy: `float`, `double` oraz `long double`. Zajmują one odpowiednio 32, 64 oraz 80 bitów. Wiemy też, że komputer nie ma fizycznej możliwości zapisania przecinka. Spróbujmy teraz zapisać jakąś liczbę wymierną w formie liczb binarnych. Nasza liczba to powiedzmy 4.25. Spróbujmy ją rozbić na sumę potęg dwójki: $4 = 1*2^2 + 0*2^1+0*2^0$. Dobra — rozpisaliśmy liczbę 4, ale co z częścią dziesiętną? Skorzystajmy z zasad matematyki — $0.25 = 2^{-2}$. Zatem nasza liczba powinna wyglądać tak: 100.01

Ponieważ komputer nie jest w stanie przechować pozycji przecinka, ktoś wpadł na prosty ale sprytny pomysł ustawienia przecinka jak najbliżej początku liczby i tylko mnożenia jej przez odpowiednią potęgę dwójki. Taki sposób przechowywania liczb nazywamy **zmiennoprzecinkowym**, a proces przekształcania naszej liczby z postaci czytelnej przez człowieka na format zmiennoprzecinkowy nazywamy **normalizacją**. Wróćmy do naszej liczby — 4.25. W postaci binarnej wygląda ona tak: 100.01, natomiast po normalizacji będzie wyglądała tak: $1.0001*2^2$. W ten sposób w pamięci komputera znajdą się dwie informacje: liczba zakodowana w pamięci z "wirtualnym" przecinkiem oraz numer potęgi dwójki. Te dwie informacje wystarczają do przechowania wartości liczby. Jednak pojawia się inny problem — co się stanie, jeśli np. będziemy chcieli przełożyć liczbę typu $\frac{1}{3}$? Otóż tutaj wychodzą na wierzch pewne niedociągnięcia komputera w dziedzinie samej matematyki. 1/3 daje w rozwinięciu dziesiętnym 0.(3). Jak zatem zapisać taką liczbę? Otóż nie możemy przechować całego jej rozwinięcia (wynika to z ograniczeń typu danych — ma on niestety skończoną liczbę bitów). Dlatego przechowuje się tylko pewne przybliżenie liczby. Jest ono tym bardziej dokładne im dany typ ma więcej bitów. Zatem do obliczeń wymagających dokładnych danych powinniśmy użyć typu double lub long double. Na szczęście w większości przeciętnych programów tego typu problemy zwykle nie występują. A ponieważ początkujący programista nie odpowiada za tworzenie programów sterujących np. lotem statku kosmicznego, więc drobne przekłamania na odległych miejscach po przecinku nie stanowią większego problemu.

Liczby zespolone

Operacje na liczbach zespolonych są częścią uaktualnionego standardu języka C o nazwie C99, który jest obsługiwany jedynie przez część kompilatorów

Podane tutaj informacje zostały sprawdzone na systemie Gentoo Linux z biblioteką GNU libc w wersji 2.3.5 i kompilatorem GCC w wersji 4.0.2

Dotychczas korzystaliśmy tylko z liczb rzeczywistych, lecz najnowsze standardy języka C umożliwiają korzystanie także z innych liczb — np. z liczb zespolonych.

Aby móc korzystać z liczb zespolonych w naszym programie należy w nagłówku programu umieścić następującą linijkę:

```
#include <complex.h>
```

Wiemy, że liczba zespolona zdeklarowana jest następująco:

```
z = a+b*i, gdzie a, b są liczbami rzeczywistymi, a i*i=(-1).
```

W pliku complex.h liczba i zdefiniowana jest jako I. Zatem wypróbujmy możliwości liczb zespolonych:

```
#include <math.h>
#include <complex.h>
#include <stdio.h>

int main ()
{
  float _Complex z = 4+2.5*I;
  printf ("Liczba z: %f+%fi\n", creal(z), cimag (z));
  return 0;
}
```

następnie kompilujemy nasz program:

```
gcc plik1.c -o plik1 -lm
```

Po wykonaniu naszego programu powinniśmy otrzymać:

```
Liczba z: 4.00+2.50i
```

W programie zamieszczonym powyżej użyliśmy dwóch funkcji — creal i cimag.

- creal — zwraca część rzeczywistą liczby zespolonej
- cimag — zwraca część urojoną liczby zespolonej

Rozdział 23

Powszechne praktyki

Rozdział ten ma za zadanie pokazać powszechnie stosowane metody programowania w C. Nie będziemy tu uczyć, jak należy stawiać nawiasy klamrowe ani który sposób nazewnictwa zmiennych jest najlepszy — prowadzone są o to spory, z których niewiele wynika. Zaprezentowane tu rozwiązania mają konkretny wpływ na jakość tworzonych programów.

Konstruktory i destruktory

W większości obiektowych języków programowania obiekty nie mogą być tworzone bezpośrednio — obiekty otrzymuje się wywołując specjalną metodę danej klasy, zwaną konstruktorem. Konstruktory są ważne, ponieważ pozwalają zapewnić obiektowi odpowiedni stan początkowy. Destruktory, wywoływane na końcu czasu życia obiektu, są istotne, gdy obiekt ma wyłączny dostęp do pewnych zasobów i konieczne jest upewnienie się, czy te zasoby zostaną zwolnione.

Ponieważ C nie jest językiem obiektowym, nie ma wbudowanego wsparcia dla konstruktorów i destruktorów. Często programiści bezpośrednio modyfikują tworzone obiekty i struktury. Jednakże prowadzi to do potencjalnych błędów, ponieważ operacje na obiekcie mogą się nie powieść lub zachować się nieprzewidywalnie, jeśli obiekt nie został prawidłowo zainicjalizowany. Lepszym podejściem jest stworzenie funkcji, która tworzy instancję obiektu, ewentualnie przyjmując pewne parametry:

```
struct string {
  size_t size;
  char *data;
};

struct string *create_string(const char *initial) {
  assert (initial != NULL);
  struct string *new_string = malloc(sizeof(*new_string));
  if (new_string != NULL) {
    new_string->size = strlen(initial);
    new_string->data = strdup(initial);
    }
  return new_string;
}
```

Podobnie, bezpośrednie usuwanie obiektów może nie do końca się udać, prowadząc do wycieku zasobów. Lepiej jest użyć destruktora:

```
void free_string(struct string *s)
{
  assert (s != NULL);
  free(s->data);  /* zwalniamy pamięć zajmowaną przez strukturę */
  free(s);        /* usuwamy samą strukturę */
}
```

Często łączy się destruktory z zerowaniem zwolnionych wskaźników.

Czasami dobrze jest ukryć definicję obiektu, żeby mieć pewność, że użytkownicy nie utworzą go ręcznie. Aby to zapewnić struktura jest definiowana w pliku źródłowym (lub prywatnym nagłówku niedostępnym dla użytkowników) zamiast w pliku nagłówkowym, a deklaracja wyprzedzająca jest umieszczona w pliku nagłówkowym:

```
struct string;
struct string *create_string(const char *initial);
void free_string(struct string *s);
```

Zerowanie zwolnionych wskaźników

Jak powiedziano już wcześniej, po wywołaniu `free()` dla wskaźnika, staje się on "wiszącym wskaźnikiem". Co gorsze, większość nowoczesnych platform nie potrafi wykryć, kiedy taki wskaźnik jest używany zanim zostanie ponownie przypisany.

Jednym z prostych rozwiązań tego problemu jest zapewnienie, że każdy wskaźnik jest zerowany natychmiast po zwolnieniu:

```
free(p);
p = NULL;
```

Inaczej niż w przypadku "wiszących wskaźników", na wielu nowoczesnych architekturach przy próbie użycia wyzerowanego wskaźnika pojawi się sprzętowy wyjątek. Dodatkowo, programy mogą zawierać sprawdzanie błędów dla zerowych wartości, ale nie dla "wiszących wskaźników". Aby zapewnić, że jest to wykonywane dla każdego wskaźnika, możemy użyć makra:

```
#define FREE(p)   do { free(p); (p) = NULL; } while(0)
```

(aby zobaczyć, dlaczego makro jest napisane w ten sposób, zobacz Konwencje pisania makr)

Przy wykorzystaniu tej techniki destruktory powinny zerować wskaźnik, który przekazuje się do nich, więc argument musi być do nich przekazywany przez referencję. Na przykład, oto zaktualizowany destruktor z sekcji Konstruktory i destruktory:

```
void free_string(struct string **s)
{
  assert(s != NULL  &&  *s != NULL);
  FREE((*s)->data);  /* zwalniamy pamięć zajmowaną przez strukturę */
  FREE(*s);          /* usuwamy strukturę */
}
```

Niestety, ten idiom nie jest wstanie pomóc w wypadku wskazywania przez inne wskaźniki zwolnionej pamięci. Z tego powodu niektórzy eksperci C uważają go za niebezpieczny, jako kreujący fałszywe poczucie bezpieczeństwa.

Konwencje pisania makr

Ponieważ makra preprocesora działają na zasadzie zwykłego zastępowania napisów, są podatne na wiele kłopotliwych błędów, z których części można uniknąć przez stosowanie się do poniższych reguł:

1. Umieszczaj nawiasy dookoła argumentów makra kiedy to tylko możliwe. Zapewnia to, że gdy są wyrażeniami kolejność działań nie zostanie zmieniona. Na przykład:
   - Źle: `#define kwadrat(x) (x*x)`
   - Dobrze: `#define kwadrat(x) ( (x)*(x) )`
   - **Przykład:** Załóżmy, że w programie makro kwadrat() zdefiniowane bez nawiasów zostało wywołane następująco: `kwadrat(a+b)`. Wtedy zostanie ono zamienione przez preprocesor na: `a+b*a+b`. Z kolejności działań wiemy, że najpierw zostanie wykonane mnożenie, więc wartość wyrażenia `kwadrat(a+b)` będzie różna od kwadratu wyrażenia `a+b`.
2. Umieszczaj nawiasy dookoła całego makra, jeśli jest pojedynczym wyrażeniem. Ponownie, chroni to przed zaburzeniem kolejności działań.
   - Źle: `#define kwadrat(x) (x)*(x)`
   - Dobrze: `#define kwadrat(x) ( (x)*(x) )`
   - **Przykład:** Definiujemy makro `#define suma(a, b) (a)+(b)` i wywołujemy je w kodzie `wynik = suma(3, 4) * 5`. Makro zostanie rozwinięte jako `wynik = 3+4*5`, co — z powodu kolejności działań — da wynik inny niż pożądany.
3. Jeśli makro składa się z wielu instrukcji lub deklaruje zmienne, powinno być umieszczone w pętli `do { ... } while(0)`, bez kończącego średnika. Pozwala to na użycie makra jak pojedynczej instrukcji w każdym miejscu, jak ciało innego wyrażenia, pozwalając jednocześnie na umieszczenie średnika po makrze bez tworzenia zerowego wyrażenia. Należy uważać, by zmienne w makrze potencjalnie nie kolidowały z argumentami makra.
   - Źle: `#define FREE(p) free(p); p = NULL;`
   - Dobrze: `#define FREE(p) do { free(p); p = NULL; } while(0)`
4. Unikaj używania argumentów makra więcej niż raz wewnątrz makra. Może to spowodować kłopoty, gdy argument makra ma efekty uboczne (np. zawiera operator inkrementacji).
   - **Przykład:** `#define kwadrat(x) ((x)*(x))` nie powinno być wywoływane z operatorem inkrementacji `kwadrat(a++)` ponieważ zostanie to rozwinięte jako `((a++) * (a++))`, co jest niezgodne ze specyfikacją języka i zachowanie takiego wyrażenia jest niezdefiniowane (dwukrotna inkrementacja w tym samym wyrażeniu).
5. Jeśli makro może być w przyszłości zastąpione przez funkcję, rozważ użycie w nazwie małych liter, jak w funkcji.

Jak dostać się do konkretnego bitu?

Wiemy, że komputer to maszyna, której najmniejszą jednostką pamięci jest bit, jednak w C najmniejsza zmienna ma rozmiar 8 bitów (czyli jednego bajtu). Jak zatem można odczytać wartość pojedynczych bitów? W bardzo prosty sposób — w zestawie operatorów języka C znajdują się tzw. **operatory bitowe**. Są to m. in.:

- & — logiczne "i"
- — — logiczne "lub"

- — logiczne "nie"

Oprócz tego są także przesunięcia (« oraz »). Zastanówmy się teraz, jak je wykorzystać w praktyce. Załóżmy, że zajmujemy się jednobajtową zmienną.

```
unsigned char i = 2;
```

Z matematyki wiemy, że zapis binarny tej liczby wygląda tak (w ośmiobitowej zmiennej): 00000010. Jeśli teraz np. chcielibyśmy "zapalić" drugi bit od lewej (tj. bit, którego zapalenie niejako "doda" do liczby wartość 2^6) powinniśmy użyć logicznego lub:

```
unsigned char i = 2;
i |= 64;
```

Gdzie $64=2^6$. Odczytywanie wykonuje się za pomocą tzw. maski bitowej. Polega to na:

1. wyzerowaniu bitów, które są nam w danej chwili niepotrzebne
2. odpowiedniemu przesunięciu bitów, dzięki czemu szukany bit znajdzie się na pozycji pierwszego bitu od prawej

Do "wyłuskania" odpowiedniego bitu możemy posłużyć się operacją "i" — czyli operatorem **&**. Wygląda to analogicznie do posługiwania się operatorem "lub":

```
unsigned char i = 3; /* bitowo: 00000011 */
unsigned char temp = 0;
temp = i & 1; /* sprawdzamy najmniej znaczący bit - czyli pierwszy z prawej */
if (temp) {
  printf ("bit zapalony");
else {
  printf ("bit zgaszony");
}
```

Jeśli nie władasz biegle kodem binarnym, tworzenie masek bitowych ułatwią ci przesunięcia bitowe. Aby uzyskać liczbę która ma zapalony bit o numerze `n` (bity są liczone od zera), przesuwamy bitowo w lewo jedynkę o `n` pozycji:

```
1 << n
```

Jeśli chcemy uzyskać liczbę, w której zapalone są bity na pozycjach `l, m, n` — używamy sumy logicznej ("lub"):

```
(1 << l) | (1 << m) | (1 << n)
```

Jeśli z kolei chcemy uzyskać liczbę gdzie zapalone są wszystkie bity poza `n`, odwracamy ją za pomocą operatora logicznej negacji

```
~(1 << n)
```

Warto władać biegle operacjami na bitach, ale początkujący mogą (po uprzednim przeanalizowaniu) zdefiniować następujące makra i ich używać:

```
/* Sprawdzenie czy w liczbie k jest zapalony bit n */
#define IS_BIT_SET(k, n)     ((k) & (1 << (n)))

/* Zapalenie bitu n w zmiennej k */
#define SET_BIT(k, n)        (k |= (1 << (n)))

/* Zgaszenie bitu n w zmiennej k */
#define RESET_BIT(k, n)      (k &= ~(1 << (n)))
```

Skróty notacji

Istnieją pewne sposoby ograniczenia ilości niepotrzebnego kodu. Przykładem może być wykonywanie jednej operacji w razie wystąpienia jakiegoś warunku, np. zamiast pisać:

```
if (warunek) {
  printf ("Warunek prawdziwy\n");
  }
```

możesz skrócić notację do:

```
if (warunek)
  printf ("Warunek prawdziwy\n");
```

Podobnie jest w przypadku pętli for:

```
for (;warunek;)
  printf ("Wyświetlam się w pętli!\n");
```

Niestety ograniczeniem w tym wypadku jest to, że można w ten sposób zapisać tylko jedną instrukcję.

Rozdział 24

Przenośność programów

Jak dowiedziałeś się z poprzednich rozdziałów tego podręcznika, język C umożliwia tworzenie programów, które mogą być uruchamiane na różnych platformach sprzętowych pod warunkiem ich powtórnej kompilacji. Język C należy do grupy języków wysokiego poziomu, które tłumaczone są do poziomu kodu maszynowego (tzn. kod źródłowy jest kompilowany). Z jednej strony jest to korzystne posunięcie, gdyż programy są szybsze i mniejsze niż programy napisane w językach interpretowanych (takich, w których kod źródłowy nie jest kompilowany do kodu maszynowego, tylko *na bieżąco* interpretowany przez tzw. interpreter). Jednak istnieje także druga strona medalu — pewne zawiłości sprzętu, które ograniczają przenośność programów. Ten rozdział ma wyjaśnić Ci mechanizmy działania sprzętu w taki sposób, abyś bez problemu mógł tworzyć poprawne i całkowicie przenośne programy.

Niezdefiniowane zachowanie i zachowanie zależne od implementacji

W trakcie czytania kolejnych rozdziałów można było się natknąć na zwroty takie jak zachowanie niezdefiniowane (ang. *undefined behaviour*) czy zachowanie zależne od implementacji (ang. *implementation-defined behaviour*). Cóż one tak właściwie oznaczają?

Zacznijmy od tego drugiego. Autorzy standardu języka C czuli, że wymuszanie jakiegoś konkretnego działania danego wyrażenia byłoby zbytnim obciążeniem dla osób piszących kompilatory, gdyż dany wymóg mógłby być bardzo trudny do zrealizowania na konkretnej architekturze. Dla przykładu, gdyby standard wymagał, że typ unsigned char ma dokładnie 8 bitów to napisanie kompilatora dla architektury, na której bajt ma 9 bitów byłoby cokolwiek kłopotliwe, a z pewnością wynikowy program działałby o wiele wolniej niżby to było możliwe.

Z tego właśnie powodu, niektóre aspekty języka nie są określone bezpośrednio w standardzie i są pozostawione do decyzji zespołu (osoby) piszącego konkretną implementację. W ten sposób, nie ma żadnych przeciwwskazań (ze strony standardu), aby na architekturze, gdzie bajty mają 9 bitów, typ char również miał tyle bitów. Dokonany wybór musi być jednak opisany w dokumentacji kompilatora, tak żeby osoba pisząca program w C mogła sprawdzić jak dana konstrukcja zadziała.

Należy zatem pamiętać, że poleganie na jakimś konkretnym działaniu programu w przypadkach zachowania zależnego od implementacji drastycznie zmniejsza przenośność kodu źródłowego.

Zachowania niezdefiniowane są o wiele groźniejsze, gdyż zaistnienie takowego może spowodować dowolny efekt, który nie musi być nigdzie udokumentowany. Przykładem może tutaj być próba odwołania się do wartości wskazywanej przez wskaźnik o wartości NULL.

Jeżeli gdzieś w naszym programie zaistnieje sytuacja niezdefiniowanego zachowania, to nie jest już to kwestia przenośności kodu, ale po prostu błędu w kodzie, chyba że świado-

mie korzystamy z rozszerzenia naszego kompilatora. Rozważmy odwoływanie się do wartości wskazywanej przez wskaźnik o wartości NULL. Ponieważ według standardu operacja taka ma niezdefiniowany skutek to w szczególności może wywołać jakąś z góry określoną funkcję — kompilator może coś takiego zrealizować sprawdzając wartość wskaźnika przed każdą dereferencją, w ten sposób niezdefiniowane zachowanie dla konkretnego kompilatora stanie się jak najbardziej zdefiniowane.

Sytuacją wziętą z życia są operatory przesunięć bitowych, gdy działają na liczbach ze znakiem. Konkretnie przesuwanie w lewo liczb jest dla wielu przypadków niezdefiniowane. Bardzo często jednak, w dokumentacji kompilatora działanie przesunięć bitowych jest dokładnie opisane. Jest to o tyle interesujący fakt, iż wielu programistów nie zdaje sobie z niego sprawy i nieświadomie korzysta z rozszerzeń kompilatora.

Istnieje jeszcze trzecia klasa zachowań. Zachowania nieokreślone (ang. *unspecified behaviour*). Są to sytuacje, gdy standard określa kilka możliwych sposobów w jaki dane wyrażenie może działać i pozostawia kompilatorowi decyzję co z tym dalej zrobić. Coś takiego nie musi być nigdzie opisane w dokumentacji i znowu poleganie na konkretnym zachowaniu jest błędem. Klasycznym przykładem może być kolejność obliczania argumentów wywołania funkcji.

Rozmiar zmiennych

Rozmiar poszczególnych typów danych (np. char, int czy long) jest różna na różnych platformach, gdyż nie jest definiowany w sztywny sposób, jak np. "long int zawsze powinien mieć 64 bity" (takie określenie wiązałoby się z wyżej opisanymi trudnościami), lecz w na zasadzie zależności typu "long powinien być nie krótszy niż int", "short nie powinien być dłuższy od int". Pierwsza standaryzacja języka C zakładała, że typ int będzie miał taki rozmiar, jak domyślna długość liczb całkowitych na danym komputerze, natomiast modyfikatory short oraz long zmieniały długość tego typu tylko wtedy, gdy dana maszyna obsługiwała typy o mniejszej lub większej długości[1].

Z tego powodu, nigdy nie zakładaj, że dany typ będzie miał określony rozmiar. Jeżeli potrzebujesz typu o konkretnym rozmiarze (a dokładnej konkretnej liczbie bitów wartości) możesz skorzystać z pliku nagłówkowego stdint.h wprowadzonego do języka przez standard ISO C z 1999 roku. Definiuje on typy int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t i uint64_t (o ile w danej architekturze występują typy o konkretnej liczbie bitów).

Jednak możemy posiadać implementację, która nie posiada tego pliku nagłówkowego. W takiej sytuacji nie pozostaje nam nic innego jak tworzyć własny plik nagłówkowy, w którym za pomocą słówka **typedef** sami zdefiniujemy potrzebne nam typy. Np.:

```
typedef unsigned char      u8;
typedef   signed char      s8;
typedef unsigned short     u16;
typedef   signed short     s16;
typedef unsigned long      u32;
typedef   signed long      s32;
typedef unsigned long long u64;
typedef   signed long long s64;
```

Aczkolwiek należy pamiętać, że taki plik będzie trzeba pisać od nowa dla każdej architektury na jakiej chcemy kompilować nasz program.

[1] Dokładniejszy opis rozmiarów dostępny jest w rozdziale Składnia.

Porządek bajtów i bitów

Bajty i słowa

Wiesz zapewne, że podstawową jednostką danych jest bit, który może mieć wartość 0 lub 1. Kilka kolejnych bitów [2] stanowi bajt (dla skupienia uwagi, przyjmijmy, że bajt składa się z 8 bitów). Często typ short ma wielkość dwóch bajtów i wówczas pojawia się pytanie w jaki sposób są one zapisane w pamięci — czy najpierw ten bardziej znaczący — **big-endian**, czy najpierw ten mniej znaczący — **little-endian**.

Skąd takie nazwy? Otóż pochodzą one z książki *Podróże Guliwera*, w której liliputy kłóciły się o stronę, od której należy rozbijać jajko na twardo. Jedni uważali, że trzeba je rozbijać od grubszego końca (big-endian) a drudzy, że od cieńszego (little-endian). Nazwy te są o tyle trafne, że w wypadku procesorów wybór kolejności bajtów jest sprawą czysto polityczną, która jest technicznie neutralna.

Sprawa się jeszcze bardziej komplikuje w przypadku typów, które składają się np. z 4 bajtów. Wówczas są aż 24 (4 silnia) sposoby zapisania kolejnych fragmentów takiego typu. W praktyce zapewne spotkasz się jedynie z kolejnościami big-endian lub little-endian, co nie zmienia faktu, że inne możliwości także istnieją i przy pisaniu programów, które mają być przenośne należy to brać pod uwagę.

Poniższy przykład dobrze obrazuje oba sposoby przechowywania zawartości zmiennych w pamięci komputera (przyjmujemy CHAR_BIT == 8 oraz sizeof(long) == 4, bez bitów wypełnienia (ang. *padding bits*)): `unsigned long zmienna = 0x01020304;` w pamięci komputera będzie przechowywana tak:

```
adres         | 0  | 1  | 2  | 3  |
big-endian    |0x01|0x02|0x03|0x04|
little-endian |0x04|0x03|0x02|0x01|
```

Konwersja z jednego porządku do innego

Czasami zdarza się, że napisany przez nas program musi się komunikować z innym programem (może też przez nas napisanym), który działa na komputerze o (potencjalnie) innym porządku bajtów. Często najprościej jest przesyłać liczby jako tekst, gdyż jest on niezależny od innych czynników, jednak taki format zajmuje więcej miejsca, a nie zawsze możemy sobie pozwolić na taką rozrzutność.

Przykładem może być komunikacja sieciowa, w której przyjęło się, że dane przesyłane są w porządku big-endian. Aby móc łatwo operować na takich danych, w standardzie POSIX zdefiniowano następujące funkcje (w zasadzie zazwyczaj są to makra):

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Pierwsze dwie konwertują liczbę z reprezentacji lokalnej na reprezentację big-endian (*host to network*), natomiast kolejne dwie dokonują konwersji w drugą stronę (*network to host*).

Można również skorzystać z pliku nagłówkowego endian.h, w którym definiowane są makra pozwalające określić porządek bajtów:

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

[2] Standard wymaga aby było ich co najmniej 8 i liczba bitów w bajcie w konkretnej implementacji jest określona przez makro CHAR_BIT zdefiniowane w pliku nagłówkowym limits.h

```
int main() {
#if __BYTE_ORDER == __BIG_ENDIAN
  printf("Porządek big-endian (4321)\n");
#elif __BYTE_ORDER == __LITTLE_ENDIAN
  printf("Porządek little-endian (1234)\n");
#elif defined __PDP_ENDIAN && __BYTE_ORDER == __PDP_ENDIAN
  printf("Porządek PDP (3412)\n");
#else
  printf("Inny porządek (%d)\n", __BYTE_ORDER);
#endif
  return 0;
}
```

Na podstawie makra __BYTE_ORDER można skonstruować funkcję, która będzie konwertować liczby pomiędzy porządkiem różnymi porządkami:

```
#include <endian.h>
#include <stdio.h>
#include <stdint.h>

uint32_t convert_order32(uint32_t val, unsigned from, unsigned to) {
  if (from==to) {
    return val;
  } else {
    uint32_t ret = 0;
    unsigned char tmp[5] = { 0, 0, 0, 0, 0 };
    unsigned char *ptr = (unsigned char*)&val;
    unsigned div = 1000;
    do tmp[from / div % 10] = *ptr++; while ((div /= 10));
    ptr = (unsigned char*)&ret;
    div = 1000;
    do *ptr++ = tmp[to / div % 10]; while ((div /= 10));
    return ret;
  }
}

#define LE_TO_H(val)  convert_order32((val), 1234, __BYTE_ORDER)
#define H_TO_LE(val)  convert_order32((val), __BYTE_ORDER, 1234)
#define BE_TO_H(val)  convert_order32((val), 4321, __BYTE_ORDER)
#define H_TO_BE(val)  convert_order32((val), __BYTE_ORDER, 4321)
#define PDP_TO_H(val) convert_order32((val), 3412, __BYTE_ORDER)
#define H_TO_PDP(val) convert_order32((val), __BYTE_ORDER, 3412)

int main ()
{
  printf("%08x\n", LE_TO_H(0x01020304));
  printf("%08x\n", H_TO_LE(0x01020304));
  printf("%08x\n", BE_TO_H(0x01020304));
  printf("%08x\n", H_TO_BE(0x01020304));
  printf("%08x\n", PDP_TO_H(0x01020304));
  printf("%08x\n", H_TO_PDP(0x01020304));
  return 0;
}
```

Ciągle jednak polegamy na niestandardowym pliku nagłówkowym endian.h. Można go wyeliminować sprawdzając porządek bajtów w czasie wykonywania programu:

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

int main() {
  uint32_t val = 0x04030201;
  unsigned char *v = (unsigned char *)&val;
  int byte_order = v[0] * 1000 + v[1] * 100 + v[2] * 10 + v[3];

  if (byte_order == 4321) {
    printf("Porządek big-endian (4321)\n");
  } else if (byte_order == 1234) {
    printf("Porządek little-endian (1234)\n");
  } else if (byte_order == 3412) {
    printf("Porządek PDP (3412)\n");
  } else {
    printf("Inny porządek (%d)\n", byte_order);
  }
  return 0;
}
```

Powyższe przykłady opisują jedynie część problemów jakie mogą wynikać z próby przenoszenia binarnych danych pomiędzy wieloma platformami. Wszystkie co więcej zakładają, że bajt ma 8 bitów, co wcale nie musi być prawdą dla konkretnej architektury, na którą piszemy aplikację. Co więcej liczby mogą posiadać w swojej reprezentacje bity wypełnienia (ang. *padding bits*), które nie biorą udziały w przechowywaniu wartości liczby. Te wszystkie różnice mogą dodatkowo skomplikować kod. Toteż należy być świadomym, iż przenosząc dane binarnie musimy uważać na różne reprezentacje liczb.

Biblioteczne problemy

Pisząc programy nieraz będziemy musieli korzystać z różnych bibliotek. Problem polega na tym, że nie zawsze będą one dostępne na komputerze, na którym inny użytkownik naszego programu będzie próbował go kompilować. Dlatego też ważne jest, abyśmy korzystali z łatwo dostępnych bibliotek, które dostępne są na wiele różnych systemów i platform sprzętowych. **Zapamiętaj**: Twój program jest na tyle przenośny na ile przenośne są biblioteki z których korzysta!

Kompilacja warunkowa

Przy zwiększaniu przenośności kodu może pomóc preprocessor. Przyjmijmy np., że chcemy korzystać ze słówka kluczowego inline wprowadzonego w standardzie C99, ale równocześnie chcemy, aby nasz program był rozumiany przez kompilatory ANSI C. Wówczas, możemy skorzystać z następującego kodu:

```
#ifndef __inline__
# if __STDC_VERSION__ >= 199901L
#  define __inline__ inline
# else
#  define __inline__
# endif
#endif
```

a w kodzie programu zamiast słówka inline stosować __inline__. Co więcej, kompilator GCC rozumie słówka kluczowe tak tworzone i w jego przypadku warto nie redefiniować ich wartości:

```
#ifndef __GNUC__
# ifndef __inline__
#  if __STDC_VERSION__ >= 199901L
#   define __inline__ inline
#  else
#   define __inline__
#  endif
# endif
#endif
```

Korzystając z kompilacji warunkowej można także korzystać z różnego kodu zależnie od (np.) systemu operacyjnego. Przykładowo, przed kompilacją na konkretnej platformie tworzymy odpowiedni plik config.h, który następnie dołączamy do wszystkich plików źródłowych, w których podejmujemy decyzje na podstawie zdefiniowanych makr. Dla przykładu, plik config.h:

```
#ifndef CONFIG_H
#define CONFIG_H

/* Uncomment if using Windows */
/* #define USE_WINDOWS */

/* Uncomment if using Linux */
/* #define USE_LINUX */

#error You must edit config.h file
#error Edit it and remove those error lines

#endif
```

Jakiś plik źródłowy:

```
#include "config.h"

/* ... */

#ifdef USE_WINDOWS
  rob_cos_wersja_dla_windows();
#else
  rob_cos_wersja_dla_linux();
#endif
```

Istnieją różne narzędzia, które pozwalają na automatyczne tworzenie takich plików config.h, dzięki czemu użytkownik przed skompilowaniem programu nie musi się trudzić i edytować ich ręcznie, a jedynie uruchomić odpowiednie polecenie. Przykładem jest zestaw autoconf i automake.

Rozdział 25

Łączenie z innymi językami

Programista, pisząc jakiś program ma problem z wyborem najbardziej odpowiedniego języka do utworzenia tego programu. Niekiedy zdarza się, że najlepiej byłoby pisać program, korzystając z różnych języków. Język C może być z łatwością łączony z innymi językami programowania, które podlegają kompilacji bezpośrednio do kodu maszynowego (Asembler, Fortran czy też C++). Ponadto dzięki specjalnym bibliotekom można go łączyć z językami bardzo wysokiego poziomu (takimi jak np. Python czy też Ruby). Ten rozdział ma za zadanie wytłumaczyć Ci, w jaki sposób można mieszać różne języki programowania w jednym programie.

Język C i Asembler

Informacje zawarte w tym rozdziale odnoszą się do komputerów z procesorem i386 i pokrewnych.

Łączenie języka C i języka asemblera jest dość powszechnym zjawiskiem. Dzięki możliwości połączenia obu tych języków programowania można było utworzyć bibliotekę dla języka C, która niskopoziomowo komunikuje się z jądrem systemu operacyjnego komputera. Ponieważ zarówno asembler jak i C są językami tłumaczonymi do poziomu kodu maszynowego, za ich łączenie odpowiada program zwany **linkerem** (popularny ld). Ponadto niektórzy producenci kompilatorów umożliwiają stosowanie tzw. **wstawek asemblerowych**, które umieszcza się bezpośrednio w kodzie programu, napisanego w języku C. Kompilator, kompilując taki kod wstawi w miejsce tychże wstawek odpowiedni kod maszynowy, który jest efektem przetłumaczenia kodu asemblera, zawartego w takiej wstawce. Opiszę tu oba sposoby łączenia obydwu języków.

Łączenie na poziomie kodu maszynowego

W naszym przykładzie założymy, że w pliku f1.S zawarty będzie kod, napisany w asemblerze, a f2.c to kod z programem w języku C. Program w języku C będzie wykorzystywał jedną funkcję, napisaną w języku asemblera, która wyświetli prosty napis “Hello world”. Z powodu ograniczeń technicznych zakładamy, że program uruchomiony zostanie w środowisku POSIX na platformie i386 i skompilowany kompilatorem gcc. Używaną składnią asemblera będzie AT&T (domyślna dla asemblera GNU) Oto plik f1.S:

```
.text
.globl _f1
```

```
_f1:
  pushl %ebp
  movl %esp, %ebp
  movl $4, %eax /* 4 to funkcja systemowa "write" */
  movl $1, %ebx /* 1 to stdout */
  movl $tekst, %ecx /* adres naszego napisu */
  movl $len, %edx /* długość napisu w bajtach */
  int $0x80 /* wywołanie przerwania systemowego */
  popl %ebp
  ret

  .data
tekst:
  .string "Hello world\n"
  len = . - tekst
```

W systemach z rodziny UNIX należy pominąć znak "_" przed nazwą funkcji f1

Teraz kolej na f2.c:

```
extern void f1 (void); /* musimy użyć słowa extern */
int main ()
{
  f1();
  return 0;
}
```

Teraz możemy skompilować oba programy:

```
as f1.S -o f1.o
gcc f2.c -c -o f2.o
gcc f2.o f1.o -o program
```

W ten sposób uzyskujemy plik wykonywalny o nazwie "program". Efekt działania programu powinien być następujący:

```
Hello world
```

Na razie utworzyliśmy bardzo prostą funkcję, która w zasadzie nie komunikuje się z językiem C, czyli nie zwraca żadnej wartości ani nie pobiera argumentów. Jednak, aby zacząć pisać obsługę funkcji, która będzie pobierała argumenty i zwracała wyniki musimy poznać działanie języka C od trochę niższego poziomu.

Argumenty

Do komunikacji z funkcją język C korzysta ze stosu. Argumenty odkładane są w kolejności od ostatniego do pierwszego. Ponadto na końcu odkładany jest tzw. **adres powrotu**, dzięki czemu po wykonaniu funkcji program "wie", w którym miejscu ma kontynuować działanie. Ponadto, początek funkcji w asemblerze wygląda tak:

```
pushl %ebp
movl %esp, %ebp
```

Zatem na stosie znajdują się kolejno: zawartość rejestru EBP, adres powrotu a następnie argumenty od pierwszego do n-tego.

Zwracanie wartości

Na architekturze i386 do zwracania wyników pracy programu używa się rejestru EAX, bądź jego "mniejszych" odpowiedników, tj. AX i AH/AL. Zatem aby funkcja, napisana w asemblerze zwróciła "1" przed rozkazem ret należy napisać:

```
movl $1, %eax
```

Nazewnictwo

Kompilatory języka C/C++ dodają podkreślnik "_" na początku każdej nazwy. Dla przykładu funkcja:

```
void funkcja();
```

W pliku wyjściowym będzie posiadać nazwę _funkcja. Dlatego, aby korzystać z poziomu języka C z funkcji zakodowanych w asemblerze, muszą one mieć przy definicji w pliku asemblera wspomniany dodatkowy podkreślnik na początku.

Łączymy wszystko w całość

Pora, abyśmy napisali jakąś funkcję, która pobierze argumenty i zwróci jakiś konkretny wynik. Oto kod f1.S:

```
.text
.globl _funkcja
_funkcja:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%esp), %eax /* kopiujemy pierwszy argument do %eax */
  addl 12(%esp), %eax /* do pierwszego argumentu w %eax dodajemy drugi argument */
  popl %ebp
  ret /* ... i zwracamy wynik dodawania... */
```

oraz f2.c:

```
#include <stdio.h>
extern int funkcja (int a, int b);
int main ()
{
printf ("2+3=%d\n", funkcja(2,3));
return 0;
}
```

Po skompilowaniu i uruchomieniu programu powinniśmy otrzymać wydruk: 2+3=5

Wstawki asemblerowe

Oprócz możliwości wstępnie skompilowanych modułów możesz posłużyć się także tzw. **wstawkami asemblerowymi**. Ich użycie powoduje wstawienie w miejsce wystąpienia wstawki odpowiedniego kodu maszynowego, który powstanie po przetłumaczeniu kodu asemblerowego. Ponieważ jednak wstawki asemblerowe nie są standardowym elementem języka C, każdy kompilator ma całkowicie odmienną filozofię ich stosowania (lub nie ma ich w ogóle). Ponieważ w tym podręczniku używamy głównie kompilatora GNU, więc w tym rozdziale zostanie omówiona filozofia stosowania wstawek asemblera według programistów GNU.

Ze wstawek asemblerowych korzysta się tak:

```
int main ()
{
  asm ("nop");
}
```

W tym wypadku wstawiona zostanie instrukcja “nop” (no operation), która tak naprawdę służy tylko i wyłącznie do konstruowania pętli opóźniających.

C++

Język C++ z racji swojego podobieństwa do C będzie wyjątkowo łatwy do łączenia. Pewnym utrudnieniem może być obiektowość języka C++ oraz występowanie w nim przestrzeni nazw oraz możliwość przeciążania funkcji. Oczywiście nadal zakładamy, że główny program piszemy w C, natomiast korzystamy tylko z pojedynczych funkcji, napisanych w C++. Ponieważ język C nie oferuje tego wszystkiego, co daje programiście język C++, to musimy “zmusić” C++ do wyłączenia pewnych swoich możliwości, aby można było połączyć ze sobą elementy programu, napisane w dwóch różnych językach. Używa się do tego następującej konstrukcji:

```
extern "C" {
/* funkcje, zmienne i wszystko to, co będziemy łączyć z programem w C */
}
```

W zrozumieniu teorii pomoże Ci prosty przykład: plik f1.c:

```
#include <stdio.h>
extern int f2(int a);

int main ()
{
  printf ("%d\n", f2(2));
  return 0;
}
```

oraz plik f2.cpp:

```
#include <iostream>
using namespace std;
extern "C" {
  int f2 (int a)
  {
    cout << "a=" << a << endl;
    return a*2;
  }
}
```

Teraz oba pliki kompilujemy:

```
gcc f1.c -c -o f1.o
g++ f2.cpp -c -o f2.o
```

Przy łączeniu obu tych plików musimy pamiętać, że język C++ także korzysta ze swojej biblioteki. Zatem poprawna postać polecenia kompilacji powinna wyglądać:

```
gcc f1.o f2.o -o program -lstdc++
```

(stdc++ — biblioteka standardowa języka C++). Bardzo istotne jest tutaj to, abyśmy zawsze pamiętali o extern “C”, gdyż w przeciwnym razie funkcje napisane w C++ będą dla programu w C całkowicie niewidoczne.

Dodatek A

Indeks alfabetyczny

Alfabetyczny spis funkcji biblioteki standardowej ANSI C (tzw. libc) w wersji C89.

Dodatek B

Indeks tematyczny

Spis plików nagłówkowych oraz zawartych w nich funkcji i makr biblioteki standardowej C. Funkcje, makra i typy wprowadzone dopiero w standardzie C99 zostały oznaczone poprzez "[C99]" po nazwie.

assert.h

Makro asercji.

- assert()

ctype.h

Klasyfikowanie znaków.

- isalnum()
- isalpha()
- isblank() [C99]
- iscntrl()
- isdigit()
- isgraph()
- islower()
- isprint()
- ispunct()
- isspace()
- isupper()
- isxdigit()
- tolower()
- toupper()

errno.h

Deklaracje kodów błędów.

- EDOM (makro)
- EILSEQ (makro) [C99]
- ERANGE (makro)
- errno (zmienna)

float.h

Właściwości typów zmiennoprzecinkowych zależne od implementacji.

limits.h

Właściwości typów całkowitych zależne od implementacji.

locale.h

Ustawienia międzynarodowe.

- localeconv()
- setlocale()

math.h

Funkcje matematyczne.

- FP_FAST_FMAF (makro) [C99]
- FP_FAST_FMAL (makro) [C99]
- FP_FAST_FMA (makro) [C99]
- FP_ILOGB0 (makro) [C99]
- FP_ILOGBNAN (makro) [C99]
- FP_INFINITE (makro) [C99]
- FP_NAN (makro) [C99]
- FP_NORMAL (makro) [C99]
- FP_SUBNORMAL (makro) [C99]
- FP_ZERO (makro) [C99]
- HUGE_VALF (makro) [C99]
- HUGE_VALL (makro) [C99]
- HUGE_VAL (makro)
- INFINITY (makro) [C99]
- MATH_ERREXCEPT (makro) [C99]
- MATH_ERRNO (makro) [C99]
- NAN (makro) [C99]
- acosh()
- acos()
- asinh()
- asin()
- atan2()
- atanh()
- atan()
- cbrt() [C99]
- ceil()
- copysign() [C99]
- cosh()
- cos()
- double_t (typ) [C99]
- erfc() [C99]
- erf() [C99]
- exp2() [C99]
- expm1() [C99]
- exp()
- fabs()
- fdim() [C99]
- flaot_t (typ) [C99]
- floor()
- fmax() [C99]
- fma() [C99]
- fmin() [C99]
- fmod()
- fpclassify() [C99]
- frexp()
- hypot() [C99]
- ilogb() [C99]
- isfinite() [C99]
- isgreaterequal() [C99]
- isgreater() [C99]
- isinf() [C99]
- islessequal() [C99]
- islessgreater() [C99]
- isless() [C99]
- isnan() [C99]
- isnormal() [C99]
- isunordered() [C99]
- ldexp()
- lgamma() [C99]
- llrint() [C99]
- llround() [C99]
- log10()
- log1p() [C99]
- log2() [C99]
- logb() [C99]
- log()

- lrint() [C99]
- lround() [C99]
- math_errhandling (makro) [C99]
- modf()
- nan() [C99]
- nearbyint() [C99]
- nextafter() [C99]
- nexttoward() [C99]
- pow()
- remainder() [C99]
- remquo() [C99]
- rint() [C99]
- round() [C99]
- scalbln() [C99]
- scalbn() [C99]
- signbit() [C99]
- sinh()
- sin()
- sqrt()
- tanh()
- tan()
- tgamma() [C99]
- trunc() [C99]

setjmp.h

Obsługa nielokalnych skoków.

- longjmp()
- setjmp()

signal.h

Obsługa sygnałów.

- raise()
- signal()

stdarg.h

Narzędzia dla funkcji ze zmienną liczbą argumentów.

- va_arg()
- va_end()
- va_start()

stddef.h

Standardowe definicje.

- offsetof()

stdio.h

Standard Input/Output, czyli standardowe wejście-wyjście.

- clearerr()
- fclose()
- feof()
- ferror()
- fflush()
- fgetc()
- fgetpos()
- fgets()
- fopen()

stdlib.h

Najbardziej podstawowe funkcje.

string.h

Operacje na łańcuchach znaków

time.h

Funkcje obsługi czasu.

- asctime()
- clock()
- ctime()
- difftime()
- gmtime()
- localtime()
- mktime()
- strftime()
- time()
- tm (struktura)

Dodatek C

Wybrane funkcje biblioteki standardowej

assert

Deklaracja

```
#define assert(expr)
```

Plik nagłówkowy

assert.h

Opis

Makro przypominające w użyciu funkcję, służy do debuggowania programów. Gdy testowany warunek logiczny `expr` przyjmuje wartość fałsz, na standardowe wyjście błędów wypisywany jest komunikat o błędzie (zawierające m.in. argument wywołania makra; nazwę funkcji, w której zostało wywołane; nazwę pliku źródłowego oraz numer linii w formacie zależnym od implementacji) i program jest przerywany poprzez wywołanie funkcji abort.

W ten sposób możemy oznaczyć w programie niezmienniki, czyli warunki, które niezależnie od wartości zmiennych muszą pozostać prawdziwe. Jeśli asercja zawiedzie, oznacza to, że popełniliśmy błąd w algorytmie, piszemy sobie po pamięci (nadając zmiennym wartości, których nigdy nie powinny mieć) albo nastąpiła po drodze sytuacja wyjątkowa, na przykład związana z obsługą operacji wejścia-wyjścia.

Można łatwo pozbyć się asercji, uwalniając kod od spowalniających obciążeń a jednocześnie nie musząc kasować wystąpień assert i zachowując je na przyszłość. Aby to zrobić, należy przed dołączeniem pliku nagłówkowego assert.h zdefiniować makro NDEBUG, wówczas makro assert przyjmuje postać:

```
#define assert(ignore) ((void)0)
```

Makro assert jest redefiniowane za każdym dołączeniem pliku nagłówkowego assert.h.

Wartość zwracana

Makro nie zwraca żadnej wartości.

Przykład

```
#include <assert.h>

int main()
{
  int err=1;
  assert(err==0);
  return 0;
}
```

Program wypisze komunikat podobny do:

```
Assertion failed: err==0, file test.c, line 6
```

Natomiast jeśli uruchomimy:

```
#define NDEBUG
#include <assert.h>

int main()
{
  int err=1;
  assert(err==0);
  return 0;
}
```

nie pojawi się żaden komunikat o błędach.

atoi

Deklaracja

```
int  atoi (const char * string)
```

Plik nagłówkowy

stdlib.h

Opis

Funkcja jako argument pobiera liczbę w postaci ciągu znaków ASCII, a następnie zwraca jej wartość w formacie *int*. Liczbę może poprzedzać dowolona ilość białych znaków (spacje, tabulatory, itp.), oraz jej znak (*plus* (+) lub *minus* (-)). Funkcja atoi() kończy wczytywać znaki w momencie napotkania jakiegokowiek znaku który nie jest cyfrą.

Wartość zwracana

Przekształcona liczba, w przypadku gdy ciąg nie zawiera cyfr zwracana jest wartość 0.

Uwagi

Znak musi bezpośrednio poprzedzać liczbę, czyli możliwy jest zapis "-2", natomiast próba potraktowania funkcją atoi ciągu "- 2" skutkuje zwracaną wartością 0.

Przykład

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  char * c_Numer = "\n\t 2004u";
  int i_Numer;
  i_Numer = atoi(c_Numer);
  printf("\n Liczba typu int: %d, oraz jako ciąg znaków:  %s \n", i_Numer, c_Numer);
  return 0;
}
```

isalnum

Deklaracja

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int isblank(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispuntc(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

Argumenty

c wartość znaku reprezentowana w jako typ unsigned char lub wartość makra EOF. Z tego powodu, przed przekazaniem funkcji argumentu typu char lub signed char należy go zrzutować na typ unsigned char lub unsigned int.

Opis

Funkcje sprawdzają czy podany znak spełnia jakiś konkretny warunek. Biorą pod uwagę ustawienia języka i dla różnych znaków w różnych *locale*'ach mogą zwracać różne wartości.

isalnum sprawdza czy znak jest liczbą lub literą,

isalpha sprawdza czy znak jest literą,

isblank sprawdza czy znak jest znakiem odstępu służącym do oddzielania wyrazów (standardowymi znakami odstępu są spacja i znak tabulacji),

iscntrl sprawdza czy znak jest znakiem sterującym,

isdigit sprawdza czy znak jest cyfrą dziesiętna,

isgraph sprawdza czy znak jest znakiem drukowalnym różnym od spacji,

islower sprawdza czy znak jest małą literą,

isprint sprawdza czy znak jest znakiem drukowalnym (włączając w to spację),

ispunct sprawdza czy znak jest znakiem przestankowym, dla którego ani isspace ani isalnum nie są prawdziwe (standardowo są to wszystkie znaki drukowalne, dla których te funkcje zwracają zero),

isspace sprawdza czy znak jest tzw. białym znakiem (standardowymi białymi znakami są: spacja, wysunięcie strony '\f', znak przejścia do nowej linii '\n', znak powrotu karetki '\r', tabulacja pozioma '\t' i tabulacja pionowa '\v'),

isupper sprawdza czy znak jest dużą literą,

isxdigit sprawdza czy znak jest cyfrą szesnastkową, tj. cyfrą dziesiętną lub literą od 'a' do 'f' niezależnie od wielkości.

Funkcja isblank nie występowała w oryginalnym standardzie ANSI C z 1989 roku (tzw. C89) i została dodana dopiero w nowszym standardzie z 1999 roku (tzw. C99).

Wartość zwracana

Liczba niezerowa gdy podany argument spełnia konkretny warunek, w przeciwnym wypadku — zero.

Przykład użycia

```
#include <ctype.h>  /* funkcje is* */
#include <locale.h> /* setlocale */
#include <stdio.h>  /* printf i scanf */

void identify_char(int c) {
  printf("  Litera lub cyfra:   %s\n", isalnum (c) ? "tak" : "nie");
#if __STDC_VERSION__ >= 199901L
  printf("  Odstęp:             %s\n", isblank (c) ? "tak" : "nie");
#endif
  printf("  Znak sterujący:     %s\n", iscntrl (c) ? "tak" : "nie");
  printf("  Cyfra dziesiętna:   %s\n", isdigit (c) ? "tak" : "nie");
  printf("  Graficzny:          %s\n", isgraph (c) ? "tak" : "nie");
  printf("  Mała litera:        %s\n", islower (c) ? "tak" : "nie");
  printf("  Drukowalny:         %s\n", isprint (c) ? "tak" : "nie");
  printf("  Przestankowy:       %s\n", ispunct (c) ? "tak" : "nie");
  printf("  Biały znak:         %s\n", isspace (c) ? "tak" : "nie");
  printf("  Wielka litera:      %s\n", isupper (c) ? "tak" : "nie");
  printf("  Cyfra szesnastkowa: %s\n", isxdigit(c) ? "tak" : "nie");
}

int main() {
  unsigned char c;
  printf("Naciśnij jakiś klawisz.\n");
  if (scanf("%c", &c)==1) {
    identify_char(c);
    setlocale(LC_ALL, "pl_PL"); /* przystosowanie do warunków polskich */
    puts("Po zmianie ustawień języka:");
    identify_char(c);
  }
  return 0;
}
```

Zobacz też

- tolower, toupper

malloc

Deklaracja

```
#include <stdlib.h>

void *calloc(size_t nmeb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

Argumenty

nmeb liczba elementów, dla których ma być przydzielona pamięć

size rozmiar (w bajtach) pamięci do zarezerwowania bądź rozmiar pojedynczego elementu

ptr wskaźnik zwrócony przez poprzednie wywołanie jednej z funkcji lub NULL

Opis

Funkcja calloc przydziela pamięć dla `nmeb` elementów o rozmiarze `size` każdy i zeruje przydzieloną pamięć.

Funkcja malloc przydziela pamięć o wielkości `size` bajtów.

Funkcja free zwalnia blok pamięci wskazywany przez `ptr` wcześniej przydzielony przez jedną z funkcji malloc, calloc lub realloc. Jeżeli `ptr` ma wartość NULL funkcja nie robi nic.

Funkcja realloc zmienia rozmiar przydzielonego wcześniej bloku pamięci wskazywanego przez `ptr` do `size` bajtów. Pierwsze `n` bajtów bloku nie ulegnie zmianie gdzie `n` jest minimum z rozmiaru starego bloku i `size`. Jeżeli `ptr` jest równy zero (tj. NULL), funkcja zachowuje się tak samo jako malloc.

Wartość zwracana

Jeżeli przydzielanie pamięci się powiodło, funkcje calloc, malloc i realloc zwracają wskaźnik do nowo przydzielonego bloku pamięci. W przypadku funkcji realloc może to być wartość inna niż `ptr`.

Jeśli jako `size`, `nmeb` podano zero, zwracany jest albo wskaźnik NULL albo prawidłowy wskaźnik, który można podać do funkcji free (zauważmy, że NULL jest też prawidłowym argumentem free).

Jeśli działanie funkcji nie powiedzie się, zwracany jest NULL i odpowiedni kod błędu jest wpisywany do zmiennej errno. Dzieje się tak zazwyczaj, gdy nie ma wystarczająco dużo miejsca w pamięci.

Przykład

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

int main(void) {
  size_t size, num = 0;
  float *tab, tmp;
```

```
  /* Przydzielenie początkowego bloku pamięci */
  size = 64;
  tab = malloc(size * sizeof *tab);
  if (!tab) {
    perror("malloc");
    return EXIT_FAILURE;
  }

  /* Odczyt liczb */
  while (scanf("%f", &tmp)==1) {
    /* Jeżeli zapełniono całą tablicę, trzeba ją zwiększyć */
    if (num==size) {
      float *ptr = realloc(tab, (size *= 2) * sizeof *ptr);
      if (!ptr) {
        free(tab);
        perror("realloc");
        return EXIT_FAILURE;
      }
      tab = ptr;
    }
    tab[num++] = tmp;
  }

  /* Wypisanie w odwrotnej kolejnosci */
  while (num) {
    printf("%f\n", tab[--num]);
  }

  /* Zwolnienie pamieci i zakonczenie programu */
  free(tab);
  return EXIT_SUCCESS;
}
```

Uwagi

Użycie rzutowania przy wywołaniach funkcji malloc, realloc oraz calloc w języku C jest zbędne i szkodliwe. W przypadku braku deklaracji tych funkcji (np. gdy programista zapomni dodać plik nagłówkowy stdlib.h) kompilator przyjmuje domyślną deklaracje, w której funkcja zwraca int. Przy braku rzutowania spowoduje to błąd kompilacji (z powodu niemożności skonwertowania liczby na wskaźnik) co pozwoli na szybkie wychwycenie błędu w programie. Rzutowanie powoduje, że kompilator zostaje zmuszony do przeprowadzenia konwersji typów i nie wyświetla żadnych błędów. W przypadku języka C++ rzutowanie jest konieczne.

Zastosowanie operatora sizeof z wyrażeniem (np. `sizeof *tablica`), a nie typem (np. `sizeof float`) ułatwia późniejszą modyfikację programów. Gdyby w pewnym momencie programista zdecydował się zmienić tablicę z tablicy floatów na tablice double'i, musiałby wyszukiwać wszystkie wywołania funkcji malloc, realloc i calloc, co nie jest konieczne przy użyciu operatora sizeof z wyrażeniem.

Warto zauważyć, że w przypadku standardowych konfiguracji systemu GNU/Linux funkcje przydzielające pamięć nigdy nie zawodzą i nie zwracają wartości NULL (dla wartości parametru `size` większego od zera).

Ponieważ dla parametru `size` równego zero funkcja może zwrócić albo wskaźnik różny od wartości NULL albo jej równy, zwykłe sprawdzanie poprawności wywołania poprzez przyrównanie zwróconej wartości do zera może nie dać prawidłowego wyniku.

Zobacz też

Wskaźniki (dokładne omówienie zastosowania)

printf

Deklaracja

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...)

#include <stdarg.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

Opis

Funkcje formatują tekst zgodnie z podanym formatem opisanym poniżej. Funkcje printf i vprintf wypisują tekst na standardowe wyjście (tj. do stdout); fprintf i vfprintf do strumienia podanego jako argument; a sprintf, vsprintf, snprintf i vsnprintf zapisują go w podanej jako argument tablicy znaków.

Funkcje vprintf, vfprintf, vsprintf i vsnprintf różnią się od odpowiadających im funkcjom printf, fprintf, sprintf i snprintf tym, że zamiast zmiennej liczby argumentów przyjmują argument typu va_list.

Funkcje snprintf i vsnprintf różnią się od sprintf i vsprintf tym, że nie zapisuje do tablicy nie więcej niż `size` znaków (wliczając kończący znak '\0'). Oznacza to, że można je używać bez obawy o wystąpienie przepełnienia bufora.

Argumenty

format format, w jakim zostaną wypisane następne argumenty

stream strumień wyjściowy, do którego mają być zapisane dane

str tablica znaków, do której ma być zapisany sformatowany tekst

size rozmiar tablicy znaków

ap wskaźnik na pierwszy argument z listy zmiennej liczby argumentów

Format

Format składa się ze zwykłych znaków (innych niż znak '%'), które są kopiowane bez zmian na wyjście oraz sekwencji sterujących, zaczynających się od symbolu procenta, po którym następuje:

- dowolna liczba flag,
- opcjonalne określenie minimalnej szerokości pola,
- opcjonalne określenie precyzji,
- opcjonalne określenie rozmiaru argumentu,

- określenie formatu.

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

Flagi

W sekwencji możliwe są następujące flagi:

- **-** (minus) oznacza, że pole ma być wyrównane do lewej, a nie do prawej.
- + (plus) oznacza, że dane liczbowe zawsze poprzedzone są znakiem (plusem dla liczb nieujemnych lub minusem dla ujemnych).
- spacja oznacza, że liczby nieujemne poprzedzone są dodatkową spacją; jeżeli flaga plus i spacja są użyte jednocześnie to spacja jest ignorowana.
- # (*hash*) powoduje, że wynik jest przedstawiony w *alternatywnej postaci*:
  - dla formatu **o** powoduje to zwiększenie precyzji, jeżeli jest to konieczne, aby na początku wyniku było zero;
  - dla formatów **x** i **X** niezerowa liczba poprzedzona jest ciągiem **0x** lub **0X**;
  - dla formatów **a**, **A**, **e**, **E**, **f**, **F**, **g** i **G** wynik zawsze zawiera kropkę nawet jeżeli nie ma za nią żadnych cyfr;
  - dla formatów **g** i **G** końcowe zera nie są usuwane.
- **0** (zero) dla formatów **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g** i **G** do wyrównania pola wykorzystywane są zera zamiast spacji za wyjątkiem wypisywania wartości nieskończoność i NaN. Jeżeli obie flagi 0 i — są obecne to flaga zero jest ignorowana. Dla formatów **d**, **i**, **o**, **u**, **x** i **X** jeżeli określona jest precyzja flaga ta jest ignorowana.

Szerokość pola i precyzja

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

Precyzja dla formatów:

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

Szerokość pola może być albo dodatnią liczbą zaczynającą się od cyfry różnej od zera albo gwiazdką. Podobnie precyzja z tą różnicą, że jest jeszcze poprzedzona kropką. Gwiazdka oznacza, że brany jest kolejny z argumentów, który musi być typu int. Wartość ujemna przy określeniu szerokości jest traktowana tak jakby podano flagę **-** (minus).

Rozmiar argumentu

Dla formatów **d** i **i** można użyć jednego ze modyfikator rozmiaru:

- **hh** — oznacza, że format odnosi się do argumentu typu signed char,
- **h** — oznacza, że format odnosi się do argumentu typu short,
- **l** (el) — oznacza, że format odnosi się do argumentu typu long,
- **ll** (el el) — oznacza, że format odnosi się do argumentu typu long long,

- **j** — oznacza, że format odnosi się do argumentu typu intmax_t,
- **z** — oznacza, że że format odnosi się do argumentu typu będącego odpowiednikiem typu size_t ze znakiem,
- **t** — oznacza, że że format odnosi się do argumentu typu ptrdiff_t.

Dla formatów **o**, **u**, **x** i **X** można użyć takich samych modyfikatorów rozmiaru jak dla formatu **d** i oznaczają one, że format odnosi się do argumentu odpowiedniego typu bez znaku.

Dla formatu **n** można użyć takich samych modyfikatorów rozmiaru jak dla formatu **d** i oznaczają one, że format odnosi się do argumentu będącego wskaźnikiem na dany typ.

Dla formatów **a**, **A**, **e**, **E**, **f**, **F**, **g** i **G** można użyć modyfikatorów rozmiaru **L**, który oznacza, że format odnosi się do argumentu typu long double.

Dodatkowo, modyfikator **l** (el) dla formatu **c** oznacza, że odnosi się on do argumentu typu wint_t, a dla formatu **s**, że odnosi się on do argumenty typu wskaźnik na wchar_t.

Format

Funkcje z rodziny printf obsługują następujące formaty:

- **d**, **i** — argument typu int jest przedstawiany jako liczba całkowita ze znakiem w postaci **[-]ddd**.
- **o**, **u**, **x**, **X** — argument typu unsigned int jest przedstawiany jako nieujemna liczba całkowita zapisana w systemie oktalnym (**o**), dziesiętnym (**u**) lub heksadecymalnym (**x** i **X**).
- **f**, **F** — argument typu double jest przedstawiany w postaci **[-]ddd.ddd**.
- **e**, **E** — argument typu double jest reprezentowany w postaci **[i]d.ddde+dd**, gdzie liczba przed kropką dziesiętną jest różna od zera, jeżeli liczba jest różna od zera, a + oznacza znak wykładnika. Format **E** używa wielkiej litery E zamiast małej.
- **g**, **G** — argument typu double jest reprezentowany w formacie takim jak **f** lub **e** (odpowiednio **F** lub **E**) zależnie od liczby znaczących cyfr w liczbie oraz określonej precyzji.
- **a**, **A** — argument typu double przedstawiany jest w formacie **[-]0xh.hhhp+d** czyli analogicznie jak dla **e** i **E**, tyle że liczba zapisana jest w systemie heksadecymalnym.
- **c** — argument typu int jest konwertowany do unsigned char i wynikowy znak jest wypisywany. Jeżeli podano modyfikator rozmiaru **l** argument typu wint_t konwertowany jest do wielobajtowej sekwencji i wypisywany.
- **s** — argument powinien być typu wskaźnik na char (lub wchar_t). Wszystkie znaki z podanej tablicy, aż do i z wyłączeniem znaku null są wypisywane.
- **p** — argument powinien być typu wskaźnik na void. Jest to konwertowany na serię drukowalnych znaków w sposób zależny od implementacji.
- **n** — argument powinien być wskaźnikiem na liczbę całkowitą ze znakiem, do którego zapisana jest liczba zapisanych znaków.

W przypadku formatów **f**, **F**, **e**, **E**, **g**, **G**, **a** i **A** wartość nieskończoność jest przedstawiana w formacie **[-]inf** lub **[-]infinity** zależnie od implementacji. Wartość NaN jest przedstawiana w postaci **[-]nan** lub **[i]nan(*sekwencja*)**, gdzie **sekwencja** jest zależna od implementacji. W przypadku formatów określonych wielką literą również wynikowy ciąg znaków jest wypisywany wielką literą.

Wartość zwracana

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

Wyjątkami są funkcje snprintf i vsnprintf, które zwracają liczbę znaków, które zostałyby zapisane do tablicy znaków, gdyby była wystarczająco duża.

Przykład użycia

```
#include <stdio.h>
int main() {
  int i = 4;
  float f = 3.1415;
  char *s = "Monty Python";
  printf("i = %i\nf = %.1f\nWskaźnik s wskazuje na napis: %s\n", i, f, s);
  return 0;
}
```

Wyświetli:

```
i = 4
f = 3.1
Wskaźnik s wskazuje na napis: Monty Python
```

Funkcja formatująca ciąg znaków i alokująca odpowiednią ilość pamięci:

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

char *sprintfalloc(const char *format, ...) {
  int ret;
  size_t size = 100;
  char *str = malloc(size);
  if (!str) {
    return 0;
  }

  for(;;){
    va_list ap;
    char *tmp;

    va_start(ap, format);
    ret = vsnprintf(str, size, format, ap);
    va_end(ap);

    if (ret<size) {
      break;
    }

    tmp = realloc(str, (size_t)ret + 1);
    if (!tmp) {
      ret = -1;
      break;
    } else {
```

```
      str = tmp;
      size = (size_t)ret + 1;
    }
  }

  if (ret<0) {
    free(str);
    str = 0;
  } else if (size-1>ret) {
    char *tmp = realloc(str, (size_t)ret + 1);
    if (tmp) {
      str = tmp;
    }
  }

  return str;
}
```

Uwagi

Funkcje snprintf i vsnprintf nie były zdefiniowane w standardzie C89. Zostały one dodane dopiero w standardzie C99.

Biblioteka glibc do wersji 2.0.6 włącznie posiadała implementacje funkcji snprintf oraz vsnprintf, które były niezgodne ze standardem, gdyż zwracały -1 w przypadku, gdy wynikowy tekst nie mieścił się w podanej tablicy znaków.

scanf

Deklaracja

W pliku nagłówkowym stdio.h:

```
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

W pliku nagłówkowym stdarg.h:

```
int vscanf(const char *format, va_list ap);
int vsscanf(const char *str, const char *format, va_list ap);
int vfscanf(FILE *stream, const char *format, va_list ap);
```

Opis

Funkcje odczytują dane zgodnie z podanym formatem opisanym niżej. Funkcje scanf i vscanf odczytują dane ze standardowego wejścia (tj. stdin); fscanf i vfscanf ze strumienia podanego jako argument; a sscanf i vsscanf z podanego ciągu znaków.

Funkcje vscanf, vfscanf i vsscanf różnią się od odpowiadających im funkcjom scanf, fscanf i sscanf tym, że zamiast zmiennej liczby argumentów przyjmują argument typu va_list.

Argumenty

format format odczytu danych

stream strumień wejściowy, z którego mają być odczytane dane

str tablica znaków, z której mają być odczytane dane

ap wskaźnik na pierwszy argument z listy zmiennej liczby argumentów

Format

Format składa się ze zwykłych znaków (innych niż znak '%') oraz sekwencji sterujących, zaczynających się od symbolu procenta, po którym następuje:

- opcjonalna gwiazdka,
- opcjonalne maksymalna szerokość pola,
- opcjonalne określenie rozmiaru argumentu,
- określenie formatu.

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

Wystąpienie w formacie białego znaku powoduje, że funkcje z rodziny scanf będą odczytywać i odrzucać znaki, aż do napotkania pierwszego znaku nie będącego białym znakiem.

Wszystkie inne znaki (tj. nie białe znaki oraz nie sekwencje sterujące) muszą dokładnie pasować do danych wejściowych.

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

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

Maksymalna szerokość pola przyjmuje postać dodatniej liczby całkowitej zaczynającej się od cyfry różnej od zera. Określa ona ile maksymalnie znaków dany format może odczytać. Jest to szczególnie przydatne przy odczytywaniu ciągu znaków, gdyż dzięki temu można podać wielkość tablicy (minus jeden) i tym samym uniknąć błędów przepełnienia bufora.

Rozmiar argumentu

Dla formatów **d**, **i**, **o**, **u**, **x** i **n** można użyć jednego ze modyfikator rozmiaru:

- **hh** — oznacza, że format odnosi się do argumentu typu wskaźnik na signed char lub unsigned char,
- **h** — oznacza, że format odnosi się do argumentu typu wskaźnik na short lub wskaźnik na unsigned short,
- **l** (el) — oznacza, że format odnosi się do argumentu typu wskaźnik na long lub wskaźnik na unsigned long,
- **ll** (el el) — oznacza, że format odnosi się do argumentu typu wskaźnik na long long lub wskaźnik na unsigned long long,
- **j** — oznacza, że format odnosi się do argumentu typu wskaźnik na intmax_t lub wskaźnik na uintmax_t,
- **z** — oznacza, że że format odnosi się do argumentu typu wskaźnik na size_t lub odpowiedni typ ze znakiem,
- **t** — oznacza, że że format odnosi się do argumentu typu wskaźnik na ptrdiff_t lub odpowiedni typ bez znaku.

Dla formatów **a**, **e**, **f** i **g** można użyć modyfikatorów rozmiaru

- **l**, który oznacza, że format odnosi się do argumenty typu wskaźnik na double lub
- **L**, który oznacza, że format odnosi się do argumentu typu wskaźnik na long double.

Dla formatów **c**, **s** i [modyfikator **l** oznacza, że format odnosi się do argumentu typu wskaźnik na wchar_t.

Format

Funkcje z rodziny scanf obsługują następujące formaty:

- **d**, **i** odczytuje liczbę całkowitą, której format jest taki sam jak oczekiwany format przy wywołaniu funkcji strtol z argumentem base równym odpowiednio 10 dla **d** lub 0 dla **i**, argument powinien być wskaźnikiem na int;
- **o**, **u**, **x** odczytuje liczbę całkowitą, której format jest taki sam jak oczekiwany format przy wywołaniu funkcji strtoul z argumentem base równym odpowiednio 8 dla **o**, 10 dla **u** lub 16 dla **x**, argument powinien być wskaźnikiem na unsigned int;
- **a**, **e**, **f**, **g** odczytuje liczbę rzeczywistą, nieskończoność lub NaN, których format jest taki sam jak oczekiwany przy wywołaniu funkcji strtod, argument powinien być wskaźnikiem na flaot;
- **c** odczytuje dokładnie tyle znaków ile określono w maksymalnym rozmiarze pola (domyślnie 1), argument powinien być wskaźnikiem na char;
- **s** odczytuje sekwencje znaków nie będących białymi znakami, argument powinien być wskaźnikiem na char;
- [odczytuje niepusty ciąg znaków, z których każdy musi należeć do określonego zbioru, argument powinien być wskaźnikiem na char;
- **p** odczytuje sekwencje znaków zależną od implementacji odpowiadającą ciągowi wypisywanemu przez funkcję printf, gdy podano sekwencję `%p`, argument powinien być typu wskaźnik na wskaźnik na void;
- **n** nie odczytuje żadnych znaków, ale zamiast tego zapisuje do podanej zmiennej liczbę odczytanych do tej pory znaków, argument powinien być typu wskaźnik na int.

Słówko więcej o formacie [. Po otwierającym nawiasie następuje ciąg określający znaki jakie mogą występować w odczytanym napisie i kończy się on nawiasem zamykającym tj.]. Znaki pomiędzy nawiasami (tzw. *scanlist*) określają możliwe znaki, chyba że pierwszym znakiem jest ^ — wówczas w odczytanym ciągu znaków mogą występować znaki nie występujące w *scanlist*. Jeżeli sekwencja zaczyna się od [] lub [^] to ten pierwszy nawias zamykający nie jest traktowany jako koniec sekwencji tylko jak zwykły znak. Jeżeli wewnątrz sekwencji występuje znak **-** (minus), który nie jest pierwszym lub drugim jeżeli pierwszym jest ^ ani ostatnim znakiem zachowanie jest zależne od implementacji.

Formaty **A**, **E**, **F**, **G** i **X** są również dopuszczalne i mają takie same działanie jak **a**, **e**, **f**, **g** i **x**.

Wartość zwracana

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

Dodatek D

Składnia

Uwaga: przedstawione tutaj informacje nie są w stanie zastąpić treści całego podręcznika.

Symbole i słowa kluczowe

Język C definiuje pewną ilość słów, za pomocą których tworzy się np. pętle itp. Są to tzw. **słowa kluczowe**, tzn. nie można użyć ich jako nazwy zmiennej, czy też stałej (o nich poniżej). Oto lista słów kluczowych języka C (według norm ANSI C z roku 1989 oraz ISO C z roku 1990):

Tabela D.1: Symbole i słowa kluczowe

Słowo	**Opis w tym podręczniku**
auto	Zmienne
break	Instrukcje sterujące
case	Instrukcje sterujące
char	Zmienne
const	Zmienne
continue	Instrukcje sterujące
default	Instrukcje sterujące
do	Instrukcje sterujące
double	Zmienne
else	Instrukcje sterujące
enum	Typy złożone
extern	Biblioteki
float	Zmienne
for	Instrukcje sterujące
goto	Instrukcje sterujące
if	Instrukcje sterujące
int	Zmienne
long	Zmienne
register	Zmienne
return	Procedury i funkcje
short	Zmienne
signed	Zmienne
sizeof	Zmienne
static	Biblioteki, Zmienne
struct	Typy złożone
switch	Instrukcje sterujące
typedef	Typy złożone
union	Typy złożone
unsigned	Zmienne
void	Wskaźniki
volatile	Zmienne
while	Instrukcje sterujące

Specyfikacja ISO C z roku 1999 dodaje następujące słowa:

- **_Bool**
- **_Complex**
- **_Imaginary**
- **inline**
- **restrict**

Polskie znaki

Pisząc program, możemy stosować polskie litery (tj. "ąćęłńóśźż") tylko w:

- komentarzach
- ciągach znaków (łańcuchach)

Niedopuszczalne jest stosowanie polskich znaków w innych miejscach.

Operatory

Operatory arytmetyczne

Są to operatory wykonujące znane wszystkim dodawanie,odejmowanie itp.:

operator	**znaczenie**
+	dodawanie
-	odejmowanie
*	mnożenie
/	dzielenie
%	dzielenie modulo — daje w wyniku samą resztę z dzielenia
=	operator przypisania — wykonuje działanie po prawej stronie i wynik przypisuje obiektowi po lewej

Operatory logiczne

Służą porównaniu zawartości dwóch zmiennych według określonych kryteriów:

Operator	**Rodzaj porównania**
==	czy równe
>	większy
>=	większy bądź równy
<	mniejszy
<=	mniejszy bądź równy
!=	czy różny(nierówny)

Są jeszcze operatory, służące do grupowania porównań (Patrz też:logika w Wikipedi):

`||`	lub(OR)
`&&`	i,oraz(AND)
`!`	negacja(NOT)

Operatory binarne

Są to operatory, które działają na bitach.

operator	funkcja	przykład
|	suma bitowa(OR)	*5 — 2* da w wyniku *7* (*00000101* **OR** *00000010* = *00000111*)
&	iloczyn bitowy	*7 & 2* da w wyniku *2* (*00000111* **AND** *00000010* = *00000010*)
~	negacja bitowa	*2* da w wyniku *253* (**NOT** *00000010* = *11111101*)
>>	przesunięcie bitów o X w prawo	*7 » 2* da w wyniku *1* (*00000111* >> 2 = *00000001*)
<<	przesunięcie bitów o X w lewo	*7 « 2* da w wyniku *28* (*00000111* << 2 = *00011100*)
^	alternatywa wyłączna	*7 ^ 2* da w wyniku *5* (*00000111* ^ *00000010* = *00000101*)

Operatory inkrementacji/dekrementacji

Służą do dodawania/odejmowania od liczby wartości jeden.

Przykłady:

Operacja	Opis operacji	Wartość wyrażenia
x++	zwiększy wartość w x o jeden	wartość zmiennej x przed zmianą
++x	zwiększy wartość w x o jeden	wartość zmiennej x powiększona o jeden
x–	zmniejszy wartość w x o jeden	wartość zmiennej x przed zmianą
–x	zmniejszy wartość w x o jeden	wartość zmiennej x pomniejszona o jeden

Parę przykładów dla zrozumienia:

```
int a=7;
if ((a++)==7) /* najpierw porównuje, potem dodaje */
  printf ("%d\n",a); /* wypisze 8 */
if ((++a)==9) /* najpierw dodaje, potem porównuje */
  printf ("%d\n", a); /* wypisze 9 */
```

Analogicznie ma się sytuacja z operatorami dekrementacji.

Pozostałe

Operacja	Opis operacji	Wartość wyrażenia
*x	operator wyłuskania dla wskaźnika	wartość trzymana w pamięci pod adresem przechowywanym we wskaźniku
&x	operator pobrania adresu	zwraca adres zmiennej
x[a]	operator wybrania elementu tablicy	zwraca element tablicy o indeksie a (numerowanym od zera)
x.a	operator wyboru składnika a ze zmiennej x	wybiera składnik ze struktury lub unii
x->a	operator wyboru składnika a przez wskaźnik do zmiennej x	wybiera składnik ze struktury, gdy używamy wskaźnika do struktury zamiast zwykłej zmiennej
sizeof(typ)	operator pobrania rozmiaru typu	zwraca rozmiar typu w bajtach

Operator ternarny

Istnieje jeden operator przyjmujący trzy argumenty — jest to operator wyrażenia warunkowego: `a ? b : c`. Zwraca on **b** gdy **a** jest prawdą lub **c** w przeciwnym wypadku.

Typy danych

Tabela D.2: Typy danych według różnych specyfikacji języka C

Typ	Opis	Inne nazwy
Typy danych wg norm C89 i C90		
char	Służy głównie do przechowywania znaków. Od kompilatora zależy, czy jest to liczba ze znakiem czy bez; w większości kompilatorów jest liczbą ze znakiem	
signed char	Typ char ze znakiem	
unsigned char	Typ char bez znaku	
short	Występuje, gdy docelowa maszyna wyszczególnia krótki typ danych całkowitych, w przeciwnym wypadku jest tożsamy z typem **int**. Często ma rozmiar jednego słowa maszynowego	**short int**, **signed short**, **signed short int**
unsigned short	Liczba typu **short** bez znaku Podobnie jak **short** używana do zredukowania zużycia pamięci przez program	**unsigned short int**
int	Liczba całkowita, odpowiadająca podstawowemu rozmiarowi liczby całkowitej w danym komputerze. Podstawowy typ dla liczb całkowitych	**signed int**, **signed**
unsigned	Liczba całkowita bez znaku	**unsigned int**
long	* Długa liczba całkowita	
unsigned long	Długa liczba całkowita	**long int**, **signed long**, **signed long int**
float	Podstawowy typ do przechowywania liczb zmiennoprzecinkowych. W nowszym standardzie zgodny jest z normą IEEE 754. Nie można stosować go z modyfikatorem **signed** ani **unsigned**	

double	Liczba zmiennoprzecinkowa podwójnej precyzji. Podobnie jak float nie łączy się z modyfikatorem **signed** ani **unsigned**	
long double	Największa możliwa dokładność liczb zmiennoprzecinkowych. Nie łączy się z modyfikatorem **signed** ani **unsigned**.	
Typy danych według normy C99		
_Bool	Przechowuje wartości 0 lub 1	
long long	Nowy typ, umożliwiający obliczeniach na bardzo dużych liczbach całkowitych bez użycia typu float	**long long int**, **signed long long**, **signed long long int**
unsigned long long	Długie liczby całkowite bez znaku	**unsigned long long int**
float _Complex	Służy do przechowywania liczb zespolonych	
double _Complex	Służy do przechowywania liczb zespolonych	
long double _Complex	Służy do przechowywania liczb zespolonych	
Typy danych definiowane przez użytkownika		
struct	Więcej o kompilowaniu.	
union	Rozmiar typu jest taki jak rozmiar największego pola	
typedef	Nowo zdefiniowany typ przyjmuje taki sam rozmiar, jak typ macierzysty	
enum	Zwykle elementy mają taką samą długość, jak typ **int**.	

Zależności rozmiaru typów danych są następujące:

- sizeof(*cokolwiek*) = sizeof(**signed** *cokolwiek*) = sizeof(**unsigned** *cokolwiek*);
- 1 = sizeof(**char**) $\leqslant$ sizeof(**short**) $\leqslant$ sizeof(**int**) $\leqslant$ sizeof(**long**) $\leqslant$ sizeof(**long long**);
- sizeof(**float**) $\leqslant$ sizeof(**double**) $\leqslant$ sizeof(**long double**);
- sizeof(*cokolwiek* **_Complex**) = 2 * sizeof(*cokolwiek*)
- sizeof(**void ***) = sizeof(**char ***) $\geqslant$ sizeof(*cokolwiek* *);
- sizeof(*cokolwiek* *) = sizeof(**signed** *cokolwiek* *) = sizeof(**unsigned** *cokolwiek* *);
- sizeof(*cokolwiek* *) = sizeof(**const** *cokolwiek* *).

Dodatkowo, jeżeli przez V(typ) oznaczymy liczbę bitów wykorzystywanych w typie to zachodzi:

- $8 \leqslant$ V(**char**) = V(**signed char**) = V(**unsigned char**);
- $16 \leqslant$ V(**short**) = V(**unsigned short**);
- $16 \leqslant$ V(**int**) = V(**unsigned int**);
- $32 \leqslant$ V(**long**) = V(**unsigned long**);
- $64 \leqslant$ V(**long long**) = V(**unsigned long long**);
- V(**char**) $\leqslant$ V(**short**) $\leqslant$ V(**int**) $\leqslant$ V(**long**) $\leqslant$ V(**long long**).

Dodatek E

Przykłady z komentarzem

Liczby losowe

Poniższy program generuje wiersz po wierszu macierz o określonych przez użytkownika wymiarach, zawierającą losowo wybrane liczby. Każdy wygenerowany wiersz macierzy zapisywany jest w pliku tekstowym o wprowadzonej przez użytkownika nazwie. W pierwszym wierszu pliku wynikowego zapisano wymiary utworzonej macierzy. Program napisany i skompilowany został w środowisku GNU/Linux.

```
#include <stdio.h>
#include <stdlib.h>  /* dla funkcji rand() oraz srand() */
#include <time.h>    /* dla funkcji [time() */

main()
{
  int i, j, n, m;
  float re;
  FILE *fp;
  char fileName[128];

  printf("Wprowadz nazwe pliku wynikowego..\n");
  scanf("%s",&fileName);

  printf("Wprowadz po sobie liczbe wierszy i kolumn macierzy oddzielone spacją..\n");
  scanf("%d %d", &n, &m);

          /* jeżeli wystąpił błąd w otwieraniu pliku i go nie otwarto,
           wówczas funkcja fclose(fp) wywołana na końcu programu zgłosi błąd
           wykonania i wysypie nam program z działania, stąd musimy umieścić
           warunek, który w kontrolowany sposób zatrzyma program (funkcja exit;)
          */
  if ( (fp = fopen(fileName, "w")) == NULL )
  {
    puts("Otwarcie pliku nie jest mozliwe!");
    exit;    /*  jeśli w procedurze glownej
              to piszemy bez nawiasow */
  }

  else  { puts("Plik otwarty prawidłowo..");  }
```

```
  fprintf(fp, "%d %d\n", n, m);
          /* w pierwszym wierszu umieszczono wymiary macierzy */

  srand( (unsigned int) time(0) );
  for (i=1; i<=n; ++i)
  {
    for (j=1; j<=m; ++j)
    {
      re = ((rand() % 200)-100)/ 10.0;
      fprintf(fp,"%.1f", re );
      if (j!=m) fprintf(fp,"    ");
    }
  fprintf(fp,"\n");
  }
  fclose(fp);
  return 0;
}
```

Zamiana liczb dziesiętnych na liczby w systemie dwójkowym

Zajmijmy się teraz innym zagadnieniem. Wiemy, że komputer zapisuje wszystkie liczby w postaci binarnej (czyli za pomocą jedynek i zer). Spróbujmy zatem zamienić liczbę, zapisaną w "naszym" dziesiątkowym systemie na zapis binarny. **Uwaga:** Program działa jedynie dla liczb od 0 do maksymalnej wartości którą może przyjąć typ `unsigned short int` w twoim kompilatorze.

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

void dectobin (unsigned short a)
{
  int licznik;

  /* CHAR_BIT to liczba bitów w bajcie */
  licznik = CHAR_BIT * sizeof(a);
  while (--licznik >= 0) {
    putchar(((a >> licznik) & 1)) ? '1' : '0');
  }
}

int main ()
{
  unsigned short a;

  printf ("Podaj liczbę od 0 do %hd: ", USHRT_MAX);
  scanf ("%hd", &a);
  printf ("%hd(10) = ", a);
  dectobin(a);
  printf ("(2)\n");

  return 0;
}
```

Zalążek przeglądarki

Zajmiemy się tym razem inną kwestią, a mianowicie programowaniem sieci. Jest to zagadnienie bardzo ostatnio popularne. Nasz program będzie miał za zadanie połączyć się z serwerem, którego adres użytkownik będzie podawał jako pierwszy parametr programu, wysłać zapytanie HTTP i odebrać treść, którą wyśle do nas serwer. Zacznijmy może od tego, że obsługa sieci jest niemal identyczna w różnych systemach operacyjnych. Na przykład między systemami z rodziny Unix oraz Windowsem różnica polega tylko na dołączeniu innych plików nagłówkowych (dla Windowsa — winsock2.h). Przeanalizujmy zatem poniższy kod:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define MAXRCVLEN 512
#define PORTNUM 80

char *query = "GET / HTTP1.1\n\n";

int main(int argc, char *argv[])
{
  char buffer[MAXRCVLEN+1];
  int len, mysocket;
  struct sockaddr_in dest;
  char *host_ip = NULL;
  if (argc != 2) {
    printf ("Podaj adres serwera!\n");
    exit (1);
    }
  host_ip = argv[1];
  mysocket = socket(AF_INET, SOCK_STREAM, 0);

  dest.sin_family = AF_INET;
  dest.sin_addr.s_addr = inet_addr(host_ip); /* ustawiamy adres hosta */
  dest.sin_port = htons (PORTNUM); /* numer portu przechowuje dwubajtowa zmienna -
           musimy ustalić porządek sieciowy - Big Endian */
  memset(&(dest.sin_zero), '\0', 8); /* zerowanie reszty struktury */

  connect(mysocket, (struct sockaddr *)&dest,sizeof(struct sockaddr));
  /* łączymy się z hostem */
  write (mysocket, query, strlen(query)); /* wysyłamy zapytanie */
  len=read(mysocket, buffer, MAXRCVLEN); /* i pobieramy odpowiedź */

  buffer[len]='\0';

  printf("Rcvd: %s",buffer);
  close(mysocket); /* zamykamy gniazdo */
  return EXIT_SUCCESS;
}
```

Powyższy przykład może być odrobinę niezrozumiały, dlatego przyda się kilka słów wyjaśnienia. Pliki nagłówkowe, które dołączamy zawierają deklarację nowych dla Ciebie funkcji — socket(), connect(), write() oraz read(). Oprócz tego spotkałeś się z nową strukturą — sockaddr_in. Wszystkie te obiekty są niezbędne do stworzenia połączenia.

Dodatek F

Informacje o pliku i historia

Historia

Ta książka została stworzona na polskojęzycznej wersji projektu Wikibooks przez autorów wymienionych poniżej w sekcji Autorzy. Najnowsza wersja podręcznika jest dostępna pod adresem `http://pl.wikibooks.org/wiki/C`.

Informacje o pliku PDF i historia

PDF został utworzony przez Derbetha dnia 17 lutego 2008 na podstawie wersji z 17 lutego 2008 podręcznika na Wikibooks. Wykorzystany został poprawiony program Wiki2LaTeX autorstwa użytkownika angielskich Wikibooks, Hagindaza. Wynikowy kod po ręcznych poprawkach został przekształcony w książkę za pomocą systemu składu LaTeX.

Najnowsza wersja tego PDF-u jest postępna pod adresem `http://pl.wikibooks.org/wiki/Image:C.pdf`.

Autorzy

Adam majewski, Akira, Arfrever, Bercik, Bla, Bociex, Cnr, CzarnyZajaczek, Darek86, Derbeth, Equadus, Evil, Faw, GDR!, Gk180984, Incuś, Karol Ossowski, Kj, Lewico, MTM, Michael, Migol, Mina86, Mtfk, Myki, Mythov, Noisy, Pawelkg, Pawlosck, Peter de Sowaro, Piotr, Piotrala, Ponton, Sasek, Sblive, Silbarad, Stefan, T ziel, Warszk, Webprog i anonimowi autorzy.

Grafiki

Autorzy i licencje grafik:

- grafika na okładce: Saint-Elme Gautier, rycina z książki *Le Corset à travers les âges*, Paryż 1893; źródło Wikimedia Commons; public domain
- logo Wikibooks: zastrzeżony znak towarowy, © & ™All rights reserved, Wikimedia Foundation, Inc.
- grafika 14.1a (strona 92): autor Claudio Rocchini, źródło Wikimedia Commons, licencja GFDL
- grafika 14.1b (strona 92): autor Adam majewski, źródło Wikimedia Commons, licencja Creative Commons Attribution 3.0 Unported

- grafia 16.1 (strona 95): autor Jarkko Piiroinen, źródo WikimediaCommons, public domain
- grafika 16.2 (strona 98): autor Jarkko Piiroinen, źródo WikimediaCommons, public domain
- grafika 17 (strona 101): autor Daniel B, źródo Wikimedia Commons, licencja GFDL
- grafika 17.2 (strona 105): autor Daniel B, źródło Wikimedia Commons, licencja GFDL
- grafika 17.3 (strona 105): autor Daniel B, źródło Wikimedia Commons, licencja GFDL
- grafika 17.4 (strona 112): autor Derrick Coetzee, źródło Wikimedia Commons, public domain
- grafika 18.1 (strona 120): autor Jarkko Piiroinen, źródo WikimediaCommons, public domain

Dodatek G

Dalsze wykorzystanie tej książki

Wstęp

Ideą Wikibooks jest swobodne dzielenie się wiedzą, dlatego też uregulowania Wikibooks mają na celu jak najmniejsze ograniczanie możliwości osób korzystających z serwisu i zapewnienie, że treści dostępne tutaj będą mogły być łatwo kopiowane i wykorzystywane dalej. Prosimy wszystkich odwiedzających, aby poświęcili chwilę na zaznajomienie się z poniższymi zasadami, by uniknąć w przyszłości nieporozumień.

Status prawny

Cała zawartość Wikibooks (o ile w danym miejscu nie jest zaznaczone inaczej; szczegóły niżej) jest udostępniona na następujących warunkach:

Udziela się zezwolenia na kopiowanie, rozpowszechnianie i/lub modyfikację treści artykułów polskich Wikibooks zgodnie z zasadami Licencji GNU Wolnej Dokumentacji (GNU Free Documentation License) w wersji 1.2 lub dowolnej późniejszej opublikowanej przez Free Software Foundation; bez Sekcji Niezmiennych, Tekstu na Przedniej Okładce i bez Tekstu na Tylnej Okładce. Kopia tekstu licencji znajduje się na stronie GNU Free Documentation License.

Zasadniczo oznacza to, że artykuły pozostaną na zawsze dostępne na zasadach open source i mogą być używane przez każdego z obwarowaniami wyszczególnionymi poniżej, które zapewniają, że artykuły te pozostaną wolne.

Grafiki i pliku multimedialne wykorzystane na Wikibooks mogą być udostępnione na warunkach innych niż GNU FDL. Aby sprawdzić warunki korzystania z grafiki, należy przejść do jej strony opisu, klikając na grafice.

Wykorzystywanie materiałów z Wikibooks

Jeśli użytkownik polskich Wikibooks chce wykorzystać materiały w niej zawarte, musi to zrobić zgodnie z GNU FDL. Warunki te to w skrócie i uproszczeniu:

Publikacja w Internecie

1. dobrze jest wymienić Wikibooks jako źródło (jest to nieobowiązkowe, lecz jest dobrym zwyczajem)

2. należy podać listę autorów lub wyraźnie opisany i funkcjonujący link do oryginalnej treści na Wikibooks lub historii edycji (wypełnia to obowiązek podania autorów oryginalnego dzieła)
3. trzeba jasno napisać, że treść publikowanego dzieła jest objęta licencją GNU FDL
4. należy podać link to tekstu licencji (najlepiej zachowanej na własnym serwerze)
5. postać tekstu nie może ograniczać możliwości jego kopiowania

Druk

1. dobrze jest wymienić Wikibooks jako źródło (jest to nieobowiązkowe, lecz jest dobrym zwyczajem)
2. należy wymienić 5 autorów z listy w rozdziale *Autorzy* danego podręcznika (w przypadku braku podobnego rozdziału — lista autorów jest dostępna pod odnośnikiem "historia" na górze strony). Gdy podręcznik ma mniej niż 5 autorów, należy wymienić wszystkich.
3. trzeba jasno napisać na przynajmniej jednej stronie, że treść publikowanego dzieła jest objęta licencją GNU FDL
4. *pełny* tekst licencji, w oryginale i bez zmian, musi być zawarty w książce
5. jeśli zostanie wykonanych więcej niz 100 kopii książki konieczne jest:
   (a) dostarczenie płyt CD, DVD, dysków lub innych nośników danych z treścia książki w formie możliwą do komputerowego przetwarzania; lub:
   (b) dostarczenie linku do strony z czytelną dla komputera formą książki (link musi być aktywny przynajmniej przez rok od publikacji; można użyć linku do spisu treści danego podręcznika na Wikibooks)

Nie ma wymogu pytania o zgodę na wykorzystanie tekstu jakichkolwiek osób z Wikibooks. Autorzy nie mogą zabronić nikomu wykorzystywania ich tekstów zgodnie z licencją GNU FDL. Można korzystać z książek jako całości albo z ich fragmentów. Materiały bazujące na treści z Wikibooks mogą być bez przeszkód sprzedawane; zyskami nie trzeba dzielić się z autorami oryginalnego dzieła.

Jeżeli w wykorzystanych materiałach z polskich Wikibooks są treści objęte innymi niż GNU FDL licencjami, użytkownik musi spełnić wymagania zawarte w tych licencjach (dotyczy to szczególnie grafik, które mogą mieć różne licencje).

Dodatek H

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying

this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Skorowidz