Programowanie w języku

C

Dla początkujących oraz średnio zaawansowanych programistów

wersja: 1.0 (27.10.2010)

Spis treści

1 Wprowadzenie....5
1.1 Informacje od autora....5
1.2 Jak napisana jest ta książka?....5
1.3 Dla kogo jest ta książka?....6
2 Podstawy języka C....6
2.1 Pierwszy program....6
2.1.1 Struktura oraz opis kodu źródłowego języka C....7
2.1.2 Komentarze....8
2.2 Zmienne i stałe....9
2.2.1 Typy zmiennych....9
2.2.2 Zakres typów zmiennych....10
2.2.3 Nazwy zmiennych i deklaracja zmiennych....11
2.2.4 Stałe....12
2.2.5 Wyrażenia stałe i stałe symboliczne....13
2.2.6 Stała wyliczenia....14
2.2.7 Zasięg zmiennych....15
2.3 Matematyka....17
2.3.1 Operatory arytmetyczne....17
2.3.2 Operatory logiczne i relacje....19
2.3.3 Operatory zwiększania, zmniejszania oraz przypisywania....21
2.3.4 Operatory bitowe....23
2.3.5 Priorytety....35
2.3.6 Funkcje matematyczne....36
3 Sterowanie programem....40
3.1 Instrukcja if – else....40
3.2 Instrukcja switch....44
3.3 Pętle....47
3.3.1 for....47
3.3.2 while....52
3.3.3 do – while....53
3.4 Instrukcja break....54
3.5 Instrukcja continue....55
3.6 Instrukcja goto, etykiety....56
4 Funkcje....57
4.1 Ogólna postać funkcji oraz funkcja zwracająca wartości całkowite....57
4.2 Funkcje zwracające wartości rzeczywiste....59
4.3 Funkcje nie zwracające wartości oraz brak argumentów....62
4.4 Pliki nagłówkowe....63
4.4.1 Kompilacja warunkowa....66
4.5 extern, static, register....68
4.5.1 extern....68
4.5.2 static....71
4.5.3 register....75
4.6 Funkcje rekurencyjne....77

5 Tablice i wskaźniki....78
5.1 Tablice....78
5.1.1 Tablice jednowymiarowe....78
5.1.2 Tablice wielowymiarowe....79
5.2 Wskaźniki....82
5.3 Przekazywanie adresu do funkcji....84
5.4 Zależności między tablicami, a wskaźnikami....85
5.5 Operacje na wskaźnikach....89
5.6 Wskaźnik typu void....91
5.7 Tablice znakowe....92
5.8 Przekazywanie tablicy do funkcji....95
5.9 Wskaźniki do wskaźników....97
5.10 Tablica wskaźników....98
6 Argumenty funkcji main....104
7 Struktury....110
7.1 Podstawowe informacje o strukturach....110
7.2 Operacje na elementach struktury....113
7.3 Przekazywanie struktur do funkcji....113
7.4 Zagnieżdżone struktury....117
7.5 Tablice struktur....119
7.6 Słowo kluczowe typedef....121
7.7 Unie....122
7.8 Pola bitowe....125
8 Operacje wejścia i wyjścia....130
8.1 Funkcja getchar i putchar....130
8.2 Funkcja printf i sprintf....131
8.3 Funkcja scanf i sscanf....135
8.4 Zmienna ilość argumentów....139
8.5 Obsługa plików....141
8.6 Pobieranie i wyświetlanie całych wierszy tekstów – funkcje: fgets, fputs....146
9 Dynamicznie przydzielana pamięć....150
10 Biblioteka standardowa....153
10.1 assert.h....153
10.2 complex.h....155
10.3 ctype.h....158
10.4 errno.h....160
10.5 iso646.h....161
10.6 limits.h....162
10.7 locale.h....164
10.8 math.h....167
10.9 setjmp.h....168
10.10 signal.h....170
10.11 stdarg.h....173
10.12 stdbool.h....173
10.13 stdio.h....174
10.13.1 Operacje na plikach....176
10.13.2 Formatowane wyjście....186

10.13.3 Formatowane wejście......189
10.13.4 Wejście i wyjście znakowe......191
10.13.5 Pozycja w pliku......194
10.13.6 Obsługa błędów......198
10.14 stdlib.h......202
10.14.1 Konwersja ciągu znaków na liczby......203
10.14.2 Pseudo-losowe liczby 209
10.14.3 Dynamicznie przydzielana pamięć......210
10.14.4 Funkcje oddziaływujące ze środowiskiem uruchomienia......212
10.14.5 Wyszukiwanie i sortowanie......218
10.14.6 Arytmetyka liczb całkowitych......222
10.15 string.h......223
10.15.1 Kopiowanie......224
10.15.2 Dołączanie......229
10.15.3 Porównywanie......230
10.15.4 Wyszukiwanie......235
10.15.5 Inne......242
10.16 time.h......244
10.16.1 Manipulacja czasem......245
10.16.2 Konwersje......250
10.16.3 Makra......254
10.16.4 Typy danych......255
11 MySQL – Integracja programu z bazą danych......257
Dodatek A......260
A.1 Zmiana katalogu......261
A.2 Tworzenie katalogu......261
A.3 Usuwanie plików i katalogów......262
A.4 Wyświetlanie zawartości katalogu......262
A.5 Kompilacja programów......263
A.6 Ustawianie uprawnień......264
A.7 Ustawianie właściciela......264
Dodatek B......265
B.1 Powłoka systemowa......265
B.2 Polecenie time, formatowanie wyników......265
Dodatek C......268
C.1 Instalacja MySQL......268
C.2 Podstawowe polecenia MySQL......269

1 Wprowadzenie

1.1 Informacje od autora

Witajcie. Informacje zawarte w tej książce nie stanowią kompletnego kompendium wiedzy z zakresu języka C, natomiast podstawowe oraz średnio zaawansowane operacje jakie można wykonywać z użyciem tego języka. Książka ta została napisana całkowicie przypadkiem, zaczęło się to bardzo niewinnie od pisania małego poradnika, który wraz z upływem wakacji rozrastał się, by wreszcie osiągnąć obecną postać. Książka w gruncie rzeczy składa się z bardzu wielu, bo aż z ponad 180 przykładów. Wszystkie przykłady zostały skompilowane z użyciem kompilatora gcc w wersji 4.4.1 na Linuksie. Starałem się tłumaczyć wszystkie zagadanienia najlepiej jak tylko potrafiłem, żeby zrozumiały to osoby nie mające bladego pojęcia na temat programowania, przez co bardziej doświadczeni programiści mogą odczuć lekki dyskomfort. Jak mi wyszło? Mam nadzieje, że ocenisz sam. W wielu programach pokazany został jedynie sposób użycia pewnych mechanizmów. W programach z prawdziwego zdarzenia wykorzystanie ich wiązałoby się z konkretnym zadaniem. Jeśli zauważysz jakiekolwiek błędy możesz wysłać mi informacje wraz z opisem, gdzie wdarł się błąd na adres `apyszczuk@gmail.com`. Książka ta udostępniana jest na licencji Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported lub nowszej. Życzę miłej lektury.

Artur Pyszczuk

1.2 Jak napisana jest ta książka?

Opis parametru	Przykład
Kody źródłowe przedstawione zostały z ramce z tłem	`#include`
Polecenia systemowe w Linuksie wyróżnione zostały w ramce z tłem	`$ cat plik.c`
Nazwy funkcji i plików nagłówkowych zostały zapisane taką czcionką	main
Nazwy typów zmiennych, deklaracje zmiennych, słowa kluczowe	int
Makra oraz pliki z kodem zostały wytłuszczone	**NULL, main.c**
Prototypy funkcji zostały zapisane taką czcionką	`int printf (...)`

Tabela 1.2.1 Informacje nawigacyjne

1.3 Dla kogo jest ta książka?

Materiał ten jest dla wszystkich tych, którzy chcą nauczyć się programować w języku C nie znając go w ogóle, bądź mają jakieś pojęcie, lecz nie wiedzą „co z czym się je”. Materiał ten może być pomocny również dla ludzi, którzy mieli już styczność z programowaniem w C, natomiast długo nie programowali i pragną odświeżyć swoją wiedzę.

2 Podstawy języka C

2.1 Pierwszy program

Pierwszy program, który napiszesz w języku C będzie miał za zadanie wyświetlenie tekstu "Hello World!". Nie tylko w C, lecz w innych językach programowania również stosuje się tego typu praktykę, by pokazać w jaki sposób informacje wyświetlane są na ekranie monitora. Tak więc w dowolnym edytorze tekstu wpisz poniższy kod (listing 2.1.1) oraz zapisz go pod nazwą **helloWorld.c** (np. w katalogu domowym).

```
#include <stdio.h>

main ()
{
     printf("Hello World!\n");
}
```

Listing 2.1.1 Pierwszy program – Hello World.

Aby skompilować nowo utworzony kod należy w konsoli systemowej przejść do katalogu, w którym znajduje się plik **helloWorld.c** oraz wydać polecenie kompilacji kompilatora gcc. Podstawowe polecenia systemu Linux znajdują się w dodatku A.

```
$ gcc helloWorld.c -o helloWorld
```

Jeśli nie zrobiłeś żadnej literówki, kompilator nie powinien wyświetlić żadnego błędu tudzież ostrzeżenia, co za tym idzie kod źródłowy powinien zostać skompilowany, a więc plik wykonywalny o nazwie **helloWorld** powinien zostać utworzony.

Aby uruchomić plik wykonywalny i zobaczyć, czy faktycznie na ekranie monitora pojawi się napis "Hello World!" w konsoli wpisz poniższe polecenie.

```
$ ./helloWorld
```

Gratulacje! Właśnie napisałeś, skompilowałeś oraz uruchomiłeś swój pierwszy program w języku C. Jeśli chcesz dowiedzieć się więcej na temat języka C, czytaj dalej ten materiał.

2.1.1 Struktura oraz opis kodu źródłowego języka C

Programy pisane w języku C oprócz ciała głównej funkcji wykorzystują jeszcze szereg innych funkcji:

- definiowanych przez programistę własnoręcznie (na listingu 2.1.1 nie występuje)
- zdefiniowanych w bibliotece standardowej (funkcja printf z listingu 2.1.1)

Każdy program może posiadać dowolną ilość funkcji, lecz warunkiem poprawnej kompilacji jest użycie funkcji main w kodzie programu. Jest to najważniejsza funkcja w programie, ponieważ skompilowany program wykonuje się od początku funkcji main aż do jej zakończenia.

Przykładowy, a zarazem bardzo prosty program pokazany na listingu 2.1.1 w pierwszej linii zawiera informację dla kompilatora, aby dołączył plik nagłówkowy stdio.h, który zawiera informacje na temat standardowego wejścia i wyjścia (standard input / output). Dzięki temu mogliśmy użyć funkcji printf, która jak już wiesz drukuje tekst na ekranie monitora (standardowe wyjście).

Kolejna linia to definicja głównej funkcji programu – funkcji main. Funkcja main oraz inne funkcje mogą przyjmować argumenty, jeśli funkcja przyjmuje jakieś argumenty, to zapisuje się je pomiędzy nawiasami. Jak widać w naszym przypadku, funkcja main nie przyjmuje żadnych argumentów (o argumentach przyjmowanych przez funkcję main oraz inne funkcje dowiesz się później).

W następnej linii występuje nawiast klamrowy otwierający ({). Pomiędzy nawiasami klamrowymi znajduje się ciało funkcji. W ciele funkcji występują definicje zmiennych lokalnych, wywołania funkcji bibliotecznych, wywołania funkcji napisanych przez programistów, generalnie rzecz biorąc, jeśli jakaś instrukcja ma zostać wykonana, to musi ona być wywołana w funkcji main. Ewentualnie, jeśli jakaś czynność, którą chcemy wykonać znajduje się w innej funkcji (w ciele innej funkcji), to dana funkcja musi zostać wywołana w ciele funkcji main.

Następna linia programu to wywołanie funkcji printf. Jak już wiesz, funkcja ta drukuje informacje zawarte w cudzysłowie (wyrażenie "Hello World!" jest argumentem funkcji). Nie mniej jednak może

być nie jasne dlaczego po znaku wykrzyknika występuje kombinacja znaków **\n**. Otóż za pomocą tej kombinacji znaków możemy przejść kursorem do nowej linii, czyli jeśli byśmy dodali w kolejnej linii następną instrukcję printf z dowolnym argumentem (dowolny napis zawarty w cudzysłowie) to uzyskalibyśmy go w nowej linii, pod napisem "Hello World!". Jeśli by nie było znaku nowej linii (**\n** – new line character) to łańcuch znaków przekazany jako parametr drugiej funkcji zostałby wyświetlony tuż za znakiem wykrzyknika, nawet bez spacji. Każda pojedyńcza instrukcja w języku C kończy się średnikiem. Nawias klamrowy zamykający (}) będący w ostatniej linii programu zamyka ciało funkcji main.

Tak więc wiesz już jak napisać prosty program w języku C, wiesz co oznaczają poszczególne części programu, oraz wiesz gdzie umieszcza się wywołania funkcji. W kolejnym podpunkcie znajdują się informacje o komentarzach.

2.1.2 Komentarze

Tworzenie komentarzy podczas pisania programów jest bardzo istotne. Oczywiście w programach, które mają stosunkowo mało linii kodu nie jest to, aż tak potrzebne. Nie mniej jednak w bardziej rozbudowanych programach (wieloplikowych) jest to ważne. Czasem napiszemy coś, zajrzymy do pliku po kilkunastu dniach i na nowo musimy czytać od początku co napisana przez nas samych funkcja robi. Komentarze są całkowicie ignorowane przez kompilator, tak więc można wpisywać tam dowolne zdania, skróty myślowe, itp. Komentarze dzieli się na dwa rodzaje:

- Komentarz liniowy
- Komentarz blokowy

Komentarz liniowy zaczyna się od dwóch znaków ukośnika (//) i kończy się wraz ze znakiem nowej linii. Jak sama nazwa wskazuje, komentarze te zajmują jedną linię, ponieważ przejście do nowej linii kończy komentarz. Komentarz blokowy zaczyna się od znaków **/***, a kończy się na znakach ***/**. Komentarze te mogą obejmować większą ilość wierszy niż jeden. Na listingu 2.1.2 pokazane zostały dwa sposoby wstawiania komentarzy.

```
#include <stdio.h>  // Dołącz plik nagłówkowy stdio.h – Komentarz liniowy
/* Tutaj mogą znajdować się prototypy funkcji
```

```
   Ale o tym trochę później....
   Komentarz wieloliniowy – blokowy
*/
main ()
{
     printf("Hello World!\n");
}
```

Listing 2.1.2 Komentarze w kodzie

2.2 Zmienne i stałe

Zmienne i stałe to obiekty, które zajmują pewien obszar w pamięci komputera, do którego możemy się odwołać podając ich nazwę lub adres (wskaźnik). Do zmiennej można wpisywać oraz zmieniać (w trakcie działania programu) wartości zależne od jej typu. Do stałych przypisujemy wartość raz (w kodzie źródłowym programu) i jej już zmienić nie możemy.

2.2.1 Typy zmiennych

Każda zmienna lub stała ma swój typ, co oznacza tyle, że może przyjmować wartości z zakresu danego typu. W poniższej tabeli przedstawione zostały typy zmiennych oraz stałych wraz z opisem jakie wartości przyjmują. Zakresy zmiennych zostały przedstawione w punkcie 2.2.2.

Typ zmiennej (stałej)	Przyjmowane wartości
int	Liczby całkowite
float	Liczby rzeczywiste pojedynczej precyzji
double	Liczby rzeczywiste podwójnej precyzji
char	Zazwyczaj pojedyncza litera (pojedynczy bajt)
short int	Krótsze liczby całkowite, niż int
long int	Dłuższe liczby całkowite, niż int
long long int	Bardzo duże liczby całkowite
long double	Dłuższe liczby rzeczywiste, niż double

Tabela 2.2.1 Typy zmiennych i stałych

Istnieją jeszcze dodatkowe przedrostki (kwalifikatory), które można dodać przed typem zmiennej, tymi słowami są:

- signed – Przedrostek umożliwiający definicję liczb dodatnich oraz ujemnych (standardowo)
- unsigned – Przedrostek umożliwiający definicję liczb tylko dodatnich oraz zera.

2.2.2 Zakres typów zmiennych

Zakresy typów zmiennych są istotnym zagadnieniem podczas pisania programu, nie można przekroczyć zakresu danego typu, ponieważ program może zachować się nie tak jakbyśmy tego chcieli. Przekroczenie zakresu w przypadku zmiennej typu int prowadzi do ustawienia w zmiennej wartości ujemnej, tzn najmniejszej wartości jaki typ int obsługuje. W poniższej tabeli znajduje się zestawienie zakresów poszczególnych typów.

Typ zmiennej	**Rozmiar (bajty)**[1]	**Zakres**	
		Od	**Do**
int	4	-2 147 483 648	2 147 483 647
float	4	$1.5 \cdot 10^{-45}$	$3.4 \cdot 10^{38}$
double	8	$5.0 \cdot 10^{-324}$	$3.4 \cdot 10^{308}$
char	1	-128	127
short int	2	-32 768	32 767
long int	4	-2 147 483 648	2 147 483 647
long long int	8	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
long double	12	$1.9 \cdot 10^{-4951}$	$1.1 \cdot 10^{4932}$

Tabela 2.2.2 Zakresy zmiennych oraz rozmiary dla liczb ze znakiem (signed)

Typ zmiennej	**Rozmiar (bajty)**	**Zakres**	
		Od	**Do**
unsigned int	4	0	4 294 967 295
unsigned char	1	0	255

1 Sprawdzone na 32-bitowym procesorze, na innych może się różnić.

unsigned short int	2	0	65535
unsigned long int	4	0	4 294 967 295
unsigned long long int	8	0	18 446 744 073 709 551 615

Tabela 2.2.3 Zakres zmiennych oraz rozmiary dla liczb bez znaku (unsigned)

2.2.3 Nazwy zmiennych i deklaracja zmiennych

Deklaracja zmiennych w języku C jest bardzo prosta. Po pierwsze podajemy jej typ, po drugie podajemy jej nazwę, na końcu definicji stawiamy średnik. Jeśli tworzymi kilka zmiennych danego typu, to możemy wypisywać ich nazwy po przecinku. Przykładowe definicje zmiennych lokalnych, oraz globalnych pokazane zostały na listingu 2.2.1

```
#include <stdio.h>
unsigned short numer;
unsigned id = 10;
main()
{
	const float podatek = 0.22;
	int i, k = 2, z;
	unsigned int iloscLudzi;
	int dolna_granica = -10;
	float cenaKawy = 5.4;
}
```

Listing 2.2.1 Definicja zmiennych

Nazwy zmiennych mogą być dowolnymi ciągami znaków, mogą zawierać cyfry przy czym cyfra nie może być pierwszym znakiem. Znak podkreślenia również jest dozwolony. Trzeba mieć na uwadze fakt, iż wielkość liter jest rozróżniana! Stała podatek jest zdefiniowana, natomiast wyraz **Podatek** nie jest stałą typu float!

Zaleca się, aby nazwy zmiennych były związane z ich docelowym przeznaczeniem. Jak widać na listingu 2.2.1 czytając nazwy użytych zmiennych mamy poniekąd informację do czego będą służyć i dlaczego takie, a nie inne typy zmiennych zostały użyte. Nazwy zmiennych takie jak i, k, z służą zazwyczaj do sterownia pętlami.

Zmienne globalne deklarowane są poza ciałem funkcji. Dodawanie przedrostka określającego jego długość bądź znak, lub jednocześnie oba można zapisać w postaci pełnej, czyli np. unsigned short int

nazwaZmiennej; bądź w skróconej formie (unsigned short nazwaZmiennej), jak pokazano na listingu 2.2.1, czyli nie pisząc słowa kluczowego int. Pisanie skróconej formy oznacza, że typem zmiennej będzie int!

Wartości zmiennych mogą zostać zainicjonowane podczas tworzenia zmiennych. Po nazwie zmiennej wstawiamy znak równości i wpisujemy odpowiednią (do typu zmiennej) wartość (wartości jakie można wpisywać przedstawione zostały w tabelach: 2.2.4 oraz 2.2.5 w podpunkcie 2.2.4 Stałe).

Jeśli tworzmy zmienną globalną i nie zainicjujemy jej żadnej wartości, to kompilator przypisze jej wartość zero, co jest odróżnieniem od zmiennych lokalnych, które nie zainicjonowane przez programistę posiadają śmieci (losowe wartości, które były w danym obszarze pamięci przed jej zajęciem).

2.2.4 Stałe

Można powiedzieć, że stałe to zmienne tylko do odczytu. Raz przypisana wartość do stałej podczas pisania kodu nie może zostać zmieniona przez użytkownika podczas używania programu. Stałą definiuje się poprzez użycie słowa kluczowego const przed typem i nazwą zmiennej. A więc deklaracje stałych typu float oraz int wyglądają następująco:

```
const float nazwaStalejFloat = yyy;        // (1)

const int nazwaStalejInt = xxx;            // (2)
```

Gdzie jako yyy, xxx możemy wpisać jedną z wartości przedstawionych w poniższych tabelach. W deklaracji (1) zamiast float można wpisać double w celu uzyskania większej dokładności (podwójna precyzja). Analogicznie w deklaracji (2) int może zostać zamieniony na long int, lub inny typ w celu podwyższenia bądź zmniejszenia zakresu.

yyy	Opis przykładowej przypisywanej wartości
10.4	Stała zmiennopozycyjna zawierająca kropkę dziesiętną
104E-1	Stała zmiennopozycyjna zawierająca wykładnik[1]

1 104E-1 = $104 \cdot 10^{-1}$ = 10.4

1.24E-3	Stała zmiennopozycyjna zawierająca kropkę dziesiętną oraz wykładnik

Tabela 2.2.4 Różne sposoby wpisywania wartości liczb rzeczywistych

xxx	**Opis przykładowej przypisywanej wartości**
145	Stała całkowita dziesiętna
10E+5	Stała całkowita dziesiętna z wykładnikiem[2]
0230	Stała całkowita zapisana w systemie ósemkowym (zero na początku)
0x143	Stała całkowita zapisana w systemie szesnastkowym (0x, lub OX na początku)[3]

Tabela 2.2.5 Różne sposoby wpisywania wartości liczb całkowitych

Przykładowe definicje stałych:

```
const int dwaMiliony = 2E6;                   // 2000000
const int liczbaHex = 0x3E8;                  // 3E8 to dziesiętnie 1000
const double malaLiczba = 23E-10;             // 0.0000000023
```

Aby sprawdzić, czy faktycznie tak jest, w ciele funkcji main wpisz poniższą linijkę kodu, która zawiera funkcję printf. O funkcji printf trochę więcej informacji zostanie podane później.

```
printf(”%d\n”, liczbaHex);
```

2.2.5 Wyrażenia stałe i stałe symboliczne

Wyrażenia stałe są to wyrażenia, które nie zależą od zmiennych. Czyli mogą zawierać stałe (const), stałe wyliczenia (enum), zwykłe wartości na sztywno wpisane w kod programu bądź stałe symboliczne.

2 Równoważny zapis: 10E5
3 Równoważny zapis: OX143

Stałą symboliczną definiuje się poza funkcjami, czyli jest globalna (dostępna dla wszystkich funkcji). Stałe symboliczne tworzy się za pomocą dyrektywy preprocesora (części kompilatora) #define. Na listingu 2.2.2 pokazane zostały cztery przykłady użycia stałej symbolicznej.

```
#include <stdio.h>
#define MAX 10
#define ILOCZYN(x,y) (x)*(y)
#define DRUKUJ(wyrazenie) printf(#wyrazenie " = %g\n", wyrazenie)
#define POLACZ_TEKST(arg1, arg2) arg1 ## arg2
main ()
{
     double POLACZ_TEKST(po, datek) = 0.22;

     printf("MAX: %d\n", MAX);
     printf("ILOCZYN: %d\n", ILOCZYN(MAX,MAX));
     DRUKUJ(10.0/5.5);
     printf("%.2f\n", podatek);
}
```

Listing 2.2.2 Użycie #define.

Druga linia powyższego kodu definiuje stałą symboliczną MAX o wartości 10. Kompilator podczas tłumaczenia kodu zamieni wszystkie wystąpienia stałej MAX na odpowiadającą jej wartość. Linia trzecia definiuje tak jakby funkcję[1] ILOCZYN, która przyjmuje dwa argumenty i je wymnaża. Czwarta linia programu tworzy makro, które po wywołaniu wstawi wyrazenie w miejsce #wyrazenie (znak # jest obowiązkowy) oraz obliczy je i wstawi w miejsce deskryptora formatu (**%g**), a wszystko to zostanie wydrukowane za pomocą funkcji printf. Za pomocą operatora ## skleja się argumenty. Składnia dla tego operatora jest taka jak pokazano w piątej linii listingu 2.2.2. W funkcji main używamy tego makra do połączenia słów po i datek, co w efekcie daje podatek. Jako iż przed nazwą stoi słowo double, a po nazwie inicjacja wartości, to słowo podatek staje się zmienną typu double. Pierwsza instrukcja printf drukuje liczbę 10, ponieważ stała MAX posiada taką wartość, natomiast druga wydrukuje wynik mnożenia liczby, która kryje się pod nazwą MAX. Oczywiście wynikiem będzie liczba 100.

2.2.6 Stała wyliczenia

Stała wyliczenia jest tworzona przy pomocy słowa kluczowego enum. Idea polega na tym, że nazwom

1 Więcej informacji o funkcjach znajduje się w rozdziale Funkcje.

zawartym w nawiasach klamrowych przyporządkowywane są liczby całkowite, począwszy od 0. Poniższy przykład ilustruje omówione zachowanie.

```
#include <stdio.h>
main ()
{
     enum wyliczenia {NIE, TAK};
     printf("%d\n", NIE);       // 0
     printf("%d\n", TAK);       // 1
}
```

Listing 2.2.3 Użycie stałej wyliczenia.

Można zdefiniować stałą wyliczenia, w której sami zadecydujemy jakie wartości będą przyporządkowane do kolejnych słów pomiędzy nawiasami klamrowymi. Jeśli nie zadeklarujemy wszystkich, to kompilator uzupełni je w sposób następujący: Znajduje ostatnią zdefiniowaną wartość wyrazu i przypisuje do kolejnego wyrazu zwiększoną wartość o jeden. Listing 2.2.4 pokazuje jak się definiuje nowe wartości i jak kompilator dopełnia te, które nie zostały uzupełnione.

```
#include <stdio.h>
main ()
{
     enum tydzien {PON = 1, WTO, SRO, CZW, PT, SOB, ND};
     printf("%d\n", PON);  // 1 - Zdefiniowane
     printf("%d\n", WTO);  // 2 - Dopełnione: Do wartości PON dodane 1
     printf("%d\n", SRO);  // 3 - WTO + 1
     printf("%d\n", CZW);  // 4 - Analogicznie pozostałe
     printf("%d\n", PT);   // 5
     printf("%d\n", SOB);  // 6
     printf("%d\n", ND);   // 7
}
```

Listing 2.2.4 Użycie stałej wyliczenia wraz z definicją wartości

Zastosowanie stałych wyliczenia zostanie pokazane w rozdziale dotyczącym sterowania programem – instrukcja switch.

2.2.7 Zasięg zmiennych

Zasięg zmiennych jest bardzo istotnym zagadnieniem, ponieważ możemy czasem próbować odwołać się do zmiennej, która w rzeczywistości w danym miejscu nie istnieje. Listing 2.2.5 pokazuje trochę

dłuższy kawałek kodu, natomiast uświadamia istotne aspekty związane z zasięgiem zmiennych.

```
#include <stdio.h>

int iloscCali = 10;

void drukujZmienna (void);

main ()
{
	int index = 5;
	printf("%d\n", index);
	printf("%d\n", iloscCali);
	drukujZmienna();
	{
		int numer = 50;
		printf("%d\n", numer);
	}
	printf("%d\n", numer);		/* Nie zadziala */
}

void drukujZmienna (void)
{
	int id = 4;
	printf("%d\n", id);
	printf("%d\n", iloscCali);
}
```

Listing 2.2.5 Zakresy zmiennych

Widać nowość na powyższym listingu, są dwie funkcję: main oraz drukujZmienna. Zakres zmiennej lokalnej, czyli takiej, która utworzona jest w dowolnej funkcji obejmuje tylko tę funkcję. To znaczy zmienna id, która jest zadeklarowana w funkcji drukujZmienna dostępna jest tylko w tej funkcji. Z funkcji main nie można się do niej odwołać i odwrotnie, zmienna index dostępna jest tylko w głównej funkcji programu. Ciekawostką może być fakt, iż zmienna numer pomimo tego, że zadeklarowana jest w funkcji głównej, nie jest dostępna w każdym miejscu funkcji main. Nawiasy klamrowe tworzą wydzielony blok, w którym utworzone zmienne dostępne są tylko pomiędzy klamrami (czyli w tym bloku). Dlatego też ostatnia instrukcja nie zadziała – kompilator zgłosi błąd, który mówi, że nie można używać zmiennej, jeśli się jej wcześniej nie zadeklaruje.

Zmienna globalna iloscCali widziana jest w każdym miejscu, tzn można jej używać w każdej funkcji oraz musi być zdefiniowana dokładnie jeden raz.

2.3 Matematyka

2.3.1 Operatory arytmetyczne

Lista operatorów arytmetycznych została podana w poniższej tabeli. Operatory te są operatorami dwuargumentowymi, czyli jak sama nazwa mówi, potrzebują dwóch argumentów.

Operator	Funkcja operatora
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Dzielenie modulo (reszta z dzielenia)

Tabela 2.3.1 Operatory arytmetyczne

Poniżej pokażę deklarację zmiennych, użycie operatorów, oraz wyświetlenie wyniku. Wszystkie poniższe instrukcję proszę wpisać w ciele funkcji main.

```
int a, b, wynik;
a = 10;
b = 7;
wynik = a + b;              // wynik = a - b; lub wynik = a * b;
printf(”%d\n”, wynik);
```

A co z dzieleniem? Z dzieleniem jest w zasadzie tak samo, natomiast trzeba wspomnieć o bardzo ważnej rzeczy. Dzielenie liczb całkowitych (typ int) w wyniku da liczbę całkowitą, czyli cyfry po przecinku zostaną obcięte. Aby tego uniknąć, tzn aby w wyniku dostać liczbę rzeczywistą przynajmniej jeden z argumentów musi być liczbą rzeczywistą (float, double) oraz zmienna przetrzymująca wynik też musi być typu rzeczywistego.

Aby dokonać tego o czym wspomniałem (odnośnie typu rzeczywistego jednego z argumentów) można postąpić na kilka sposobów. Pierwszy z nich, chyba najprostszy – zadeklarować argument jako zmienną typu rzeczywistego.

```
float a, wynik;
int b;
```

```
a = 10.0;
b = 7;
wynik = a / b;                  // wynik = 1.428571
printf(”%f\n”, wynik);
```

Drugim sposobem jest pomnożenie jednego z argumentów przez liczbę rzeczywistą, czyli żeby nie zmienić tej liczby, a zmienić tylko jej typ, możemy pomnożyć ją przez 1.0.

```
int a, b;
float wynik;
a = 10;
b = 7;
wynik = a*1.0 / b;              // wynik = 1.428571
printf(”%f\n”, wynik);
```

Trzeci sposób korzysta z operatora rzutowania (o nim jeszcze nie było wspomniane). Operator rzutowania ma postać:

```
(typ_rzutowania) wyrazenie;
```

Co oznacza tyle, że rozszerza, bądź zawęża dany typ zmiennej ustawiając nowy. Jeśli wyrazenie jest zmienną typu int to po zrzutowaniu na float będzie tą samą liczbą tylko dodatkowo wartość jej będzie składać się z kropki, po której nastąpi zero (np. 10.0). W drugą stronę też można, jeśli zmienna jest typu float, a zrzutujemy ją na int to końcówka, czyli kropka i liczby po kropce zostaną obcięte i zostanie sama liczba całkowita. A więc w naszym przypadku można by było zrobić to w następujący sposób:

```
int a, b;
float wynik;
a = 10;
b = 7;
wynik = (float)a / b;
printf(”%f\n”, wynik);
```

Dzielenie modulo powoduje wyświetlenie reszty z dzielenia dwóch argumentów. Operator reszty

z dzielenia używany jest tylko dla liczb (typów) całkowitych. Na poniższym przykładzie zostało to pokazane.

```
int a = 10, b = 7, c = 5, wynik;
wynik = a % b;              // 3
printf(”%d\n”, wynik);
wynik = a % c;              // 0
printf(”%d\n”, wynik);
```

2.3.2 Operatory logiczne i relacje

W języku C istnieją operatory logiczne, dzięki którym możemy sprawdzać warunki, które z kolei mogą sterować programem. Operatory logiczne zostały przedstawione w niniejszej tabeli.

Operator	Funkcja operatora
Relacje	
>	Większy niż
>=	Większy lub równy niż
<	Mniejszy niż
<=	Mniejszy lub równy niż
Operatory przyrównania	
==	Równy
!=	Różny
Operatory logiczne	
&&	Logiczne „i”
||	Logiczne „lub”

Tabela 2.3.2 Relacje, operatory przyrównania oraz operatory logiczne

Każdy z wyżej wymienionych operatorów jest operatorem dwu argumentowym, więc jego sposób użycia i zastosowanie można przedstawić na poniższym przykładowym kodzie:

```
#include <stdio.h>

main ()
{
  const int gornaGranica = 10;
```

```
  const int dolnaGranica = -10;
  int a = 4;

  if (a >= dolnaGranica && a <= gornaGranica)
      if (a % 2 == 0)
      {
          printf("Liczba a (%d) zawiera sie w przedziale: ", a);
          printf("<%d;%d> i jest parzysta\n", dolnaGranica, gornaGranica);
      }
      else
      {
          printf("Liczba a (%d) zawiera sie w przedziale: ", a);
          printf("<%d;%d> i jest nie parzysta\n", dolnaGranica,
gornaGranica);
      }
  else
      printf("Liczba nie zawiera sie w podanym zakresie\n");
}
```

Listing 2.3.1 Użycie operatorów

W tym miejscu skupimy się bardziej tylko na użyciu operatorów, w jaki sposób się ich używa, jak to działa itp. Działanie instrukcji if–else zostanie omówione w podpunkcie 3.1.

Spójrzmy na poniższe wyrażenie, w którym użyto dwóch operatorów relacji i jednego operatora logicznego.

```
a >= dolnaGranica && a <= gornaGranica
```

Priorytety operatorów zostały opisane w podpunkcie 2.3.5, ale tak krótko: Operatory **>=** i **<=** mają priorytet większy niż operator &&, dzięki tej informacji nie musimy stosować nawiasów, ponieważ najpierw wykona się warunek sprawdzający czy a jest większe lub równe od dolnej granicy (dolnaGranica). W tym momencie trzeba troszkę nawiązać do tego co to znaczy logiczne „i”. W tabeli poniżej zostały podane kombinacje bramki logicznej „i”, z której korzysta operator &&.

X1	**X2**	**Y**
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 2.3.3 Bramka logiczna „i”

Potraktujmy wyrażenie a >= dolnaGranica jako zmienną X1, wyrażenie a <= gornaGranica jako zmienną X2[1]. A całość a >= dolnaGranica && a <= gornaGranica jako zmienną Y.

Jeśli pierwszy warunek jest spełniony, czyli a jest większe bądź równe wartości zmiennej dolnaGranica, to to wyrażenie przyjmuje wartość 1, można sobie to wyobrazić, że do zmiennej X1 przypisujemy wartość 1. Teraz spójrz na tabelę 2.3.3, Y równa się 1 wtedy i tylko wtedy, gdy X1 i X2 równają się jednocześnie 1. Skoro nasze X1 przyjęło wartość 1, to jest sens sprawdzenia drugiego warunku, które oznaczyliśmy jako X2. Jeśli X2 równa się 1, czyli warunek został spełniony (a mniejsze lub równe wartości zmiennej gornaGranica) to Y równa się 1. Jeśli Y równa się 1 to zostanie sprawdzony kolejny warunek, na parzystość liczby a (ponieważ, jeśli Y równa się 1, to if(1) jest wartością prawdziwą i zostaną wykonywane polecenia po if, jeśli Y równałby się 0, to if(0) jest fałszywe, więc wykonały by się instrukcje znajdujące się po else[2]). Jeśli liczba a dzieli się przez 2 bez reszty (czyli reszta z dzielenia liczby a przez 2 równa się 0) to liczba jest parzysta, w przeciwnym wypadku liczba jest nie parzysta.

W gruncie rzeczy to by było tyle jeśli chodzi o operatory. Warto zaznajomić się z priorytetami, które operatory mają wyższe, które niższe. W razie nie pewności można użyć nawiasów, które zapewniają wykonanie instrukcji w nawiasie przed tymi spoza nawiasów.

2.3.3 Operatory zwiększania, zmniejszania oraz przypisywania

W poniższej tabeli znajduje się zestawienie operatorów zwiększania oraz zmniejszania, a pod tabelą sposób użycia.

Funkcja operatora	Deklaracja operatora	Nazwa operatora
Zwiększanie	n++	Post – inkrementacja
	++n	Pre – inkrementacja
Zmniejszanie	n--	Post – dekrementacja
	--n	Pre – dekrementacja

Tabela 2.3.4 Operatory zwiększania, zmniejszania

1 Możemy to traktować jako zmienną, ponieważ równie dobrze w kodzie z listingu 2.3.1 moglibyśmy zdefiniować zmienną int X1 = a >= dolnaGranica; oraz zmienną int X2 = a <= gornaGranica; i w warunku wstawić if (X1 && X2). Sposób pisania jest dowolny i zależy od przyzwyczajeń programisty.

2 Omówienie sposobu działania instrukcji if-else znajduje się w podpunkcie 3.1

Poniżej pokaże sposób użycia tych operatorów w kodzie, oraz wytłumaczę zasadę ich działania. Przykład będzie z operatorem inkrementacji (zwiększania). Zasada działania operatora dekrementacji jest analogiczna.

```
#include <stdio.h>

main ()
{
	int a = 0, n = 0;
	a = n++;
	printf("a = %d\n", a);		// 0
	printf("n = %d\n", n);		// 1

	a = 0;
	n = 0;

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

Listing 2.3.2 Użycie operatora przypisania oraz zwiększania

A więc tak, operator post – inkrementacji tak samo jak operator pre – inkrementacji zwiększa wartość swojego argumentu. Różnica między nimi jest taka, że z pomocą operatora post – inkrementacji zmienna a w wyrażeniu a = n++ będzie zawierała „starą” wartość zmiennej n, czyli tą przed zwiększeniem. Operator ten działa w taki właśnie sposób, że zwiększa wartość zmiennej dopiero po tym jak zostanie ona użyta.

Operator pre – inkrementacji zwiększa wartość zmiennej jeszcze przed jej użyciem, dlatego w tym przypadku zmienna a i n będą miały wartość 1.

Operatory pre i post – dekrementacji działają w analogiczny sposób, tylko, że zmniejszają wartość swojego argumentu.

Wyrażenie, które ma postać n = n + 5 można i z reguły zapisuje się w innej, krótszej postaci za pomocą operatora przypisania, definiowanego jako n += 5. Ogólna postać operatora przypisania to: X= Gdzie X może być jednym z następujących znaków:

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

Poniżej występują przykładowe deklaracje operatora przypisania.

```
int i = 2, k = 3;
i += k;          // i = i + k; i = 5
i *= k + 2       // i = i * (k + 2); i = 10
```

2.3.4 Operatory bitowe

Aby dobrze zrozumieć operację na bitach, trzeba zrobić pewne wprowadzenie o liczbach binarnych (bin). W tym miejscu nie będę się skupiał na sposobie w jaki się przelicza liczby z jednego systemu na drugi, bo nie to jest celem naszych rozważań. Do wszystkich tych czynności należy użyć kalkulatora, który potrafi wyświetlać wartości w różnych systemach liczbowych.

A więc tak, najpierw opiszę operatory bitowe, a później na przykładach pokaże zasadę ich działania wraz z opisem. W tabeli poniżej znajdują się operatory bitowe oferowane przez język C. Operatory te można stosować do manipulowania bitami jedynie argumentów całkowitych!

Operator	**Nazwa operatora**
&	Bitowa koniunkcja (AND)
|	Bitowa alternatywa (OR)
^	Bitowa różnica symetryczna (XOR)
<<	Przesunięcie w lewo
>>	Przesunięcie w prawo
~	Dopełnienie jedynkowe

Tabela 2.3.5 Operatory bitowe

Zacznijmy więc od bitowej koniunkcji (&). Bitowa koniunkcja używana jest do zasłaniania (zerowania) pewnych bitów z danej liczby. Przydatną rzeczą może okazać się tabela 2.3.3 z punktu 2.3.2, która to jest tablicą prawdy dla logicznej koniunkcji (logiczne „i”).

Dajmy na przykład liczbę dziesiętną 1435, której reprezentacją binarną jest liczba 10110011011 (wiersz pierwszy). Ogólnie użycie operatora koniunkcji bitowej można rozumieć jako porównanie parami bitów liczby, na której operację wykonujemy oraz liczby, o której zaraz powiem.

Jeśli chcemy zasłonić pewną ilość bitów, np. pięć bitów licząc od lewej strony. Musimy w takim wypadku do drugiego wiersza wstawić zera pod bitami, które chcemy zasłonić, a na resztę bitów wstawić jedynki (wiersz drugi).

1	0	1	1	0	0	1	1	0	1	1
0	0	0	0	0	1	1	1	1	1	1

Dzieje się tak ponieważ taka jest zasada działania bramki logicznej AND (logiczne „i"). Wszędzie tam gdzie występuje choć jedno zero, wynikiem ogólnie będzie zero (wstawiamy zero, więc zerujemy wynik danego bitu). A tam gdzie wstawimy jedynkę, wartości nie zmienimy (może być zero, a może być jeden).

To był taki wstęp teoretyczny, żeby wiedzieć o co tam w ogóle chodzi. Teraz czas przejść do tego, jaką trzeba liczbę użyć, by zasłonić taką, a nie inną ilość bitów. Bierzemy liczbę z drugiego wiersza, zamieniamy ją na wartość ósemkową (liczbę ósemkową w C wstawiamy poprzedzając ją zerem!, np. 077), bądź szesnastkową (liczbę szesnastkową wstawiamy poprzedzając ją 0x, lub 0X) i wstawiamy do polecenia. Listing 2.3.3 pokazuje jak to zrobić.

```
#include <stdio.h>

main ()
{
     int liczba;
     liczba = 1435;
     liczba = liczba & 0x3F;          // 0x3F = 00000111111
     printf("%d\n", liczba);          // 27
}
```

Listing 2.3.3 Użycie operatora koniunkcji bitowej

Wynikiem jest liczba 27, ponieważ pięć bitów licząc od lewej zostało wyzerowanych i z naszej binarnej liczby 10110011011 została liczba 011011, co po zamienieniu na dziesiętną wartość daje 27. Alternatywny sposób zasłaniania pewnej ilości bitów (sposób bardziej praktyczny) znajduje się w zadaniach: 2.6 oraz 2.7.

Bitowa alternatywa (|) działa w trochę inny sposób, niż bitowa koniunkcja, która „czyściła" pewną ilość bitów, a mianowicie ustawia bity na tych pozycjach, na których chcemy.

Dla przykładu weźmy liczbę dziesiętną 1342, której reprezentacją binarną jest liczba 10100111110. Tworzymy tabelkę tak jak w poprzednim przykładzie i z pomocą operatora bitowej alternatywy możemy ustawić bity, na dowolnej pozycji. Właściwie to możemy zmienić z zera na jeden konkretny bit. Operacja zamiany z jedynki na zero, to bitowa koniunkcja. Tabela 2.3.6 przedstawia tablicę prawdy

dla logicznego „lub”.

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 2.3.6 Tablica prawdy logicznego „lub” (OR)

Powyższa tabela może okazać się przydatna w zrozumieniu sposobu działania operatora bitowej alternatywy. Czyli mówiąc w skrócie działa to na zasadzie takiej, jeśli porównamy dwa bity logicznym „lub”, to nie zmieni ostatecznej wartości bitu wartość zero. Natomiast jeśli chcemy zmienić wartość to musimy wstawić jedynkę. Tabela 2.3.6 pokazuje to dokładnie. Jeśli gdziekolwiek występuje jedynka, to wartością końcową jest jedynka.

Powracając do naszego przykładu, załóżmy, że chcemy zrobić liczbę dziesiętną 2047, której reprezentacją binarną jest liczba 11111111111. Wpisujemy do tabeli liczbę, która pod zerami będzie miała jedynki. Tą liczbą jest 01011000001, jej reprezentacją w systemie szesnastkowym jest 2C1.

1	0	1	0	0	1	1	1	1	1	0
0	1	0	1	1	0	0	0	0	0	1

Bardzo podobnie do poprzedniego wygląda niniejszy listing. Różnicą oczywiście jest operator bitowy.

```
#include <stdio.h>

main ()
{
	int liczba;
	liczba = 1342;
	liczba |= 0x2C1;			// liczba = liczba | 0x2C1;
	printf("%d\n", liczba);		// 2047
}
```

Listing 2.3.4 Użycie operatora bitowej alternatywy

Operator bitowej różnicy symetrycznej (^) ustawia wartość jeden, jeśli dwa porównywane ze sobą bity mają różne wartości. Tabela 2.3.7 pokazuje tablicę prawdy dla logicznej różnicy symetrycznej.

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 2.3.7 Tabela prawdy logicznej różnicy symetrycznej

Weźmy na przykład liczbę dziesiętną 1735, której reprezentacją binarną jest liczba 11011000111. Tworzymy po raz kolejny tabelkę i wpisujemy do pierwszego wiersza reprezentację binarną naszej liczby. Spójrzmy na tabelę 2.3.7 żeby zmienić ostatecznie wartość bitu na zero, to pod jedynkami musimy wpisać jedynki, a pod zerami zera. Żeby wartość bitu została zmieniona na jeden, to pod jedynkami musimy wpisać zero, a pod zerami jeden.

Aby przerobić naszą liczbę na liczbę 1039, której reprezentacją binarną jest liczba 10000001111 musimy wpisać takie liczby w wierszu drugim, by po sprawdzeniu ich z tablicą prawdy różnicy symetrycznej uzyskać wartość binarną liczby 1039.

1	1	0	1	1	0	0	0	1	1	1
0	1	0	1	1	0	0	1	0	0	0

Wartością w drugim wierszu tabeli jest ciąg cyfr 01011001000, którego reprezentacją szesnastkową jest 2C8. Listing 2.3.5 pokazuje już znaną metodę operacji na bitach.

```
#include <stdio.h>

main ()
{
     int liczba;
     liczba = 1735;
     liczba ^= 0x2C8;                 // liczba = liczba ^ 0x2C8;
     printf("%d\n", liczba);          // 1039
}
```

Listing 2.3.5 Użycie operatora bitowej różnicy symetrycznej (XOR)

Operator przesunięcia służy jak sama nazwa wskazuje do przesunięcia bitów. Jeśli mamy liczbę dziesiętną 28, której reprezentacją binarną jest liczba 11100, to użycie operatora przesunięcia w lewo spowoduje przesuniecie wszystkich bitów w lewo o konkrerną ilość pozycji. Poniższa tabela prezentuje

to zachowanie. W pierwszym wierszu wpisana jest binarna wartość dziesiętnej liczby 28. W wierszu drugim po wykonaniu przesunięcia w lewo o 2. Jak widać, po przesunięciu z prawej strony zostały dopisane zera. Listing 2.3.6 pokazuje jak używa się przesunięcia w lewo w języku C.

0	0	1	1	1	0	0
1	1	1	0	0	0	0

```
#include <stdio.h>

main (void)
{
	int liczba = 28;
	liczba <<= 2;                    // liczba = liczba << 2;
	printf("%d\n", liczba);          // 112
}
```

Listing 2.3.6 Użycie operatora przesunięcia w lewo.

Operator przesunięcia w prawo działa analogicznie do tego opisanego przed chwilą, z tą różnicą, że przesuwa bity w prawo. Zakładając więc, że przesuwamy liczbę 28, poniższa tabela pokazuje, że po przesunięciu wartością naszej liczby będzie 7.

0	0	1	1	1	0	0
0	0	0	0	1	1	1

```
#include <stdio.h>

main (void)
{
	int liczba = 28;
	liczba >>= 2;                  // liczba = liczba >> 2;
	printf("%d\n", liczba);        // 7
}
```

Listing 2.3.7 Użycie operatora przesunięcia w prawo

Ostatni operator bitowy to operator dopełnienia jedynkowego. Operator ten przyjmuje jeden argument i neguje wszystkie bity (czyli zamienia ich wartości z zera na jeden i odwrotnie). Niech przykładem będzie liczba 77, której reprezentacją binarną jest liczba 1001101. Tworzymy więc tabelę tak jak w poprzednich przykładach i wpisujemy tę wartość do pierwszego wiersza.

0	…	0	1	0	0	1	1	0	1
1	…	1	0	1	1	0	0	1	0

W drugim natomiast wierszu zostały pokazane bity po negacji. Nasza liczba (77) składała się z większej ilości bitów niż 7. W przykładzie poniższym użyto typu unsigned, który zajmuje 32 bity. Reszta bitów przed negacją miała wartość zero, która nie zmieniała wyniku (lewa strona tabeli). Po negacji wszystkie te zera zostały zamienione na jedynki i wynik jest bardzo dużą liczbą. Poniższy listing pokazuje omówione zachowanie.

```
#include <stdio.h>

main ()
{
      unsigned x;
      x = 77;
      /*  x: 00000000000000000000000001001101
       * ~x: 11111111111111111111111110110010
       */

      printf("%u\n", ~x);   // 4294967218
}
```

Listing 2.3.8 Użycie operatora bitowej negacji

Aby nasz wynik był tym, którego oczekujemy (tabela – drugi wiersz; tło zaznaczone kolorem) musimy zasłonić pozostałe bity. Do tego zadania użyjemy operatora bitowej koniunkcji. Skoro chcemy, aby tylko 7 bitów było widocznych, to musimy ustawić na nich jedynkę. Wartością, która będzie nam potrzebna to 1111111, a szesnastkowo 7F. Tak więc podany niżej listing pokazuje, że po negacji oraz zasłonięciu otrzymujemy liczbę zgodną z założeniami, czyli 50.

```
#include <stdio.h>

main ()
{
      unsigned x;
      x = 77;               // 00000000000000000000000001001101
      x = ~x;               // 11111111111111111111111110110010
      x &= 0x7F;            // 00000000000000000000000000110010

      printf("%u\n", x);     // 50
}
```

Listing 2.3.9 Otrzymana liczba zgodna z założeniem

Aby pokazać jakieś zastosowanie tych operatorów pokażę rozwiązania zadań 2.6 oraz 2.7 z książki: „*Język ANSI C”* autorstwa Dennisa Ritchie oraz Briana Kernighana. W zadaniach tych należało napisać funkcje i tak właśnie je tutaj zaprezentuję. W razie, gdyby coś było nie jasne, to w rozdziale 4. zostały opisane funkcje.

Zadanie 2.6

Napisz funkcję setbits(x, p, n, y) zwracającą wartość x, w której n bitów – poczynając od pozycji p – zastąpiono przez n skrajnych bitów z prawej strony y. Pozostałe bity x nie powinny ulec zmianie.

Odpowiedź:

Przede wszystkim, aby zrozumieć sens tego zadania należy narysować sobie tabelki tak jak te poniżej. Obrazuje to problem, dzięki czemu łatwiej możemy sobie wyobrazić jakie bity mają być zamienione na jakie. Tak więc zadanie rozwiążemy w kilku poniższych krokach.

Krok 1

Przyjmujemy dwie dowolne liczby za x, oraz y i wpisujemy je do tabeli. Niech wartością x będzie 1023, której reprezentacją binarną jest 1111111111, a jako y przyjmijmy wartość 774, binarnie 1100000110. Jako n rozumiemy ilość bitów do zamiany, a jako p pozycję od której te bity zamieniamy. Przyjmijmy za n wartość 5, a za p wartość 7. Kolorem żółtym został zaznaczony obszar, w który mamy wstawić bity z obszaru „szarego”.

x	1	1	1	1	1	1	1	1	1	1
y	1	1	0	0	0	0	0	1	1	0
Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 2

W kroku drugim negujemy wartość y za pomocą operatora dopełnienia jedynkowego. Można by się zastanawiać dlaczego to robimy. Jest to wyjaśnione w kroku czwartym. Póki co tworzymy tabelkę i zapisujemy zanegowaną wartość y. W programie przypiszemy zanegowaną wartość y do zmiennej y, tak więc w kolejnym punkcie będę operował zmienną y jako wartością zanegowaną.

y	1	1	0	0	0	0	0	1	1	0
y = ~y	0	0	1	1	1	1	1	0	0	1
Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 3

W tym kroku zasłaniamy wartości, które nie są nam potrzebne (potrzebne jest tylko pięć bitów licząc od prawej strony. Bity: 0, 1, 2, 3, 4). Do tego celu używamy bitowej koniunkcji.

0	0	0	0	0	0	0	0	0	0	0
~0	1	1	1	1	1	1	1	1	1	1
~0<<n	1	1	1	1	1	0	0	0	0	0
~(~0<<n)	0	0	0	0	0	1	1	1	1	1
y	0	0	1	1	1	1	1	0	0	1
y &= ~(~0<<n)	0	0	0	0	0	1	1	0	0	1

W pierwszej kolumnie wpisaliśmy same zera, w drugiej je zanegowaliśmy. Otrzymane jedynki przesuneliśmy o n (w naszym przypadku 5) pozycji (wiersz trzeci). W wierszu czwartym negujemy to co otrzymaliśmy w wierszu poprzednim. W tym miejscu otrzymaliśmy maskę, która zasłoni nam nie potrzebne bity, a zostawi tylko pięc bitów z prawej strony. Przed ostatni wiersz to y (zanegowana wartość z poprzedniego kroku). W ostatnim wierszu wpisujemy wartość porównania wiersza przed ostatniego z maską (za pomocą bramki „i”). Całość zapisujemy w y.

Krok 4

W tym kroku przesuwamy wartość y o pewną ilość miejsc. O tym ile tych miejsc jest decyduje wzór: p + 1 – n. Czyli w naszym przypadku y przesuwamy o 3 miejsca.

y	0	0	0	0	0	1	1	0	0	1
y << p + 1 - n	0	0	1	1	0	0	1	0	0	0

W tym miejscu chce powiedzieć dlaczego na początku (krok 2) zanegowaliśmy naszą wartość y. Jest to związane bezpośrednio z tym krokiem, a mianowicie z przesunięciem wartości y o wyliczoną na podstawie wyżej wymienionego wzoru ilość pozycji. Po lewej stronie mamy same zera (względem kolorowego tła), po prawej trzy zera a nasza wartość w kolorowym tle jest negacją wartości, którą

musimy wstawić w wyznaczone miejsce. Kolejną i przedostatnią rzeczą jaką musimy zrobić to zanegować wartość y, by otrzymać te wartości, które chcemy i zamiast zer jedynki, by przy ostatniej czynności – porównaniu parami bitów (za pomocą bramki „i”) nie usunąć żadnego innego bitu.

Krok 5

Negacja wartości znajdującej się pod y oraz przypisanej tej zanegowanej wartości do zmiennej y.

y	0	0	1	1	0	0	1	0	0	0
y = ~y	1	1	0	0	1	1	0	1	1	1

Krok 6

Porównanie wartości zanegowanej z liczbą kryjącą się pod x.

x	1	1	1	1	1	1	1	1	1	1
y	1	1	0	0	1	1	0	1	1	1
x & y	1	1	0	0	1	1	0	1	1	1
Nr bitu	9	8	7	6	5	4	3	2	1	0

A więc w tych paru krokach wstawiliśmy pięc początkowych bitów zmiennej y do zmiennej x na pozycjach bitów od 3 do 7. Listing tego programu prezentuje się następująco (dużo krótszy niż mogło by się wydawać). Została napisana funkcja, która jest wywoływana z funkcji main.

```
#include <stdio.h>

unsigned setbits (unsigned x, int p, int n, unsigned y);

main()
{
      unsigned x = 1023, y = 774;
      int p = 7, n = 5;

      printf("%u\n", setbits(x, p, n, y));  // 823
}

unsigned setbits (unsigned x, int p, int n, unsigned y)
{
      y = ~y;                  // Krok 2
      y &= ~(~0 << n);         // Krok 3
      y <<= p + 1 - n;         // Krok 4
      y = ~y;                  // Krok 5
```

```
    return x & y;          // Krok 6
}
```

Listing 2.3.10 Funkcja setbits

Zadanie 2.7

Napisz funkcję invert(x, p, n) zwracającą wartość x, w której n bitów – poczynając od pozycji p – zamieniono z 1 na 0 i odwrotnie. Pozostałe bity x nie powinny ulec zmianie.

Odpowiedź:

Tak jak w poprzednim przykładzie, podzielimy wykonanie naszego zadania na kilka kroków, w których będziemy rysować tabelki dla pełnego zrozumienia postawionego nam problemu. Rozwiązanie tego zadania może być trochę dłuższe niż poprzedniego, nie mniej jednak zasady postępowania są podobne.

Krok 1

W tym kroku przyjmujemy jakąś liczbę za x, niech tą liczbą będzie 621, której reprezentacją binarną jest 1001101101. Wybieramy ilość bitów, które chcemy zanegować, oraz pozycję od której tę ilość bitów będziemy liczyć. Odpowiednio zmienna n, oraz p. Załóżmy niech n będzie 3, a p równa się 6. Czyli musimy zanegować bity: 6, 5, 4.

Przed	1	0	0	1	1	0	1	1	0	1
Po	1	0	0	0	0	1	1	1	0	1
Nr bitu	9	8	7	6	5	4	3	2	1	0

Wartość **Po** dziesiętnie to 541 i właśnie takiego wyniku się spodziewamy.

Krok 2

W tym kroku przypisujemy do zmiennej pomocniczej x1 przesuniętą wartość x o p + 1 – n pozycji w prawo.

x	1	0	0	1	1	0	1	1	0	1
x1 = x >> p+1-n	0	0	0	0	1	0	0	1	1	0
Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 3

W kroku trzecim negujemy wartość zmiennej x1, i przypisujemy ją do x1.

x1	0	0	0	0	1	0	0	1	1	0
x1 = ~x1	1	1	1	1	0	1	1	0	0	1
Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 4

W kroku czwartym tworzymy maskę (tak jak w poprzednim zadaniu), dzięki której zasłonimy nie potrzebne bity, a zostawimy tylko te na których nam zależy, czyli bity: 0, 1, 2. Wynik całej operacji przypisujemy do zmiennej pomocniczej x1.

0	0	0	0	0	0	0	0	0	0	0
~0	1	1	1	1	1	1	1	1	1	1
~0 << n	1	1	1	1	1	1	1	0	0	0
~(~0 << n)	0	0	0	0	0	0	0	1	1	1
x1	1	1	1	1	0	1	1	0	0	1
x1 & ~(~0 << n)	0	0	0	0	0	0	0	0	0	1
Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 5

W kroku piątym przesuwamy bity zmiennej x1 o p + 1 – n pozycji w lewo.

x1	0	0	0	0	0	0	0	0	0	1
x1 << p+1–n	0	0	0	0	0	1	0	0	0	0
Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 6

W kroku tym tworzymy pomocniczą zmienną z, która będzie służyła do wyzerowania pewnej ilości bitów zmiennej x, którą następnie uzupełnimy bitami przygotowanymi w kroku 5. Wartość przed ostatniego wiersza przypisujemy do zmiennej z.

~0	1	1	1	1	1	1	1	1	1	1
~0 << n	1	1	1	1	1	1	1	0	0	0
z = ~(~0 << n)	0	0	0	0	0	0	0	1	1	1

Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 7

W kroku tym przesuwamy wartość z o p + 1 – n pozycji w lewo.

z	0	0	0	0	0	0	0	1	1	1
z << p+1-n	0	0	0	1	1	1	0	0	0	0
Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 8

W kroku tym negujemy wartość zmiennej z.

z	0	0	0	1	1	1	0	0	0	0
z = ~z	1	1	1	0	0	0	1	1	1	1
Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 9

W tym kroku zerujemy wartość bitów, na które mamy ustawić pewne bity (założenie zadania) za pomocą zmiennej z. Widać w tabelce z kroku 8, że pewne bity są zerami, więc jak porównamy parami (bitowa koniunkcja) zmienną z wraz ze zmienną x, to wyzerujemy pewne bity. Wartość tę przypisujemy do zmiennej pomocniczej g (wiersz trzeci).

x	1	0	0	1	1	0	1	1	0	1
z	1	1	1	0	0	0	1	1	1	1
g = x & z	1	0	0	0	0	0	1	1	0	1
Nr bitu	9	8	7	6	5	4	3	2	1	0

Krok 10

Teraz zostało już tylko ustawić bity w miejsce, gdzie zostały one wyzerowane. W kroku piątym przygotowaliśmy specjalnie do tego celu tę liczbę. Teraz za pomocą bitowej alternatywy ustawimy te bity.

x1	0	0	0	0	0	1	0	0	0	0
g	1	0	0	0	0	0	1	1	0	1
x1 | g	1	0	0	0	0	1	1	1	0	1

Nr bitu	9	8	7	6	5	4	3	2	1	0

Tak o to otrzymalismy liczbę 1000011101, której reprezentacją dziesiętną jest liczba 541. Poniższy listing pokazuje jak wyglądają te operację w języku C.

```
#include <stdio.h>

unsigned invert (unsigned x, int p, int n);

main ()
{
	unsigned x = 621;
	int p = 6;
	int n = 3;

	printf("%u\n", invert(x, p, n));
}
unsigned invert (unsigned x, int p, int n)
{
	unsigned x1, z, g;

	x1 = x >> p + 1 - n;		// Krok 2
	x1 = ~x1;			// Krok 3
	x1 &= ~(~0 << n);		// Krok 4
	x1 <<= p + 1 - n;		// Krok 5
	z = ~(~0 << n);			// Krok 6
	z <<= p + 1 - n;		// Krok 7
	z = ~z;				// Krok 8
	g = x & z;			// Krok 9
	return x1 | g;			// Krok 10
}
```

Listing 2.3.11 Funkcja invert.

2.3.5 Priorytety

Poniżej znajduje się tabela z priorytetami operatorów. Nie wszystkie jeszcze zostały omówione, lecz zostaną w kolejnych rozdziałach. Najwyżej w tabeli znajdują się operatory z najwyższym priorytetem. Kolejne wiersze mają coraz niższe priorytety. Operatory w jednym wierszu mają jednakowy priorytet.

Operatory	Łączność
() [] -> .	Lewostronna
! ~ ++ -- + - * & (typ) sizeof	Prawostronna

* / %	Lewostronna
+ -	Lewostronna
<< >>	Lewostronna
< <= > >=	Lewostronna
== !=	Lewostronna
&	Lewostronna
^	Lewostronna
|	Lewostronna
&&	Lewostronna
||	Lewostronna
?:	Prawostronna
= += -= *= /= %= ^= |= <<= >>=	Prawostronna
,	Lewostronna

Tabela 2.3.8 Tabela priorytetów

2.3.6 Funkcje matematyczne

Wszystkie opisane w tym rozdziale funkcje korzystają z pliku nagłówkowego math.h, który musi zostać dołączony, aby funkcje te były rozpoznawane. Poniższy zapis:

```
double sin (double);
```

Oznacza po kolei:

- double – Typ zwracanej wartości – double
- sin – nazwa funkcji
- (double) – Ilość (jeśli więcej niż jeden, to typy wypisane są po przecinku) i typ argumentów.

Wszystkie funkcje opisane w tym rozdziale zwracają wartość typu double. Argumenty x, y są typu double, a argument n typu int. Kompilacja odbywa się prawie tak samo jak w poprzednich przykładach, nie mniej jednak dołączamy przełącznik (opcję) **-lm**. A więc polecenie będzie wyglądać następująco:

```
$ gcc plik.c -o plik -lm
```

Jeszcze jedna uwaga. Wartości kątów funkcji trygonometryczny wyraża się w radianach. W poniższej tabeli znajduje się zestawienie funkcji matematycznych.

Nazwa funkcji	Deklaracja	Dodatkowe informacje
Sinus	sin(x)	-
Cosinus	cos(x)	-
Tangens	tan(x)	-
Arcus sinus	asin(x)	$y \in \langle \frac{-\pi}{2} ; \frac{\pi}{2} \rangle , x \in \langle -1 ; 1 \rangle$ [1]
Arcus cosinus	acos(x)	$y \in \langle 0 ; \pi \rangle , x \in \langle -1 ; 1 \rangle$
Arcus tangens	atan(x)	$y \in \langle \frac{-\pi}{2} ; \frac{\pi}{2} \rangle$
Arcus tangens	atan2(y,x)	$\tan^{(-1)}(\frac{y}{x}) \in \langle -\pi ; \pi \rangle$
Sinus hiperboliczny	sinh(x)	-
Cosinus hiperboliczny	cosh(x)	-
Tangens hiperboliczny	tanh(x)	-
Funkcja wykładnicza	exp(x)	e^x
Logarytm naturalny	log(x)	$\ln(x), x > 0$
Logarytm o podstawie 10	log10(x)	$\log_{10}(x), x > 0$
Potęgowanie[2]	pow(x,y)	x^y
Pierwiastkowanie	sqrt(x)	$\sqrt{x}, x \geqslant 0$
Najmniejsza liczba całkowita	ceil(x)	Nie mniejsza niż x – wynik typu double
Największa liczba całkowita	floor(x)	Nie większa niż x – wynik typu double
Wartość bezwzględna	fabs(x)	$\lvert x \rvert$
-	ldexp(x,n)	$x \cdot 2^n$
-	frexp(x, int *exp)	Rozdziela x na znormalizowaną część ułamkową z przedziału $\langle 0.5 ; 1 \rangle$ i wykładnik potęgi 2. Funkcja zwraca część ułamkową, a wykładnik potęgi wstawia do *exp. Jeśli x = 0, to obie części wyniku równają się 0.

1 y – Mam na myśli y jako wartości funkcji, nie argument, który występuje w innych funkcjach
2 Błąd zakresu wystąpi gdy: x = 0 i y <= 0 lub x < 0 i y nie jest całkowite

-	modf(x, double *u)	Rozdziela x na część całkowitą i ułamkową, obie z takim samym znakiem co x. Część całkowitą wstawia do *u i zwraca część ułamkową
-	fmod(x,y)	Zmiennopozycyjna reszta z dzielenie x/y z tym samym znakiem co x;

Tabela 2.3.9 Funkcje z biblioteki math.h

Przykład

Oblicz wyrażenie podane poniżej oraz wyświetl dodatkowe dwa wyniki: zaokrąglone w dól, oraz w górę do liczb całkowitych.

$$y=\frac{\frac{1}{2}\cdot\sin^2(0.45)+2\cdot tg(\sqrt{2})}{\log_{10}(14)+2\cdot e^4}$$

Odpowiedź

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

main ()
{
	double y, licznik, mianownik;
	double yGora, yDol;
	double p1, p2, p3, p4;			// Zmienne pomocnicze

	p1 = pow(sin(0.45), 2);
	p2 = tan(sqrt(2));
	p3 = log10(14);
	p4 = exp(4);

	licznik = (1.0/2)*p1 + 2*p2;
	mianownik = p3 + 2*p4;

	y = licznik / mianownik;
	yGora = ceil(y);
	yDol = floor(y);

	printf("y = \t\t\t%f\n", y);
	printf("Zaokraglone w gore: \t%f\n", yGora);
	printf("Zaokraglone w dol: \t%f\n", yDol);
}
```

Listing 2.3.12 Odpowiedź do zadania

Zdecydowanie łatwiej utworzyć zmienne pomocnicze, w których wpiszemy pojedyncze operacje, aniżeli wpisywać całą formułę do zmiennej `y`. Jestem prawie pewien, że w pewnym momencie pogubiłbyś się z nawiasami. Oczywiście sposób ten nie jest z góry narzucony, ma za zadanie ułatwić napisanie tego kodu.

3 Sterowanie programem

Przez sterowanie programem rozumie się zbiór instrukcji, które potrafią wybierać w zależności od danego im parametru czy mają zrobić to, czy coś innego. W przykładzie z listingu 2.3.1 pokazana została instrukcja warunkowa if–else, która w tym rozdziale zostanie omówiona bardziej szczegółowo.

3.1 Instrukcja if – else

Pomimo iż wyrażenia warunkowe były używane w poprzednim rozdziale, to pozwolę sobie opisać tutaj dokładniej zasadę ich działania, oraz pokazać drugi sposób sprawdzania warunku. A więc zacznijmy od składni instrukcji if–else która wygląda następująco:

```
if (wyrazenie)
      akcje1
else
      akcje2
```

Gdzie jako wyrazenie można wstawić dowolny warunek z użyciem operatorów logicznych, przyrównania oraz relacji. A jako akcje dowolne instrukcje wykonywane przez program (przypisanie wartości zmiennym, wyświetlenie komunikatu, itp). Dla przykładu weźmy trywialny przykład, który sprawdza czy liczba a jest większa lub równa liczbie b i drukuje odpowiedni komunikat. Listing 3.1.1 prezentuje kod programu.

```
#include <stdio.h>

main ()
{
      int a = 4;
      int b = 7;

      if (a >= b)
            printf("a wieksze lub rowne b\n");
      else
            printf("a mniejsze lub rowne b\n");
}
```

Listing 3.1.1 Użycie instrukcji if – else.

Naszym wyrażeniem warunkowym na listingu 3.1.1 jest: a >= b. Jeśli jest to prawda, czyli wartość zmiennej a jest większa lub równa wartości zmiennej b, to to wyrażenie przyjmuje wartość jeden, w przeciwnym wypadku zero. Co to znaczy? Znaczy to tyle, iż gdybyśmy przypisali to wyrażenie do zmiennej pomocniczej, np. w ten sposób:

```
int z = a >= b;
```

to zmienna z posiadałaby wartość jeden (jeśli warunek spełniony) lub zero (jeśli warunek nie spełniony). W ten sposób możemy wstawić zmienną pomocniczą do instrukcji if–else w ten sposób:

```
if (z)
```

if (1) jest prawdziwe[1], czyli wykonaja się akcje1, czyli napis: „a większe lub równe b" zostanie wydrukowany. If (0) jest fałszywe, czyli nie wykonają się akcje1, tylko te akcje, które są po słowie kluczowym else (jeśli istnieje – o tym za chwilę), czyli akcje2.

Jeśli chcemy aby wykonało się więcej instrukcji niż jedna (np. wyświetlenie tekstu i przypisanie zmiennej pewnej wartości – kod poniżej), jesteśmy zmuszeni użyć nawiasów klamrowych po słowie kluczowym if i/lub else.

```
if (wyrazenie)
{
      printf("Dowolny napis\n");
      i++;
}
else
{
      printf("Dowolny napis\n");
      i--;
}
```

Można w ogóle nie używać słowa kluczowego else, w razie nie spełnionego warunku, nic się nie stanie, program przechodzi do wykonywania kolejnych instrukcji.

Jeśli zagnieżdżamy instrukcje if–else, to musimy pamiętać o bardzo istotnej rzeczy, a mianowicie instrukcja else zawsze należy do ostatniej instrukcji if. Czyli w podanym niżej przykładzie instrukcja else wykona się wtedy, kiedy zmienna n będzie większa od 2 oraz nie parzysta.

1 if (x) jest prawdziwe dla wszystkich wartości x, z wyjątkiem zera.

```
#include <stdio.h>

main ()
{
	int n = 3;

	if (n > 2)
		if (n % 2 == 0)
			printf("%d jest parzysta\n", n);
		else
			printf("%d jest nie parzysta\n", n);
}
```

Listing 3.1.2 Zagnieżdżone instrukcje if – else.

Jeśli chcielibyśmy, aby zagnieżdzona instrukcja if nie posiadała części else, a za to „główny” if takową posiadał musimy użyć nawiasów klamrowych. Listing poniżej pokazuje jak to zrobić.

```
#include <stdio.h>

main ()
{
	int n = 2;

	if (n > 2)
	{
		if (n % 2 == 0)
			printf("%d jest parzysta\n", n);
	}
	else
		printf("Liczba: %d jest mniejsza lub rowna 2\n", n);
}
```

Listing 3.1.3 Zagnieżdżone instrukcje if – else kontynuacja

Istnieje trzy argumentowy operator ?:, którego postać wygląda następująco:

```
wyrazenie1 ? wyrazenie2 : wyrazenie3
```

Operator ten działa w sposób następujący, obliczane jest wyrazenie1 i jeśli jest prawdziwe (różne od zera), to obliczane jest wyrazenie2 i ono staje się wartością całego wyrażenia. W przeciwnym wypadku obliczane jest wyrazenie3 i ono jest wartością całego wyrażenia. Poniższy listing demonstruje działanie operatora ?:.

```c
#include <stdio.h>

main ()
{
	int a = 3, z;

	z = (a > 5) ? 5 : (a + 6);
	printf("%d\n", z);		// 9
}
```

Listing 3.1.4 Operator trzyargumentowy „?:”

W sterowaniu programem występuje jeszcze jedna konstrukcja związana z if-else, a mianowicie jest to else-if. Poniżej znajduje się deklaracja tejże konstrukcji, a pod nią opis wraz z przykładem.

```
if (wyrazenie1)
	akcje1
else if (wyrazenie2)
	akcje2
else
	akcje3
```

Jak widać instrukcja ta jest rozbudowaną instrukcją if–else, zasada działania tej instrukcji jest niemalże identyczna. Najpierw sprawdzane jest wyrazenie1, jeśli jest fałszywe, to program ma dodatkowe wyrażenie do sprawdzenia – wyrazenie2, jeśli jest prawdziwe to wykonują się akcje2, w przeciwnym wypadku wykonują się akcje3. Chodzi o to, że w jednej instrukcji if–else może znajdować się kilka warunków do sprawdzenia, tak więc konstrukcji else–if można dopisać tyle, ile się ich potrzebuje. Jeśli program ustali prawdziwość któregokolwiek z wyrażeń to zostają wykonane instrukcje związane z tym wyrażeniem i tylko te instrukcje, reszta jest pomijana. Poniżej znajduje się przykład z użyciem konstrukcji else–if.

```c
#include <stdio.h>
main ()
{
	int n = 24;
	if (n >= 0 && n <= 10)
		printf("%d zawiera sie w przedziale: <%d; %d>\n", n, 0, 10);
	else if (n >= 11 && n <= 20)
		printf("%d zawiera sie w przedziale: <%d; %d>\n", n, 11, 20);
	else if (n >= 21 && n <= 30)
		printf("%d zawiera sie w przedziale: <%d; %d>\n", n, 21, 30);
```

```
	else if (n >= 31)
		printf("Liczba %d jest wieksza lub rowna 31\n", n);
	else
		printf("Liczba jest ujemna\n");
}
```

Listing 3.1.5 Użycie konstrukcji else – if

Jak widać pierwszy warunek jest fałszywy, drugi również. Trzeci warunek jest prawdziwy tak więc zostanie wykonana instrukcja wyświetlająca napis, że liczba zawiera się w przedziale <21; 30> i tylko ta instrukcja zostanie wykonana, reszta warunków nie jest sprawdzana.

3.2 Instrukcja switch

Instrukcja switch również służy do podejmowania decyzji. Można by powiedzieć, że jest podobna trochę do konstrukcji else–if, lecz sposób jej definicji trochę się różni. Poniżej znajduje się ogólna postać omawianej instrukcji.

```
switch (wyrazenie)
{
	case wyrazenie_stale1: akcje1;
	case wyrazenie_stale2: akcje2;
	default: akcje3;
}
```

Na poniższym przykładzie pokażę zastosowanie i opiszę działanie danej instrukcji.

```
#include <stdio.h>

main ()
{
	int n = 3;

	switch (n)
	{
		case 1:	printf("Poniedzialek\n");
				break;
		case 2:	printf("Wtorek\n");
				break;
		case 3:	printf("Sroda\n");
				break;
```

```
            case 4:    printf("Czwartek\n");
                       break;
            case 5:    printf("Piatek\n");
                       break;
            case 6:    printf("Sobota\n");
                       break;
            case 7:    printf("Niedziela\n");
                       break;
            default:   printf("Nie ma takiego dnia\n");
                       break;
      }
}
```

Listing 3.2.1 Użycie instrukcji switch

Zasada działania instrukcji switch jest następująca. Po słowie kluczowym switch występuje argument w nawiasie, który przyjmuje dowolną wartość. Pomiędzy klamrami znajdują się pewne przypadki (case – wyrażenia stałe, nie mogą zależeć od zmiennych). Po otrzymaniu argumentu instrukcja sprawdza, czy przekazana wartość jest równa wartości jednego z wymienionych przypadków i jeśli tak jest to wykonuje akcje, które są zdefiniowane po dwukropku. Instrukcja break po każdej akcji jest bardzo istotna, ponieważ przerywa działanie instrukcji switch, po wykonaniu zaplanowanej akcji. Gdybyśmy nie użyli break, to instrukcja poniżej zostałaby wykonana, a po niej kolejna i jeszcze następne, aż do zamykającego instrukcję switch nawiasu klamrowego.

Przypadek default oznacza, że żaden z przypadków nie mógł zostać spełniony (argument miał inną wartość, niż te zdefiniowane przez case). Przypadek default może zostać pominięty i w razie, gdy argument będzie się różnił od wymienionych przypadków, żadna akcja nie zostanie podjęta. W rozdziale 2.2.6 (Stała wyliczenia) powiedziane było, że zastosowanie stałych wyliczenia będzie pokazane w tym rozdziale, tak więc listing 3.2.2 pokazuje połączenie instrukcji switch i enum.

```
#include <stdio.h>

main()
{
      enum tydzien {PON = 1, WT, SR, CZ, PT, SOB, ND};
      int n = 10;

      switch (n)
      {
            case PON:  printf("Poniedzialek\n");
                       break;
            case WT:   printf("Wtorek\n");
                       break;
```

```
            case SR:   printf("Sroda\n");
                       break;
            case CZ:   printf("Czwartek\n");
                       break;
            default:   printf("Nie ma takiego dnia\n");
                       break;
            case PT:   printf("Piatek\n");
                       break;
            case SOB:  printf("Sobota\n");
                       break;
            case ND:   printf("Niedziela\n");
                       break;
      }
}
```

Listing 3.2.2 Użycie instrukcji switch oraz enum

Zasada działania jest analogiczna, z tą różnicą, że zamiast cyfr wpisaliśmy stałe wyliczenia, które zdefiniowaliśmy w piątej linii za pomocą enum, który został opisany w punkcie 3.2.2. Widać również, że przypadek default nie musi być ostatni, kolejność deklaracji jest dowolna. Pomimo iż nie trzeba po ostatnim przypadku pisać break, to lepiej to zrobić. Po dodaniu kolejnego przypadku można zapomnieć o tym, a późniejsze znalezienie błędu może zająć trochę czasu.

Innym również przydatnym aspektem może okazać się możliwość wpisywania pewnego zakresu podczas definicji przypadku. Listing 3.2.3 pokazuje to zachowanie.

```
#include <stdio.h>

main()
{
      int n = 31;

      switch (n)
      {
            case 0 ... 10 :   printf("%d zawiera sie w przedziale: ", n);
                        printf("<%d; %d>\n", 0, 10);
                        break;
            case 11 ... 20 :  printf("%d zawiera sie w przedziale: ", n);
                        printf("<%d; %d>\n", 11, 20);
                        break;
            case 21 ... 30 :  printf("%d zawiera sie w przedziale: ", n);
                        printf("<%d; %d>\n", 21, 30);
                        break;
            default :  if (n >= 31)
                       printf("Liczba %d jest wieksza lub rowna 31\n", n);
                     else
                       printf("Liczba jest ujemna\n");
```

```
                        break;
        }
}
```
Listing 3.2.3 Użycie instrukcji switch wraz z zakresem w przypadku

Jak widać program działa analogicznie do tego z listingu 3.1.5. Trzeba było i tak użyć instrukcji if, ponieważ zakresy muszą być z góry ustalone. Więc w przypadku default sprawdzamy, czy liczba jest większa lub równa 31, jeśli to fałsz to liczba jest ujemna.

Ważne jest to, aby trzy kropki były oddzielone spacją między granicami przedziałów. Jeśli by nie były – program się nie skompiluje.

3.3 Pętle

Zauważmy, że jeśli potrzebujemy wykonać pewną ilość operacji, która w gruncie rzeczy różni się tylko argumentem to nie potrzebujemy wpisywać wszystkich tych instrukcji w kodzie programu. Wystarczy użyć jednej z dostępnych metod zapętlenia fragmentu kodu i tam wpisać daną instrukcję.

Przykłady użycia petli zostały pokazane w kolejnych podpunktach tego rozdziału które odnoszą się do poszczególnych rodzajów pętli. Za pomocą każdej pętli można wykonać dokładnie to samo. Więc dlaczego jest ich tyle? Ponieważ za pomocą nie których konstrukcji łatwiej jest zrealizować pewne zadanie.

W przykładach tych użyto dotąd nie poznanej instrukcji scanf, która pobiera wartości od użytkownika, zostanie ona omówiona bardziej szczegółowo w rozdziale 8. Operacje wejścia i wyjścia. W każdej z tych instrukcji można użyć nawiasów klamrowych w celu zgrupowania większej ilości akcji.

3.3.1 for

Konstrukcja pętli for wygląda następująco:

```
for (wyrazenie1; wyrazenie2; wyrazenie3)
      akcje
```

Poniższy kod prezentuje użycie pętli for. W programie tym użytkownik będzie musiał podać dolną oraz górną granicę, a program obliczy ile jest między nimi liczb parzystych oraz wyświetli te liczby.

```
#include <stdio.h>

main ()
{
	int dolnaGranica, gornaGranica;
	int i, ilParz = 0;

	printf("Dolna granica: ");
	scanf("%d", &dolnaGranica);
	printf("Gorna granica: ");
	scanf("%d", &gornaGranica);

	printf("Pomiedzy zakresem: <%d; %d> ", dolnaGranica, gornaGranica);
	printf("liczbami parzystymi sa: ");

	for (i = dolnaGranica; i <= gornaGranica; i++)
		if (i % 2 == 0)
		{
			ilParz++;
			printf("%d ", i);
		}

	printf("\nOgolna ilosc liczb parzystych z tego zakresy to: %d\n", 
ilParz);
}
```

Listing 3.3.1.1 Użycie pętli for do obliczenia ilości parzystych liczb

Definiujemy zmienne, które będą przechowywać dolną oraz górną granicę. Po nazwach oczywiście widać, które to. Zmienna i jest zmienną sterującą pętlą. Zmienna ilParz przechowuje ilość parzystych liczb, musimy wyzerować jej wartość przed użyciem, ponieważ gdybyśmy tego nie zrobili, to pewnie dostalibyśmy inny wynik niż oczekiwaliśmy. Dzieje się tak, ponieważ w zmiennych mogą występować śmieci (omówione w rozdziale 2.2.3)

Funkcja scanf zostanie omówiona dokładniej w rozdziale dotyczącym operacji wejścia i wyjścia. Tutaj natomiast musimy wiedzieć, że za pomocą tej funkcji możemy pobrać dane od użytkownika (bez użycia zabezpieczeń funkcja ta jest mało praktyczna, bo gdy wpiszemy literę zamiast liczby program się „wykrzacza”).

Działanie pętli for powinniśmy rozumieć następująco:

- i = dolnaGranica – Przypisanie do zmiennej sterującej pętlą wartości, którą wpisał użytkownik
- i <= gornaGranica – Warunek sprawdzany przed wykonaniem każdej iteracji

- i++ – zwiększenie zmiennej sterującej

Gdy program wchodzi do tej pętli to pierwszą czynnością jaką robi jest przypisanie wartości, którą wpisał użytkownik jako dolną granicę do zmiennej sterującej. Następnie sprawdzany jest warunek, czy i jest mniejsze lub równe od górnej granicy, jeśli tak jest, to zostaną wykonane akcje (tutaj instrukcja if). Jeśli akcji jest więcej, to podobnie jak w przypadku instrukcji if zamyka się je w nawiasy klamrowe. Po wykonaniu akcji zwiększany jest licznik za pomocą instrukcji i++ i po raz kolejny sprawdzany jest warunek, jeśli tym razem warunek jest fałszywy, to pętla kończy swoje działanie.

W pętli for znajduje się instrukcja if, która sprawdza czy aktualna wartość kryjąca się pod zmienną sterującą i dzieli się przez dwa bez reszty, jeśli tak jest, to liczba ta jest parzysta. Zmienna ilParz zostaje zwiększona o jeden oraz parzysta liczba jest wyświetlana na ekran monitora.

Każde wyrażenie instrukcji for można pominąć. Jeśli pominiemy warunek, to będzie to traktowane tak jakby zawsze był spełniony. Aby uzyskać pętlę nieskończoną, możemy napisać ją w ten sposób.

```
for (;;)
{
...
}
```

Pętlę taką można przerwać za pomocą instrukcji break, oraz return.
Jeśli pomijamy pierwszy składnik, czyli ustawienie wartości zmiennej sterującej to wypadało by, żeby zmienna ta była ustawiona tuż przed pętlą ręcznie, bądź miała wartość oczekiwaną (jak w naszym przypadku – podanej przez użytkownika). Zwiększenie licznika może być częścią wykonywanych akcji, więc przykładowa deklaracja instrukcji for poniżej pokazuje analogiczne skutki jak listing 3.3.1.1.

```
i = dolnaGranica;
for (; i <= gornaGranica;)
{
	// w tym miejscu instrukcja if {...}
	i++;
}
```

Pętle można zagnieżdżać. Zagnieżdżone pętle używa się do operacji na tablicach wielowymiarowych (w rozdziale 5. znajdują się informacje o tablicach). Przykład zagnieżdżonych pętli for znajduje się poniżej. Jest to tabliczka mnożenia.

```
#include <stdio.h>

main ()
{
      int i, j;

      for (i = 0; i <= 10; i++)
            for (j = 0; j <= 10; j++)
                  if (!i && !j)
                        printf("*\t");
                  else if (!i && j)
                        printf("%d%c", j, (j == 10) ? '\n' : '\t');
                  else if (i && !j)
                        printf("%d%c", i, (j == 10) ? '\n' : '\t');
                  else
                        printf("%d%c", i * j, (j == 10) ? '\n' : '\t');

}
```

Listing 3.3.1.2 Zagnieżdżone pętle

Przykład może i wygląda strasznie, lecz postaram się go omówić najbardziej szczegółowo jak potrafię. Poniżej znajduje się tabela 3.3.1, w której de facto jest tabliczka mnożenia, ale jest również dodatkowy wiersz (pierwszy), oraz dodatkowa kolumna (pierwsza) w której są napisane numery iteracji, czyli jakie wartości będą posiadać zmienne sterujące i, oraz k. Dla lepszej czytelności tabliczka zaczyna się od następnej strony.

Więc zaczynamy, pierwsza pętla (zewnętrzna), w której zmienną sterującą jest i odpowiada za operacje na wierszach, wewnętrzna pętla (zmienna sterująca j) odpowiada za manipulowanie kolumnami. Tak więc najpierw zmienna i przyjmuje wartość zero (wiersz zerowy[1] - żółte komórki), później wykona się 11 razy pętla wewnętrzna (kolumny j = 0, …, j = 10). Jeśli j osiągnie wartość jedenaście to pętla jest przerywana i zwiększany jest licznik zmiennej i. Po raz kolejny wykonuje się pętla wewnętrza. Operacje te powtarzają się do czasu, gdy warunek pętli zewnętrznej jest prawdziwy.

1 Przez wiersz zerowy rozumiemy ten, w którym i = 0.

	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	j = 10
i = 0	*	1	2	3	4	5	6	7	8	9	10
i = 1	1	1	2	3	4	5	6	7	8	9	10
i = 2	2	2	4	6	8	10	12	14	16	18	20
i = 3	3	3	6	9	12	15	18	21	24	27	30
i = 4	4	4	8	12	16	20	24	28	32	36	40
i = 5	5	5	10	15	20	25	30	35	40	45	50
i = 6	6	6	12	18	24	30	36	42	48	54	60
i = 7	7	7	14	21	28	35	42	49	56	63	70
i = 8	8	8	16	24	32	40	48	56	64	72	80
i = 9	9	9	18	27	36	45	54	63	72	81	90
i = 10	10	10	20	30	40	50	60	70	80	90	100

Tabela 3.1.1 Tabliczka mnożenia wraz z pomocniczymi informacjami

Warunki, które są w pętli wewnętrznej sterują procesem wypełniania informacji w tabeli. Pierwszy warunek if (!i && !j) jest spełniony „Jeśli i jest zerem i jednocześnie j jest zerem”. Jeśli ten warunek jest spełniony, a spełniony jest tylko raz, to wstaw gwiazdkę (zielona komórka). Użyta konstrukcja else–if pozwala na sprawdzenie dodatkowych warunków, a mianowicie else if (!i && j), który mówi tak: „Jeśli i jest zerem i j jest różne od zera”, co dokładnie daje nam wiersz zerowy (żółty wiersz). Jeśli ten warunek jest spełniony, to uzupełnij go wartościami zmiennej j. Tutaj, jak i w kolejnych dwóch funkcjach printf użyto operatora trójargumentowego, żeby zrobić nową linię po dziesiątej kolumnie, a pomiędzy kolumnami wstawić tabulację.

Kolejny warunek else if (i && !j) tłumaczy się w ten sposób: „Jeśli i jest różne od zera i j jest zerem” co jest odzwierciedleniem zerowej kolumny (niebieska kolumna). Z kolei jeśli ten warunek jest spełniony to uzupełnij go wartościami zmiennej i. Jeśli żaden z powyższych warunków nie został spełniony to uzupełnij tabliczkę mnożenia (uzupełniamy wierszami) – wartościami i * j.

3.3.2 while

Pętla while jest w niektórych sytuacjach lepszym rozwiązaniem, niż pętla for, np. wtedy, kiedy nie znamy z góry ilości wykonań pętli. Konstrukcja pętli while wygląda następująco.

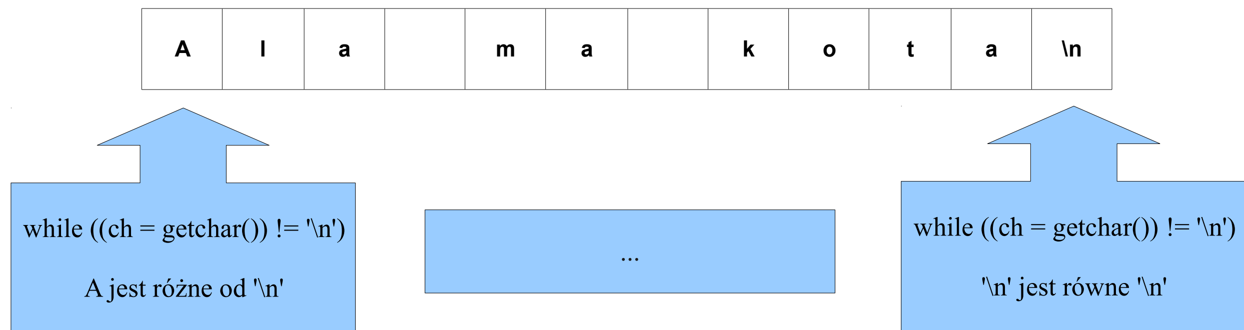
```
while (warunek)

      akcje
```

I działa w sposób następujący, jeśli warunek zostanie spełniony, to akcje zostaną wykonane. W przeciwnym wypadku akcje się nie wykonują. Dla przykładu zaprezentowano poniższy listing, w którym użytkownik wpisuje znaki, a program później wyświetla je pojedyńczo, każdy w nowej linii. Przykład może nie jest zbyt ambitny, lecz pokazuje, że do tego typu zadań lepiej nadaje się pętla while, niż for.

```
#include <stdio.h>

main ()
{
      int ch;

      while ((ch = getchar()) != '\n')
            printf("Znak: %c\n", ch);
}
```

Listing 3.3.2.1 Użycie pętli while

Jeszcze wyjaśnienie odnośnie dotąd nie używanej funkcji getchar. Funkcja ta pobiera znak od użytkownika, jeśli znaków jest więcej, a z reguły jest, ponieważ enter (znak nowej linii) też jest znakiem – znaki te przechowywane są w obszarze pamięci zwanym buforem. Aby wyświetlić wszystkie znaki z bufora, można użyć pętli while, która tak zadeklarowana jak na listingu 3.3.2.1 skończy swoje działanie dopiero wtedy, gdy natrafi na znak nowej linii. Poniższy obrazek może ułatwić zrozumienie tego. Niech wprowadzonymi znakami będą „Ala ma kota”.



Rys. 3.3.2.1 Zastosowanie pętli while

Działanie pętli jest następujące, sprawdza po kolei znaki będące w buforze i wyświetla je, jeśli są różne od znaku nowej linii (\n).

3.3.3 do – while

Pętla do–while działa podobnie do pętli while z tą różnicą, że warunek sprawdzany jest po wykonaniu akcji, co za tym idzie, pętla wykona się przynajmniej jeden raz. Konstrukcja pętli do–while pokazana została poniżej.

```
do
	akcje
while (warunek)
```

Przykładem zastosowania może być poniższy listing, w którym użytkownik wpisuje liczbę i jeśli jest ona różna od 19 to dostaje komunikat o tym, że liczba ta jest mniejsza, większa bądź z nie dozwolonego zakresu.

```
#include <stdio.h>

main ()
{
	int liczba;

	do
	{
		printf(": ");
		scanf("%d", &liczba);

		switch (liczba)
```

```
          {
               case 1 ... 18 : printf("Za mala\n");
                          break;
               case 20 ... 40 : printf("Za duza\n");
                           break;
               default:   if (liczba == 19)
                               printf("Trafiles!\n");
                          else
                               printf("Nie z tego zakresu!\n");
                          break;
          }

     } while (liczba != 19);

}
```

Listing 3.3.3.1 Użycie pętli do – while

Brak sprawdzenia poprawności wprowadzanych danych uniemożliwia jego bezbłędną pracę, chodziło tutaj natomiast o pokazanie samego zastosowania pętli do–while, która jak widać wykonuje najpierw akcje (tutaj pobranie danych, sprawdzenie warunków, wyświetlenie informacji) a później sprawdza warunek, czy pobrana liczba jest różna od 19. Jeśli tak jest, to akcje ponownie są wykonywane. Pętla przestaje się wykonywać jeśli wprowadzoną liczbą jest liczba 19.

3.4 Instrukcja break

Z instrukcją break spotkaliśmy się już przy instrukcji switch, która kończyła jej działanie. Z użyciem break w pętlach for, while, do–while jest analogicznie – przerywa ona natychmiastowo działanie pętli i przechodzi do wykonywania kolejnych (jeśli występują) instrukcji poza pętlą. Niech przykładem zastosowania instrukcji break będzie poniższy listing.

```
#include <stdio.h>
main ()
{
     int i, granica = 30;
     for (i = 0; ;i++)
          if (i == 30)
               break;
          else
               printf("%d%c", i, (i == granica - 1) ? '\n' : ' ');
}
```

Listing 3.4.1 Użycie instrukcji break

Jak widać w pętli for nie ma drugiego wyrażenia (warunku) zakończenia pętli. Gdybyśmy nie użyli instrukcji break to pętla wykonywała by się w nieskończoność.

Trzeba pamiętać o jednym, instrukcja break przerywa pętle w której została zapisana. Jeśli instrukcja break znajdzie się w zagnieżdżonej pętli to przerwie tylko jej działanie, pętla zewnętrzna będzie działać tak, jak działała.

3.5 Instrukcja continue

Instrukcja continue trochę różni się od instrukcji break. W zasadzie też „coś” przerywa, lecz nie działanie całej pętli, a aktualnie wykonywanej iteracji. Akcje pod instrukcją continue są ignorowane. Po wykonaniu instrukcji continue w przypadku pętli for zwiększany jest licznik zmiennej sterującej, a w przypadku pętli while oraz do–while sprawdzany jest warunek.

Niech przykładem użycia instrukcji continue będzie listing pokazujący, że faktycznie ta instrukcja przerywa aktualnie wykonywane czynności.

```
#include <stdio.h>

main ()
{
	int i;
	int przed = 0, po = 0;

	for (i = 0; i < 10; i++)
	{
		przed++;
		continue;
		po++;
	}
	printf("Przed: %d\n", przed);	// 10
	printf("Po: %d\n", po);		// 0
}
```

Listing 3.5.1 Użycie instrukcji continue

3.6 Instrukcja goto, etykiety

W języku C występuje instrukcja goto, która to w miejscu wywołania robi bezwarunkowy skok do miejsca oznaczonego konkretną etykietą. Składnia tej instrukcji jest następująca.

```
goto nazwa_etykiety;

nazwa_etykiety: akcje
```

W poniższym przykładzie została użyta instrukcja goto, która drukuje cyfry od 0 do 9. Na kolejnym listingu została pokazana pętla for, która robi dokładnie to samo. Długość listingów mówi sama za siebie, którą wersję lepiej użyć. Nie mniej jednak instrukcję goto czasem używa się do wyjścia z bardzo zagłębionych pętli. Lecz w większości przypadków można się obejść bez niej. Za pomocą instrukcji goto możemy się odwołać tylko do tych etykiet, które są zdefiniowane w tej samej funkcji, z której ma być ona wywołana.

```
#include <stdio.h>

main ()
{
     int i = 0;

start:     if (i >= 10)
                goto stop;
           else
           {
                printf("%d\n", i);
                goto zwieksz;
           }
zwieksz:   i++;
           goto start;
stop:      printf("Koniec\n");
}
```

Listing 3.6.1 Zasada działania pętli for przy pomocy goto

```
#include <stdio.h>
main ()
{
     int i;
     for (i = 0; i < 10; i++)
          printf("%d\n", i);
     printf("Koniec\n");
}
```

Listing 3.6.2 Wyświetlanie cyfr od 0 do 9

4 Funkcje

Każdy program posiada funkcje. Do tej pory używalismy funkcji zdefiniowanych w bibliotece standardowej. W tym punkcie postaram się wyjaśnić jak tworzy się funkcje własnoręcznie, jak rozdzielać funkcje pomiędzy plikami, jak je wywoływać itp.

W gruncie rzeczy funkcje są bardzo przydatne, ponieważ jeśli jakąś operację mamy wykonać dwukrotnie z różnymi wartościami, to po co pisać właściwie ten sam kod dwa razy, skoro można napisać to raz i wywołać funkcję dwa razy z różnymi argumentami. Jest to oszczędność czasu, miejsca oraz przede wszystkim zyskujemy na czytelności kodu. W poniższych podpunktach zawarte są te wszystkie informacje, które pomogą nam w używaniu funkcji.

4.1 Ogólna postać funkcji oraz funkcja zwracająca wartości całkowite

Ogólna postać funkcji ma się następująco:

```
typ_zwracanej_wartosci nazwa_funkcji (parametry funkcji jesli wystepuja)
{
      cialo funkcji
}
```

Jako typ zwracanej wartości rozumiemy jeden z typów dostępnych w języku C (int, float, itp. lub void – o tym trochę później).

Nazwa funkcji – nazwa za pomocą której będziemy się odwoływać do naszej funkcji. Zasady nazywania funkcji są takie same jak zasady nazywania zmiennych.

Parametry funkcji – funkcja może przyjmować jakieś parametry i jeśli tak jest, to wpisujemy nazwy typów wraz z nazwami zmiennych rozdzielonych przecinkami pomiędzy nawiasami okrągłymi.

W ciele nowo tworzonych funkcji podobnie jak w funkcji main występują różnego rodzaju instrukcje. Do funkcji odwołujemy się podając jej nazwę. Jeśli funkcja przyjmuje jakieś argumenty to podajemy je, rozdzielając je przecinkami pomiędzy nawiasami okrągłymi. Jeśli funkcja nie przyjmuje żadnych argumentów, to zostawiamy nawiasy puste. Dla pełnego zrozumienia jak to wszystko działa pokazany

został listing 4.1.1 w którym występują dwie funkcje, główna oraz nwd[1].

```
#include <stdio.h>

int nwd (int liczba1, int liczba2);

main ()
{
	printf("NWD(%d, %d) = %d\n", 10, 14, nwd(10, 14));
	printf("NWD(%d, %d) = %d\n", 28, 14, nwd(28, 14));
	printf("NWD(%d, %d) = %d\n", 100, 30, nwd(100, 30));
	printf("NWD(%d, %d) = %d\n", 1024, 64, nwd(1024, 64));

	return 0;
}

int nwd (int liczba1, int liczba2)
{
	int c;
	while (liczba2 != 0)
	{
		c = liczba1 % liczba2;
		liczba1 = liczba2;
		liczba2 = c;
	}
	return liczba1;
}
```

Listing 4.1.1 Funkcja największy wspólny dzielnik

Zacznijmy więc od początku, deklarację int nwd (int liczba1, int liczba2); nazywa się prototypem funkcji. Prototyp funkcji[2] informuje kompilator o tym, jakiego typu jest dana funkcja oraz jakiego typu parametry przyjmuje. Jak widać funkcja nwd przyjmuje dwa parametry typu całkowitego. Zasadniczo w prototypie funkcji nie trzeba pisać nazw zmiennych, wystarczą same typy, tak więc zapis int nwd (int, int); jest równoważny. Jeśli typ zwracanej wartości zostanie pominięty, to funkcja jest traktowana jak funkcja zwracająca typ int.

Odnośnie wywołania funkcji powiem za chwilę. Najpierw zacznę omawiać naszą nową funkcję – nwd. Pierwsza linia definicji funkcji nwd musi być taka sama jak prototyp funkcji, z tym że tutaj już muszą być podane nazwy parametrów, które funkcja przyjmuje, ponieważ funkcja tworzy zmienne lokalne o takich właśnie nazwach i na nich wykonuje operacje. Do funkcji przekazywane są argumenty przez wartość, znaczy to tyle, że wartości argumentów zostają skopiowane i przypisane do zmiennych

1 Największy Wspólny Dzielnik – Funkcja bazująca na algorytmie Euklidesa
2 Wyjaśnienie użyteczności prototypów funkcji zostało przedstawione w punkcie 4.2

lokalnych (parametrów funkcji). Tak więc wartości przekazane z funkcji wywołującej funkcję nwd nie ulegną zmianie. O innym sposobie przekazywania argumentów dowiesz się w rozdziale 5.

W prototypie funkcji występował średnik na końcu, w definicji go nie ma. Ciało funkcji znajduje się pomiędzy nawiasami klamrowymi.

Ciało funkcji nwd posiada pewne instrukcje (algorytm Euklidesa). Ważna natomiast jest instrukcja return liczba1; która zwraca wartość całkowitą[3] w miejscu wywołania funkcji.

W funkcji main drukujemy największy wspólny dzielnik dla czterech par liczb. Mam nadzieję, że widzisz już zalety używania funkcji. Raz napisana funkcja została wywołana czterokrotnie z różnymi argumentami.

Jak już pewnie zauważyłeś funkcja main też posiada instrukcję return 0; Nie jest to przypadek, ponieważ funkcja main też powinna zwracać wartość do miejsca wywołania (system operacyjny).

Funkcje mogą zwrócić wartość, która nie będzie wykorzystana. Oczywiście jest to bez sensu, ale nic nie stoi na przeszkodzie by wywołać funkcję nwd, nie przypisując jej do żadnej zmiennej. Jeśli przypisujemy do zmiennej wywołanie funkcji, to należy zwrócić uwagę na typy zmiennych.

Nasza funkcja nwd została wywołana jako argument funkcji printf, po obliczeniu i zwróceniu wartości, wynik zostanie wyświetlony na ekranie.

4.2 Funkcje zwracające wartości rzeczywiste

Instnieją funkcje zwracające inne wartości niż wartości całkowite. Wiele funkcji z biblioteki standardowej (np. math.h) jak np. sin, czy cos zwracają wartości typu double. Definicja funkcji jest analogiczna, więc nie będę jej tu przedstawiał. Różnica polega na tym, że typem zwracanej wartości jest jeden z typów rzeczywistych używanych w języku C. Dla przykładu podany został listing 4.2.1, w którym została napisana funkcja sum_geom, która oblicza sumę n początkowych wyrazów ciągu geometrycznego. Podczas wywołania funkcji argumenty należy podać w takiej kolejności: a1 (wyraz pierwszy), q (iloraz), n (ilość wyrazów).

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

3 Wartość całkowitą ponieważ typem zwracanej wartości przez tę funkcję jest typ int.

```
double sum_geom(double a1, double q, unsigned n);
main()
{
     unsigned n = 200;
     double a1 = -2.0;
     double q = 0.25;
     printf("Dla ciagu o parametrach: a1 = %.2f, q = %.2f\n", a1, q);
     printf("Suma %u poczatkowych wyrazow = %f\n", n, sum_geom(a1, q, n));
     return 0;
}
double sum_geom(double a1, double q, unsigned n)
{
     double sum;
     if (q == 1)
           return n*a1;
     else
           sum = a1 * (1 - pow(q, n))/(1 - q);
     return sum;
}
```

Listing 4.2.1 Suma ciągu geometrycznego

W powyższym przykładzie w funkcji main wywołujemy funkcję sum_geom, która za pomocą wzorów na sumę ciągu geometrycznego oblicza sumę n początkowych wyrazów. Obliczoną wartość wstawia w miejsce wywołania. Wartość ta zostanie wyświetlona na ekranie monitora.

Jeśli zastanawiałeś się po co są prototypy funkcji, to postaram się to wytłumaczyć w tym miejscu. De facto funkcja nie musiałaby mieć prototypu jeśli znajduje się w tym samym pliku co funkcja z której zostanie ona wywołana oraz jej definicja znajduje się nad funkcją wywołującą. Poniższy przykład pokazuje, że program w takiej postaci skompiluje się bez błędów i wyświetli poprawny wynik.

```
#include <stdio.h>

double podziel(double a, double b)
{
     return a/b;
}

main ()
{
     printf("4/9 = %f\n", podziel(4.0,9.0));    // 0.444444
     return 0;
}
```

Liting 4.2.2 Bez prototypu funkcji

Jeśli natomiast funkcja podziel znajdzie się poniżej funkcji main oraz nie będzie prototypu funkcji, to kompilator wyświetli następujące błędy i ostrzeżenia:

```
bez_prototypu.c: In function ‘main’:
bez_prototypu.c:6: warning: format ‘%f’ expects type ‘double’, but argument 2
has type ‘int’
bez_prototypu.c: At top level:
bez_prototypu.c:10: error: conflicting types for ‘podziel’
bez_prototypu.c:6: note: previous implicit declaration of ‘podziel’ was here
```

Ostrzeżenie o tym, że format **%f** wymaga typu rzeczywistego, a argument jest typu całkowitego może wydać się nieco dziwne. Dlaczego kompilator pokazuje taką informację, skoro funkcja ma zadeklarowany typ double? Otóż odpowiedź na to pytanie jest następująca.

Jeśli nie ma prototypu funkcji (czyli nie ma jawnej informacji o tym jaki typ wartości funkcja zwraca) to funkcja ta jest deklarowana w pierwszym miejscu w ktorym występuje, czyli tutaj: printf("4/9 = %f\n", podziel(4.0,9.0)); Jak było wspomniane wcześniej, jeśli typ zwracanej wartości nie zostanie zadeklarowany, to funkcja jest traktowana jakby zwracała typ int. Funkcja podziel z powodu iż nie została nigdzie wcześniej zadeklarowana, jest deklarowana właśnie w tej linii. A ponieważ chcemy wyświetlić liczbę rzeczywistą – użyliśmy deskryptora formatu **%f**, który oczekuje liczby typu rzeczywistego, a okazuje się, że „dostaje” liczbe całkowitą.

Słowo „dostaje” zostało zapisane w cudzysłowie ponieważ program się nie skompilował, więc tak naprawdę nic nie dostał. Odpowiedź na poprzednie pytanie ma dużo wspólnego z kolejnym pytaniem odnośnie błędu w linii dziesiątej, która mówi o konfikcie typów funkcji podziel.

Jak już pewnie się domyślasz deklaracja oraz definicja funkcji (deklaracja + ciało funkcji) muszą być jednakowych typów. Nasza funkcja została potraktowana jak zwracająca typ int, a definicja mówi co innego, a mianowicie, że zwraca typ double. Dlatego też ten błąd został wyświetlony przez kompilator.

Jednym ze sposobów na poprawną kompilację omówionego przypadku jest deklaracja prototypu funkcji globalnie. Drugi sposób to deklaracja prototypu w funkcji z której dana funkcja ma zostać wywołana. Poniższe listingi kompilują się bez wyżej wymienionych błędów i ostrzeżeń.

```
#include <stdio.h>
double podziel (double a, double b);

main ()
{
     printf("4/9 = %f\n", podziel(4.0,9.0));    // 0.444444
```

```
	return 0;
}

double podziel(double a, double b)
{
	return a/b;
}
```

Listing 4.2.3 Deklaracja prototypu

```
#include <stdio.h>

main ()
{
	double podziel (double a, double b);
	printf("4/9 = %f\n", podziel(4.0,9.0));    // 0.444444
	return 0;
}


double podziel(double a, double b)
{
	return a/b;
}
```

Listing 4.2.4 Deklaracja prototypu wewnątrz main

4.3 Funkcje nie zwracające wartości oraz brak argumentów

W języku C występują też funkcje, które nie zwracają żadnych wartości. W rozdziale dotyczącym tablic i wskaźników pokazane zostało szersze zastosowanie tych funkcji, aniżeli przedstawione w tym podpunkcie. Różnica w definicji pomiędzy funkcjami, które nie zwracają wartości a tymi zwracającymi jest następująca. Jako typ zwracanej wartości wpisujemy słowo kluczowe void. Jako przykład podany został listing 4.3.1, w którym to drukowane są liczby z odstępem czasowym. Przykład oczywiście mało praktyczny, nie mniej jednak chodzi o pokazanie sposobu deklarowania tego typu funkcji, oraz brak argumentów, o którym zaraz powiem.

```
#include <stdio.h>
void poczekaj(void);
main (void)
{
	int i;
	printf("Drukuj liczby z odstepem czasowym\n");
```

```
	for (i = 1; i < 50; i++)
	{
		printf("%d\n", i);
		poczekaj();
	}
	return 0;
}
void poczekaj(void)
{
	int i;
	for (i = 0; i < 25E6; i++)
		;
}
```

Listing 4.3.1 Użycie funkcji nie zwracającej wartości

Pierwszą rzeczą inną w stosunku do poprzednich programów jaką można zauważyć jest użycie słowa kluczowego void pomiędzy nawiasami funkcji zarówno poczekaj jak i main. Słowa tego używa się po to, aby powiedzieć kompilatorowi, że funkcja nie przyjmuje żadnych argumentów. Dzięki temu zachowujemy większę kontrolę poprawności. Od tej chwili w programach, w których występują funkcje nie przyjmujące argumentów będziemy wpisywać w miejsce parametrów słowo kluczowe void zamiast zostawiać puste nawiasy.

4.4 Pliki nagłówkowe

Jeśli program, który piszemy zawiera dużo funkcji (przez nas napisanych) to jest sens zastanowienia się, czy nie lepiej było by podzielić go na parę plików, z których każdy odpowiadałby za konkretną czynność. Poniższy przykład, który pokażę jest oczywiście fikcyjnym problemem, bowiem wielkość tego programu jest bardzo mało i można by było wszystko zapisać w jednym pliku. Nie mniej jednak chcę pokazać pewne czynności o których trzeba pamiętać dzieląc kod programu pomiędzy kilkoma plikami.

Nazwa pliku	Zawartość	Listing
main.c	Wywołania funkcji	4.4.1
prototypes.h	Prototypy funkcji, zmienna globalna, stała wyliczenia	4.4.2

pobierztekst.c	Definicja funkcji	4.4.3
wyswietltekst.c	Definicja funkcji, zmienna globalna	4.4.4
skopiujtekst.c	Definicja funkcji	4.4.5

Tabela 4.4.1 Podział programu na pliki – opis

A więc zacznijmy od początku, pliki main.c oraz prototypes.h mają taką postać.

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

main (void)
{
	char line[MAXLENGHT];
	extern int id;

	pobierzTekst(line);	/* Pobieranie tekstu do line[] */
	wyswietlTekst(line);	/* Wyswietlanie tekstu z line[] */
	skopiujTekst(line);	/* Kopiowanie tesktu line[] do bufor[] */
	wyswietlTekst(bufor);	/* Wyswietlanie tekstu z bufor */
	printf("%d\n", id);
	return 0;
}
```

Listing 4.4.1 main.c

```
#define MAXLENGHT 1000

char bufor[MAXLENGHT];

void pobierzTekst(char line[]);
void wyswietlTekst(char line[]);
void skopiujTekst(char line[]);
```

Listing 4.4.2 prototypes.h

W tym miejscu należy się pewne wyjaśnienie odnośnie kilku szczegółów zawartych w tych dwóch listingach. Po pierwsze druga linijka w listingu 4.4.1 #include "prototypes.h" informuje preprocesor o tym, aby dołączył plik prototypes.h znajdujący się w tym samym katalogu co main.c.

W funkcji main deklarujemy tablicę znaków line o długości MAXLENGHT – zadeklarowana w prototypes.h. Dzięki temu, że dołączyliśmy plik z prototypami, w którym znajduje się stała

symboliczna możemy jej użyć podczas tworzenia tablicy znaków. Kolejna linijka to deklaracja zmiennej zewnętrznej, której definicja znajduje się w innym pliku (wyswietltekst.c). Słowo kluczowe extern jest tutaj potrzebne, było by zbędne, jeśli cały kod programu byłby w jednym pliku, tak jak na listingu 2.2.5. Kolejnymi liniami są wywołania funkcji, które znajdują się w innych plikach. Linia wyswietlTekst(bufor); pobiera jako argument zmienną globalną. W tym wypadku nie musimy deklarować jej przed użyciem wraz ze słowem kluczowym extern ponieważ definicja zmiennej bufor została dołączona wraz z plikiem prototypes.h. Więcej o słowie kluczowym extern znajduje się w punkcie 4.5. Kolejne trzy listingi to definicje funkcji, które zostały użyte w funkcji głównej.

```
#include <stdio.h>
#include "prototypes.h"
void pobierzTekst(char line[])
{
	int c;
	int i = 0;

	while ((c = getchar()) != EOF && c != '\n' && i < MAXLENGHT - 1)
		line[i++] = c;
	line[i] = '\0';
}
```

Listing 4.4.3 pobierztekst.c

```
#include <stdio.h>
int id = 10;
void wyswietlTekst(char line[])
{
	printf("%s\n", line);
}
```

Listing 4.4.4 wyswietlteskt.c

```
#include "prototypes.h"

void skopiujTekst(char line[])
{
	int i = 0;
	int c;

	while ((c = line[i]) != '\0')
		bufor[i++] = c;
	bufor[i] = '\0';
}
```

Listing 4.4.5 skopiujtekst.c

A więc po kolei, funkcja pobierzTekst przyjmuje jako argument tablicę znaków (tablice omówione są w rozdziale 5.). Tworzymy pomocniczą zmienną c, do której przypisujemy funkcję getchar, która pobiera znaki od użytkownika. Jeśli c (pobrany znak) jest różne od End Of File (stała symboliczna oznaczająca koniec pliku – dlatego też musieliśmy dołączyć plik nagłówkowy stdio.h) i c jest różne od znaku nowego wiersza oraz i jest mniejsze od MAXLINE – 1 to przypisz pobrany znak do tablicy na pozycji i a po przypisaniu zwiększ i o 1. Jeśli warunek nie został spełniony (wartość i została zwiększona w poprzednim kroku) to przypisz znak końca tablicy na pozycji i.

Funkcja wyswietlTekst to funkcja wyświetlająca tekst, jako argument przyjmuje tablicę znaków. Funkcja skopiujTekst również przyjmuje tablicę znaków jako argument. Tworzymy zmienną pomocniczą i z wartością początkową zero oraz zmienną c, do której przypisujemy znak z tablicy z pozycji i oraz sprawdzamy, czy jest on różny od znaku końca tablicy. Jeśli tak jest to do tablicy bufor na pozycji i przypisujemy ten właśnie znak, oraz zwiększamy i o 1. Jeśli c równa się \0 to przerywamy pętlę i przypisujemy do tablicy na pozycji i znak końca.

Aby skompilować wszystkie te pliki wystarczy wpisać polecenie:

```
$ gcc main.c pobierztekst.c skopiujtekst.c wyswietltekst.c -o main
```

Bądź skróconą wersję – pod warunkiem, że w katalogu znajdują się tylko te pliki z rozszerzeniem .c o których w tym podpunkcie mówiliśmy.

```
$ gcc *.c -o main
```

4.4.1 Kompilacja warunkowa

Kompilacja warunkowa opiera się o proste sprawdzenie warunku podczas tłumaczenia kodu. Jeśli wartość pewnego kontrolnego parametru wynosi tyle, to skompiluj tę część kodu, jeśli parametr posiada inną wartość to nie kompiluj tego, a skompiluj coś innego. Na poniższym przykładzie pokaże co miałem na myśli pisząc o kontrolnym parametrze.

```
#include <stdio.h>
#if KONTROLA == 1
      double z2 = 10.0;
#elif KONTROLA ==  2
      double z2 = 15.0;
#elif KONTROLA ==  3
```

```
      double z2 = 20.0;
#else
      #define X
#endif
main (void)
{
      #if !defined(X)
      printf("%.1f\n", z2);
      #endif
      return 0;
}
```

Listing 4.4.1.1 Kompilacja warunkowa

Konstrukcja #elif założenia ma te same co kontrukcja else-if. Sprawdzany jest warunek, czy KONTROLA równa się 1, 2 lub 3. Jeśli tak jest to definiujemy zmienną z2 typu rzeczywistego z odpowiednią wartością. Jeśli zmienna kontrolna jest różna od wymienionych wartości lub nie jest zdefiniowana w ogóle to definiujemy X za pomocą #define.

Wartość wyrażenia defined (X) ma wartość jeden, jeśli X zostało zdefiniowane i zero w przeciwnym wypadku. My użyliśmy negacji, czyli instrukcja printf wykona się wtedy, jeśli X nie zostało zdefiniowane (czyli KONTROLA ma jedną z wymienionych wartości).

Pewnie się zastanawiasz dlaczego nigdzie nie ma zdefiniowanej stałej KONTROLA. Otóż można to zrobić na dwa sposoby. Pierwszy z nich polega na umieszczeniu definicji #define KONTROLA WARTOŚĆ na górze pliku. Drugi natomiast polega na użyciu specjalnej opcji podczas kompilacji i o tym właśnie zaraz napiszę.

Za pomocą opcji **-D** polecenia gcc możemy ustawić pewną wartość makra podczas kompilacji. Czyli w naszym wypadku, aby na ekranie pojawiła się wartość 15.0 musimy skompilować program w następujący sposób.

```
$ gcc nazwa_pliku.c -o nazwa_pliku -D KONTROLA=2
```

Funkcja main z listingu 4.4.1.1 mogłaby wyglądać nieco inaczej. Preprocesor posiada wbudowane instrukcje #ifdef oraz #ifndef, które odpowiednio oznaczają „jeśli zdefiniowano” oraz „jeśli nie zdefiniowano”. Czyli treść naszej funkcji main mogłaby wyglądać następująco:

```
#ifndef(X)
printf("%.1f\n", z2); #endif
```

4.5 extern, static, register

Wszystkie te słowa kluczowe odnoszą się do zmiennych, a extern i static również do funkcji. W poprzednim punkcie poniekąd omówione zostało słowo extern, nie mniej jednak w tym miejscu przedstawię trochę więcej szczegółów. Mam nadzieje, że powtórzone informacje bardziej pomogą, niż zaszkodzą.

4.5.1 extern

Słowo kluczowe extern używane jest do powiadomienia kompilatora o tym, iż używana będzie zmienna globalna, która nie jest zadeklarowana w pliku z którego będziemy się do niej odwoływać i jej definicja nie jest dołączana w pliku nagłówkowym. extern służy również do powiadomienia kompilatora o tym, iż zmienna jest zdefiniowana w danym pliku, natomiast wywoływana jest przed jej definicją. Poniższe przykłady mam nadzieję wyjaśnią ten teoretyczny wstęp.

Przykład 4.5.1.1

Posiadamy dwa pliki: main.c, glob.c. W drugim pliku mamy definicję zmiennej globalnej o nazwie id wraz z przypisaną wartością. Z pliku main.c próbujemy się odwołać do tej zmiennej. Poniżej znajdują się listingi, oraz dodatkowy opis.

```
#include <stdio.h>
main (void)
{
      printf("%d\n", id);
      return 0;
}
```

Listing 4.5.1.1 main.c

```
int id = 150;
```

Listing 4.5.1.2 glob.c

W tym przypadku, podczas kompilacji otrzymamy błąd informujący nas o tym, że kompilator nie zna tej zmiennej, tzn, że zmienna jest nie zadeklarowana.

```
main.c: In function 'main':
main.c:5: error: 'id' undeclared (first use in this function)
main.c:5: error: (Each undeclared identifier is reported only once
main.c:5: error: for each function it appears in.)
```

Problem ten można rozwiązać deklarując zmienną id w funkcji main używając słowa kluczowego extern. Jeśli plik main.c poprawimy i będzie wyglądał jak na listingu poniżej, kompilacja przebiegnie bez problemowo.

```
#include <stdio.h>

main (void)
{
	extern int id;
	printf("%d\n", id);
	return 0;
}
```

Listing 4.5.1.3 poprawiony main.c

Jeszcze dla sprostowania parę słów o definicji i deklaracji. W naszym przypadku definicja zmiennej id znajduje się w pliku glob.c – jest tam przypisana do niej wartość. W funkcji main jest deklaracja funkcji id z użyciem extern i tam już nie jest potrzebna wartość. W tym miejscu informujemy tylko kompilator, że taka zmienna istnieje i jest ona w innym pliku (lub w dalszej części pliku – o tym zaraz) oraz, że chcemy jej użyć.

Przykład 4.5.1.2

Posiadamy jeden plik main.c, w którym definicja zmiennej globalnej znajduje się po jej wywołaniu. Na poniższym listingu znajduje się omówiona sytuacja.

```
#include <stdio.h>

main (void)
{
	printf("%d\n", id);
	return 0;
}

int id = 150;
```

Listing 4.5.1.4 main.c

Jeśli tak będziemy chcieli skompilować ten plik, to otrzymamy błąd identyczny jak w punkcie 4.5.1.1. Kompilator nie zna tej zmiennej. Rozwiązania są dwa, albo zadeklarujemy zmienną id przed funkcją main, lub zostawimy ją w tym miejscu, w którym jest i użyjemy słowa extern podczas deklarowania zmiennej w funkcji main. Drugi sposób został pokazany na poniższym listingu.

```
#include <stdio.h>
main (void)
{
      extern int id;
      printf("%d\n", id);
      return 0;
}
int id = 150;
```

Listing 4.5.1.5 poprawione main.c

Przykład 4.5.1.3

Posiadamy dwa pliki: main.c, naglowki.h w funkcji main odwołujemy się do zmiennej, która jest zdefiniowana w pliku nagłówkowym bez użycia słowa kluczowego extern. Kompilacja przebiega bez problemu i zmienna zostaje użyta. Plik nagłówkowy musi zostać dołączony przez dyrektywę #include. Poniższe listingi pokazują ten sposób.

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

main (void)
{
     printf("%d\n", id);
     return 0;
}
```

Listing 4.5.1.6 main.c

```
int id = 15;
```

Listing 4.5.1.7 naglowki.h

W tym przypadku nie musimy deklarować zmiennej wewnątrz używanej funkcji z użyciem extern. Dołączyliśmy plik nagłówkowy, w którym definicja zmiennej została zawarta, więc kompilator wie już o tej zmiennej i może jej użyć.

Słowo kluczowe extern ma jeszcze zastosowanie do funkcji. W punkcie 4.2 przedstawiona została

sytuacja (błąd wraz z opisem) co by się stało, gdyby definicja funkcji znajdowała się po jej wywołaniu i jednocześnie nie było by prototypu funkcji. Zawsze lepiej użyć prototypu funkcji, natomiast istnieje też możliwość, aby go nie używać, a użyć słowa kluczowego extern w funkcji, która chce się odwołać do wybranej funkcji będącej w tym samym lub innym pliku. Poniższy listing prezentuje daną sytuację.

```
#include <stdio.h>

main (void)
{
	extern double podziel();
	printf("%f\n", podziel(1.0, 2.0));
	return 0;
}

double podziel (double a, double b)
{
	return a/b;
}
```

Listing 4.5.1.8 Kolejne użycie extern

W tym przykładzie widać, że funkcja podziel jest w tym samym pliku, lecz jeśli chodzi o same zastosowanie extern, to równie dobrze mogłaby być ona w innym pliku. Widzimy, że w funkcji main deklarujemy funkcję podziel zwracającą typ double. Nie potrzeba deklarować parametrów, wystarczy poinformować kompilator o tym, że chcemy użyć takiej funkcji, która jest zdefiniowana w dalszej części, bądź w innym pliku.

4.5.2 static

Zmienne statyczne można definiować zarówno globalnie jak i lokalnie. Różnica między statyczną a zwykłą zmienną globalną jest taka, że statyczne zmienne globalne mają zasięg od miejsca wystąpienia do końca kompilowanego pliku. Czyli jeśli umieścimy pewną zmienną w innym pliku, to będzie ona dostępna tylko dla funkcji znajdujących się w tamtym pliku. Użycie extern nic nie pomoże, do zmiennej nie będziemy mogli się odwołać. Jest to metoda na ukrycie zmiennych przed innymi funkcjami, które nie potrzebują mieć do niej dostępu. Zmienne statyczne mogą mieć taką samą nazwę jak inne zmienne globalne i nie będzie to kolidować pod warunkiem, że definicje tych dwóch zmiennych nie występują w jednym pliku. Na poniższych przykładach zostało to pokazane.

Przykład 4.5.2.1

Mamy dwa pliki: main.c, wynik.c. W tym drugim mamy zdefiniowaną zmienną statyczną globalną z zainicjowaną wartością początkową. Używamy jej do operacji w funkcji wynik. W funkcji main odwołujemy się do funkcji wynik i otrzymujemy wynik operacji sumy argumentów podzielonych przez c. Aby sprawdzić czy można się odwołać do zmiennej statycznej z innego pliku usuń komentarze otaczające deklarację extern oraz instrukcję printf drukującą zmienną c. Informacje od kompilatora powinny być pouczające.

```
#include <stdio.h>
double wynik (double, double);
main (void)
{
      /* extern double c; */
      printf("%f\n", wynik(1.0, 10.0));
      /* printf("%f\n", c); */
      return 0;
}
```

Listing 4.5.2.1 main.c

```
static double c = 100.0;

double wynik (double a, double b)
{
      return (a + b)/c;
}
```

Listing 4.5.2.2 wynik.c

Przykład 4.5.2.2

W dwóch plikach mamy definicję dwóch zmiennych globalnych o tych samych nazwach, z czego jedna jest zmienną globalną statyczną.

```
#include <stdio.h>
int indeks = 11923;
int nr_indeksu (void);

main (void)
{
      printf("%d\n", indeks);
      printf("%d\n", nr_indeksu());
      return 0;
}
```

Listing 4.5.2.3 main.c

```
static int indeks = 22943;

int nr_indeksu (void)
{
	return indeks;
}
```

Listing 4.5.2.4 ind.c

Widać, że w pliku ind.c zmienna globalna nazywa się tak samo, nie przeszkadza to, jeśli jest to zmienna statyczna. Jeśli nie było by słowa static, to kompilator wyświetli błąd z informacją, że występuje wielokrotna definicja zmiennej, co jest oczywiście zabronione. Jeśli zmienna globalna zwykła i statyczna były by zdefiniowane w jednym pliku o tych samych nazwach, to również kompilator wyświetli bład.

Przykład 4.5.2.3

Funkcje z natury są globalne i dostępne dla wszystkich części programu. Można natomiast użyć static, by funkcja była widziana tylko w pliku, w którym występuje jej definicja. Na poniższych listingach widać, ze funkcję ilosc_cali można użyć tylko w pliku **il_cal.c**, natomiast próba odwołania się do tej funkcji z innego pliku, kończy się błędem.

```
#include <stdio.h>
double zamien (double);
main (void)
{
	/*
	extern double ilosc_cali ();
	printf("5 cm to %f cali\n", ilosc_cali(5.0));
	*/
	printf("5 cm to %f cali\n", zamien(5.0));
	return 0;
}
```

Listing 4.5.2.5 main.c

```
static double ilosc_cali (double centymetr)
{
	const double cal = 0.3937007874;
	return centymetr * cal;
}

double zamien (double centymetr)
```

```
{
      return ilosc_cali (centymetr);
}
```

Listing 4.5.2.6 il_cal.c

Oczywiście można by było funkcję ilosc_cali zdefiniować normalnie i to za jej pomocą otrzymywać wynik bezpośrednio. Natomiast chciałem pokazać, że z pomocą słowa static możemy „ukryć" również funkcję.

Istnieją jeszcze zmienne statyczne wewnętrzne, czyli lokalne dla funkcji. Zmienne te w odróżnieniu od zwykłych zmiennych lokalnych różnią się tym, iż nie znikają po zakończeniu działania funkcji, tzn przy kolejnym wywołaniu pamiętają poprzednią wartość. Najlepszym przykładem na to będzie poniższy listing, w którym z funkcji main wywołujemy drugą funkcję trzykrotnie, za każdym razem funkcja ta drukuje po raz który została wywołana. Gdyby w funkcji ilosc_wywolan zmienna i nie była zadeklarowana jako static, to dostalibysmy wyniki nie zgodne z oczekiwaniami. Funkcja pokazywała by za każdym razem gdy została wywołana, że wywoływana jest po raz pierwszy.

```
#include <stdio.h>

void ilosc_wywolan (void);

main (void)
{
      ilosc_wywolan();
      ilosc_wywolan();
      ilosc_wywolan();
      return 0;
}

void ilosc_wywolan (void)
{
      static int i = 1;

      printf("Funkcja wywolana po raz: %d\n", i);
      i++;
}
```

Listing 4.5.2.7 Zmienna lokalna statyczna

4.5.3 register

Ideą zmiennych rejestrowych jest to, że dają one informacje kompilatorowi, że będą bardzo często wykorzystywane. Zmienne te umieszczane są w rejestrach maszyny, na której są używane dzięki czemu program może wykonywać się szybciej. Wszystko zależy jednak od kompilatora, ponieważ kompilator może całkowicie zignorować słowo kluczowe register i traktować ją normalnie. Dla sprawdzenia tego mechanizmu przeprowadziłem pewien test. Uruchomiłem program, który właściwie złożony jest tylko z jednej pętli, którą wykonuje 1E9 razy (1 i dziewięć zer). Raz zmienną sterującą jest zmienna i, a raz zmienna i rejestrowa. Ciekawostką może być fakt, iż przeprowadziłem test na dwóch komputerach. Dane komputerów zamieszczone zostały w tabeli 4.5.3.1. Wyniki zostały umieszczone w tabelach 4.5.3.2 oraz 4.5.3.3. Różnice są zdumiewające.

Tabela	**4.5.3.2**	**Tabela**	**4.5.3.3**
Nazwa procesora	Intel Pentium 4 M-740 (Centrino Sonoma)	Nazwa Procesora	Intel Core 2 Duo P7450
Taktowanie procesora	1.7GHz	Taktowanie procesora	2.13GHz
Dodatkowe informacje	Cache 2MB FSB 533MHz	Dodatkowe informacje	Cache 3MB FSB 1066MHz
Pamięć RAM	1536MB DDR II	Pamięć RAM	4096MB DDR II

Tabela 4.5.3.1 Wykaz sprzętu

int i		**register int i**	
Lp	Czas [s]	Lp	Czas [s]
1	6.612	1	6.213
2	6.695	2	6.244
3	6.700	3	6.226
4	6.649	4	6.260
5	6.735	5	6.247
6	6.721	6	6.384
7	6.784	7	6.293

8	6.725	8	6.276
9	6.740	9	6.222
10	6.788	10	6.291
Średni czas	6.715	Średni czas	6.266
Średnia różnica		0.449	

Tabela 4.5.3.2 Porównanie zmiennej register ze zwykłą – komputer 1.

int i		**register int i**	
Lp	Czas [s]	Lp	Czas [s]
1	3.872	1	2.369
2	3.869	2	2.363
3	3.870	3	2.368
4	3.886	4	2.361
5	3.877	5	2.370
6	3.878	6	2.379
7	3.870	7	2.367
8	3.873	8	2.366
9	3.877	9	2.372
10	3.874	10	2.373
Średni czas	3.875	Średni czas	2.369
Średnia różnica		1.506	

Tabela 4.5.3.3 Porównanie zmiennej register ze zwykłą – komputer 2.

Kod programu zawarty jest w listingu poniżej

```
#include <stdio.h>

main (void)
{
	register int i;
	for (i = 0; i < 1E9; i++)
		;
	return 0;
}
```

Listing 4.5.3.1 Zmienna register

Więcej informacji na temat sprawdzania czasu wykonywania się programów znajduje się w dodatku B.

4.6 Funkcje rekurencyjne

Funkcje rekurencyjne są to takie funkcje, które wywołują same siebie w pewnym momencie. Dość znanym przykładem może być funkcja silnia, która oblicza iloczyn liczb początkowych[1]. Niech przykładem funkcji rekurencyjnej będzie poniższy listing.

```
#include <stdio.h>

int silnia (int arg);

main(void)
{
	int n = 5;
	printf("%d! = %d\n", n, silnia(n));
	return 0;
}

int silnia (int arg)
{
	if (!arg)
		return 1;
	else
		return arg * silnia (arg-1);
}
```

Listing 4.6.1 Silnia – funkcja rekurencyjna

Cała rekurencja odbywa się w przedostatniej linii, w której wywołujemy tę samą funkcję z argumentem pomniejszonym o 1. Aby zrozumieć jak to tak naprawdę działa postaram się opisać to w kilku zdaniach. Z funkcji main wywołujemy funkcję silnia z argumentem 5. Sterowanie przechodzi do funkcji silnia i sprawdza warunek, czy argument (5) jest równy zero. Jeśli nie jest (a na razie nie jest) to za pomocą instrukcji return zwróć wartość arg · silnia (arg – 1). Czyli w naszym przypadku 5 · silnia(4). silnia(4) wykonuje analogiczną pracę – zwraca 4 · silnia (3) i tak dalej, aż argument dojdzie do 1, w którym to zwróci 1 · silnia(0). silnia(0) zwraca po prostu 1 (warunek w funkcji). Czyli do funkcji printf zostanie zwrócona wartość 5 · 4 · 3 · 2 · 1 · 1.

1 Silnie oznacza się wykrzyknikiem. 5! = 1 · 2 · 3 · 4 · 5

5 Tablice i wskaźniki

Pomimo iż w poprzednich rozdziałach pojawiały się tablice, to jednak nie zostały one wytłumaczone na tyle obszernie, by można było to pominąć. O wskaźnikach na razie nie było mowy, tak więc w tym rozdziale zostaną one omówione dosyć szczegółowo. Tablice i wskaźniki mają ze sobą wiele wspólnego, natomiast najpierw zostaną omówione tablice oddzielnie, później wskaźniki, a w punkcie 5.3 zostaną pokazane zależności między tablicami, a wskaźnikami.

5.1 Tablice

Tablicę można było by sobie wyobrazić jako worek, do którego wrzucamy te same przedmioty, tylko o różnych kolorach. Jeśli tworzymy tablicę liczb całkowitych, to możemy mieć w niej tylko liczby całkowite, lecz wartości naturalnie mogą się różnić. Tablice zdecydowanie ułatwiają pracę z danymi konkretnego typu. Bezsensownie jest tworzyć 100 zmiennych typu int, nadawać im dziwne nazwy (oczywiście można by było nazwać je schematycznie np.: a1, …, a100, lecz jest to bezsensu), a potem odwoływać się do nich jakimiś dziwnymi sposobami. W poniższych podpunktach zostały omówione tablice jedno i wielowymiarowe.

5.1.1 Tablice jednowymiarowe

Tablicę definiujemy raz, podając jej nazwę, a w nawiasach kwadratowych rozmiar. Jeśli chcemy zainicjować wartości tablicy podczas jej tworzenia to możemy to zrobić stawiając po zamykającym nawiasie kwadratowym znak równości, a następnie wpisując wartości odpowiedniego typu rozdzielone przecinkami pomiędzy nawiasami klamrowymi. Przykładowe deklaracje tablic zostały pokazane poniżej.

```
int tab[5];

double axp[7];

int mpx[] = {3, 7, 9, 11};

double kxx[10] = {13.3, 19.0, 19.9, 192.4};
```

Teraz pewne wyjaśnienia się należą. Tablice z niezainicjowanymi wartości przechowują śmieci. Jeśli

inicjujemy wartości to nie musimy podawać rozmiaru tablicy, kompilator sam tę wartość obliczy. Natomiast jeśli podajemy wartość tablicy i inicjujemy tylko część elementów, to pozostałe elementy są zerami (tablica kxx).

Niech poniższa tabela będzie reprezentacją tablicy liczb calkowitych int tab[10];

10	15	20	145	43	234	14	0	0	11
tab[0]	tab[1]	tab[2]	tab[3]	tab[4]	tab[5]	tab[6]	tab[7]	tab[8]	tab[9]

Tabela 5.1.1.1 Tablica liczb całkowitych

Do elementów tablicy odwołujemy się za pomocą nazwy tablicy oraz jej indeksu. Bardzo ważną sprawą jest to, iż w języku C tablicę indeksuje się od wartości 0. Tak więc w tablicy dziesięcio elementowej ostatni element dostępny jest pod indeksem 9, co zostało pokazane w tabeli 5.1.1.1. Tak więc, aby wyświetlić wszystkie wartości tablicy tab wystarczy „wrzucić” ją do pętli. Zostało to pokazane na poniższym listingu.

```
#include <stdio.h>
main (void)
{
	int tab[10] = {10, 15, 20, 145, 43, 234, 14, 0, 0, 11};
	int i;
	for (i = 0; i < 10; i++)
		printf("tab[%d] = %d\n", i, tab[i]);
	return 0;
}
```

Listing 5.1.1.1 Drukowanie wszystkich elementów tablicy

Gdybyśmy utworzyli 10 zmiennych typu int, to aby wydrukować wszystkie, musielibyśmy wpisać 10 instrukcji printf. Niech ten przykład będzie dowodem na to, iż łatwiej operuje się tablicami, jeśli dane są jednego typu, niż ogromną ilością zmiennych.

5.1.2 Tablice wielowymiarowe

Tablice wielowymiarowe definiuje się bardzo podobnie z tą różnicą, że zamiast jednego rozmiaru podaje się dwa. O co tutaj chodzi? Wybraźmy sobie macierz znaną z matematyki (lub tabliczkę

czekolady – miejsce każdej kostki mogło by zostać określone na podstawie podania pozycji w wierszu oraz pozycji w kolumnie) jest to przykład tablicy dwuwymiarowej. Dla przykładu i lepszego zrozumienia pokazana została poniżej tabela, w której umieszczone są nazwy, przez które odwołujemy się do konkretnych pól.

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]
a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]

Tabela 5.1.2.1 Tablica dwuwymiarowa

Na poniższym listingu tworzymy tablicę 5x5 z zainicjowanymi wartościami (o tym za chwilę). Do poszczególnych pól odwołujemy się analogicznie jak w przypadku tablic jednowymiarowych z tą różnicą, że dodajemy drugi indeks. Patrząc na powyższą tabelkę widzimy, że aby dostać się do środkowego elementu musimy podać dwa indeksy o numerze 2, czyli a[2][2];

```
#include <stdio.h>
#define MAX 5
main (void)
{
      double tab[MAX][MAX] = {
                        {0.0, 0.1, 0.2, 0.3, 0.4},
                        {1.0, 1.1, 1.2, 1.3, 1.4},
                        {2.0, 2.1, 2.2, 2.3, 2.4},
                        {3.0, 3.1, 3.2, 3.3, 3.4},
                        {4.0, 4.1, 4.2, 4.3, 4.4}
                        };
      int i, j;
      for (i = 0; i < MAX; i++)
            for (j = 0; j < MAX; j++)
                printf("%.1f%c", tab[i][j], (j == MAX - 1) ? '\n' : '\t');
      return 0;
}
```

Listing 5.1.2.1 Tablica dwuwymiarowa

Za pomocą dwóch pętli drukujemy wszystkie wartości tablicy dwuwymiarowej. Pierwsza pętla – zewnętrzna – odpowiada za wyświetlanie wierszy, a wewnętrzna za wyświetlanie kolumn. W instrukcji printf występuje jako trzeci argument operator trzyargumentowy, który sprawdza, czy mamy do czynienia z piątą kolumną. Jeśli tak jest to stawia znak nowego wiersza, w przeciwnym wypadku stawia tabulację. Wszystko to w celu lepszej prezentacji danych.

Sposób inicjowania tablic wielowymiarowych jest podobny do inicjowania tablic jednowymiarowych. Róznica polega na tym, że w jednowymiarowych tablicach po przecinku podajemy wartości, w tablicy dwuwymiarowej podajemy tablicę, czyli wartości rozdzielone przecinkami w nawiasach klamrowych. Po ostatniej wartości może wystąpić przecinek, lecz nie musi. Przykład inicjowania tablicy trzy wymiarowej oraz wyświetlania wszystkich wartości pokazany został poniżej.

```
#include <stdio.h>

main (void)
{
	double tab[2][2][2] = {
				{ {1.0, 2.0}, {3.0, 4.0} },
				{ {5.0, 6.0}, {7.0, 8.0} }
			};
	int i, j, k;

	for (i = 0; i < 2; i++)
		for (j = 0; j < 2; j++)
			for (k = 0; k < 2; k++)
			printf("tab[%d][%d][%d] = %.1f\n", i, j, k, tab[i][j][k]);
	return 0;
}
```

Listing 5.1.2.2 Tablica trzywymiarowa

Tablice o wymiarze większym niż dwa raczej rzadziej są stosowane niż tablice jedno i dwuwymiarowe. Nie mniej jednak sposób tworzenia takich tablic został pokazany. Jeszcze jedna uwaga, w tablicach wielowymiarowych, pierwszy rozmiar może zostać pominięty – kompilator sam ustali jego wartość. A więc poniższe deklaracje są równoważne.

```
double tab[][MAX] = { ... };
double tab[][2][2] = { ... };
```

5.2 Wskaźniki

Wskaźniki są to zmienne, które nie przechowują wartości danego typu, tylko przechowują adresy do tych zmiennych. Za pomocą wskaźników można robić szereg operacji, których nie można by zrobić za pomocą zwykłych zmiennych, ale o tym trochę później.

Deklaracja wskaźnika polega na dodaniu gwiazdki pomiędzy typem a nazwą zmiennej. Poniżej zostały pokazane trzy równoważne deklaracje wskaźników. Nie ma różnicy czy gwiazdka będzie bezpośrednio przed nazwą, bezpośrednio po typie, czy „po środku” kwestia przyzwyczajeń, ja osobiście preferuję sposób pierwszy. Podczas deklarowania kilku zmiennych oraz wskaźników w jednej linii (listing) sposób pierwszy wydaje się najbardziej intuicyjny.

```
int *wsk_i;                // wskaznik na typ int

double* wsk_d;             // wskaznik na typ double

char * wsk_i;              // wskaznik na typ char
```

Jeśli już stworzyliśmy wskaźnik do konkretnego typu, to teraz wypadałoby przypisać adres jakiejś zmiennej, do tego wskaźnika. Poniżej zaprezentowany został listing z użytecznymi właściwościami wskaźników. Omówienie tych właściwości znajduje się poniżej.

```
#include <stdio.h>

main (void)
{
     int x = 10, y, *wsk_x, *w2;
     wsk_x = &x;
     w2 = wsk_x;

     printf("wsk:\t%p\t%d\n", wsk_x, *wsk_x);
     printf("x:  \t%p\t%d\n", &x, x);

     *wsk_x = 15;
     y = x;
     printf("x: \t%p\t%d\n", &x, x);
     printf("y: \t%p\t%d\n", &y, y);

     *wsk_x += 10;
     printf("x:  \t%p\t%d\n", &x, x);

     ++*wsk_x;
     printf("wsk:  \t%p\t%d\n", wsk_x, *wsk_x);
```

```
    (*wsk_x)++;
    printf("x:  \t%p\t%d\n", wsk_x, *wsk_x);

    *wsk_x++;
    printf("x:  \t%p\t%d\n", wsk_x, *wsk_x);
    printf("w2: \t%p\t%d\n", w2, *w2);
    return 0;
}
```

Listing 5.2.1 Użycie wskaźnika i pewne operacje

Stworzyliśmy zmienną typu całkowitego z zainicjowaną wartością, zmienną bez zainicjowanej wartości oraz dwa wskaźniki do typu całkowitego. Do zmiennej wskaźnikowej musimy przypisać adres. Tak więc za pomocą operatora adresu **&** wyciągamy adres ze zmiennej stojącej po prawej stronie tego operatora[1] i przypisujemy ten adres do wsk_x. Jeśli dwie zmienne są typu wskaźnikowego to możemy przypisać jedną wartość do drugiej. Tak więc w2 będzie wskazywała na to co wsk_x, czyli na x. Jak widać na listingu 5.2.1 pierwsze dwie instrukcje printf wydrukują dokładnie to samo, ponieważ zmienna wsk_x wskazuje na zmienną x. Obie zmienne odnoszą się do tego samego miejsca w pamięci, co widać po adresie, który zostanie wydrukowany. Różnica polega na odwołaniu się do zmiennych wskaźnikowych. Argumenty drugi i trzeci z drugiej instrukcji printf są właściwie znane. Przed chwila było powiedziane, że jeśli znak **&** stoi po lewej stronie zmiennej to nie jest to wartość zmiennej, a jej adres. Natomiast w pierwszej instrukcji jest trochę inaczej, ponieważ zmienna wsk_x jest zmienną wskaźnikową, czyli wartością, którą przechowuje jest adres, to po prostu w miejscu w którym ma zostać wydrukowany adres wpisujemy tylko tę nazwę. Jeśli chcemy zmienić wartość kryjącą się pod zmienną x, lub po prostu odwołać się do niej, ale za pomocą zmiennej wskaźnikowej to musimy tę wartość wyciągnąć spod tego adresu. Do tego celu używamy operatora wyłuskania (lub operator dereferencji), który jest gwiazdką stojącą po lewej stronie zmiennej wskaźnikowej. Po zmianie wartości z użyciem wsk_x, zmienna x będzie posiadała nową wartość, dlatego też zmienna y będzie posiadała nową wartość x. Używając operatora wyłuskania mamy dostęp do wartości zmiennej, tak więc możemy wykonywać wszystkie operacje jakie można było wykonywać na zwykłych zmiennych. Operacja ++*wsk_x zwiększy o 1 wartość, która kryje się pod adresem, który przechowuje ta zmienna (w naszym przypadku x). Operator post – inkrementacji też można zastosować, natomiast trzeba użyć nawiasów, bo w przeciwnym wypadku zwiększymy adres wskaźnika. Tak też się stało w kolejnej instrukcji i adres będzie zwiększony o **1 ▪ rozmiar typu**, a wartością będą śmieci.

1 Nie należy mylić jednoargumentowego operatora adresu & z dwuargumetnowym operatorem bitowej koniunkcji &.

5.3 Przekazywanie adresu do funkcji

Jak wiadomo parametry przekazywane funkcjom przekazują im tylko swoje wartości. W funkcji tworzone są lokalne zmienne i na nich wykonuje się operacje, a na końcu zwraca się wynik operacji do miejsca wywołania. Za pomocą wskaźników możemy manipulować wartościami przekazywanymi do funkcji, tzn do tej pory wartości zmiennych z funkcji wywołującej nie mogły ulec zmianie. Za pomocą wskaźników istnieje taka możliwość i pokaże na przykładzie pewne zastosowanie tego mechanizmu. Weźmy na przykład funkcję fabs ze standardowej biblioteki, która zwraca wartość bezwzględną liczby podanej jako argument. Funkcja ta zwraca wartość bezwzględną, lecz nie zmienia wartości swojego argumentu, co zostało pokazane na poniższym listingu.

```
#include <stdio.h>
#include <math.h>
main (void)
{
	int x = -5;
	printf("%.0f\n", fabs(x));
	printf("%d\n", x);
	return 0;
}
```

Listing 5.3.1 Użycie fabs

Napiszemy teraz funkcję pobierającą adres obiektu (zmiennej). W funkcji tej będą wykonywane operacje bezpośrednio na zmiennej, która znajduje się w funkcji wywołującej, co za tym idzie, funkcja abs2 zmieni wartość swojego argumentu.

```
#include <stdio.h>
void abs2 (int *);
main (void)
{
	int x = -5;
	printf("%d\n", x);
	abs2(&x);
	printf("%d\n", x);
	return 0;
}

void abs2 (int *arg)
{
	if (*arg < 0)
		*arg = -*arg;
}
```

Listing 5.3.2 Przekazywanie adresu do funkcji

W funkcji abs2 używamy operatora wyłuskania do sprawdzenia czy kryjąca się wartość pod przekazanym adresem jest mniejsza od zera i jeśli tak jest, to zmieniamy jej znak na przeciwny. Po wywołaniu tej funkcji zmienna x w funkcji main zmieni wartość na dodatnią, jeśli uprzednio było ujemna. Na listingu 5.3.1 tylko wydrukowaliśmy wartość dodatnią, natomiast wartość zmiennej nie uległa zmianie, bo funkcja nie miała bezpośredniego dostępu do jej adresu. A więc zapamiętać należy jedno, jeśli używamy wskaźników, to trzeba używać ich mądrze, bo można zmienić wartości, których zmienić nie chcieliśmy. Dzięki poprzedniemu przykładowi wiemy już, że możemy zmieniać wartości kilku zmiennych po wywołaniu jednej funkcji. Za pomocą return mogliśmy zwrócić jedną wartość.

W rozdziale 2.2.4 dotyczącym stałych powiedziane było, że stałej nie można zmienić, a jednak za pomocą wskaźnika da się to zrobić, potraktuj ten przykład jako ciekawostkę.

```
#include <stdio.h>
void cConst (int *);
int main (void)
{
      const int x = 10;
      printf("%d\n", x);
      cConst((int *)&x);
      printf("%d\n", x);
      return 0;
}

void cConst (int *arg)
{
      *arg = 4;
}
```

Listing 5.3.3 Zmiana wartości const za pomocą wskaźnika

Jako argument wywołania funkcji podajemy (int *)&x, dokładnie oznacza to tyle, że adres zmiennej x, która jest typu const int rzutujemy na int *, dzięki temu możemy zmienić tę wartość. Jeśli nie było by rzutowania, to kompilator wyświetli informację, że przekazujemy zły typ danych. Z kolei jeśli parametrem funkcji byłby const int *, to kompilator wyświetli błąd z informacją o tym, że nie można zmieniać wartości zmiennych tylko do odczytu.

5.4 Zależności między tablicami, a wskaźnikami

Tablice i wskaźniki są dość mocno powiązane ze sobą. Może nie widać tego na pierwszy rzut oka, lecz

jak przyjrzymy się bliżej strukturze pamięci wypełnionej przez tablicę oraz zauważymy pewne właściwości wskaźników, to dostrzeżemy to podobieństwo. Poniższy schemat pamięci może okazać się pomocny. Tablica tab jest typu int, dlatego też każdy element tablicy zajmuje po 4 bajty, co łatwo można sprawdzić sprawdzając róznicę pomiędzy adresami.

	tab[0]	tab[1]	tab[2]	tab[3]	tab[4]	
…	0xbfeb0390	0xbfeb0394	0xbfeb0398	0xbfeb039c	0xbfeb03a0	…

Widzimy, że tablica tab zajmuje 5 komórek pamięci, z czego tab[0] zaczyna się na adresie 0xbfeb0390 a kończy się na początku tab[1] czyli 0xbfeb0394. tab[4] zaczyna się od 0xbfeb03a0, a kończy się na 0xbfeb03a4 (nie pokazano). Adresy są liczbami szesnastkowymi, więc możemy sprawdzić ile ta tablica zajmuje bajtów. 0Xbfeb03a4 – 0xbfeb0390 = 14 – wynik otrzymaliśmy w systemie szesnastkowym. 14 szesnastkowo to 20 dziesiętnie, tak więc nasza tablica zajmuje 20 bajtów.

W tym miejscu warto wspomnieć o operatorze sizeof, który zwraca ilość bajtów zajmowanych przez argument. Tak więc jeśli chcemy sprawdzić ile na naszej maszynie zajmuje typ int wystarczy wpisać poniższą instrukcję w ciele funkcji main.

```
printf("%d\n", sizeof (int));
```

Operator sizeof zwraca wartość całkowitą zajmowanego miejsca przez dany typ, zmienną, tablicę, itp. A skoro jesteśmy już przy możliwości sprawdzenia ile zajmuje tablica, to czemu by tego nie wkorzystać? Wpisz powyższą instrukcję zamieniając argument operatora sizeof z int na tab (nazwa naszej tablicy) by przekonać się, że nasza tablica faktycznie zajmuje 20 bajtów. Możemy zrobić coś więcej, w punkcie 5.1 powiedziane było, że nie trzeba wpisywać rozmiaru tablicy, wystarczy ją zainicjować dowolnymi wartościami a kompilator sam ustali rozmiar. Owszem, tak można zrobić, ale skoro nie wiemy ile jest elementów tablicy, to jak mamy te dane np. wyświetlić? Jaką wartość wpisać do pętli for jako ograniczenie, by nie przekroczyć ilości elementów tablicy? Odpowiedź jest właściwie prosta – instrukcja sizeof. Poniższy listing będzie tego dowodem.

```
#include <stdio.h>

main (void)
{
     int tab[] = {8, 4, 3, 9, 1, 1, 23, 2, 11, 23};
     int i;
```

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

Listing 5.4.1 Użycie sizeof

Wywołanie sizeof(tab) zwraca ilość bajtów zajmowanych przez tablicę tab, a sizeof (int) zwraca ilość zajmowanych bajtów przez typ int, czyli przez jeden element tablicy. Iloraz tych dwóch wartości daje liczbę elementów tablicy. Bezpieczniejszym sposobem może okazać się sizeof (tab) / sizeof (tab[0]), ponieważ jeśli zmienimy typ tablicy np. na double to ilorazem byłby rozmiar tablicy double i rozmiar typu int, a to nie była by ilość elementów tej tablicy. sizeof(tab[0]) podaje rozmiar pojedynczego elementu, który jest takiego samego typu jak tablica.

Tworząc tablicę rezerwujemy pewną ilość miejsca w pamięci począwszy od jakiegoś adresu, co widać na pomocniczym schemacie z początku tego rozdziału. Powiedziane było w punkcie 5.2, że za pomocą operatora wyłuskania można dodać wartość, lub wykonać inną operację na wartości, kryjącej się pod danym adresem. Operacje można wykonywać również na adresach. Możemy dodać lub odjąć „jeden" do adresu. „Jeden" to nie jeden bajt, tylko **1 · rozmiar typu**. Czyli jeśli nasza tablica jest typu int, to dodanie jeden do wskaźnika przesunie go o 1 · sizeof (int), co zresztą widać po różnicy między komórkami.

Tablicę można traktować jako wskaźnik do zarezerwowanej pamięci, który zaczyna się od pewnego miejsca – tab[0]. Równoważną postacią do tab[0] jest podane samej nazwy tablicy, w naszym przypadku tab, ponieważ domyślnie nazwa tablicy wskazuje na pierwszy jej element. Jeśli teraz dodamy do tab (czyli adresu pierwszego elementu) jeden, to przesuniemy wskaźnik na następny element tablicy, czyli tab[1]. A żeby uzyskać wartość elementu kryjącego się pod adresem tab+1 musimy użyć poznanego już wcześniej operatora wyłuskania i zastosować go w następujący sposób *(tab+1). Ta postać jest alternatywą dla tab[1]. Poniższy listing pokazuje alternatywne wyświetlanie elementów tablicy.

```
#include <stdio.h>
main (void)
{
     int tab[] = {8, 4, 3, 9, 1, 1, 23, 2, 11, 23};
     int i;
```

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

Listing 5.4.2 Alternatywne wyświetlanie elementów tablicy

Teraz możemy zrobić pewnie dość dziwną, może nie zrozumiałą rzecz, ale zadeklarujemy wskaźnik, który będzie wskazywał na pierwszy element tablicy i za pomocą tego wskaźnika będziemy się odwoływać do elementów tablicy.

```
#include <stdio.h>
main (void)
{
	int tab[] = {8, 4, 3, 9, 1, 1, 23, 2, 11, 23};
	int i;
	int *wsk;
	wsk = tab; /* wsk = &tab[0] */
	for (i = 0; i < sizeof (tab) / sizeof (tab[0]); i++)
		printf("%d ", *(wsk+i));
	printf("\n");
	return 0;
}
```

Listing 5.4.3 Wyświetlanie elementów tablicy za pomocą wskaźnika

Przypisujemy do wskaźnika adres pierwszego elementu tablicy. W zmiennej wskaźnikowej wsk mamy już adres początku tablicy, więc jak było już powiedziane dodanie do adresu jeden przesuwa wskaźnik o rozmiar elementu, tak więc za pomocą wskaźnika wsk możemy przesuwać się na adresy poszczególnych elementów tablicy i wyświetlać ich wartości. Jeszcze jedna uwaga, wyrażenie tab+i (wsk+i) przesuwa wskaźnik o i · sizeof (int) względem początkowego adresu, lecz nie zapisuje nowej pozycji w zmiennej tab (wsk).

Należy pamiętać o jednej bardzo ważnej rzeczy, wskaźnik jest zmienną, więc może wskazywać na cokolwiek, włączając w to adres początku tablicy. Natomiast nazwa tablicy zmienną nie jest, więc pewne operacje są niedozwolone. Poniższy listing prezentuje te różnice.

```
#include <stdio.h>
#define SIZE 5
main (void)
{
	int tab[SIZE] = {3, 4, 5, 1, 2};
```

```
	int i, *wsk;
	wsk = tab;

	for (i = 0; i < SIZE; i++)
		printf("%d ", *(wsk+i));
	printf("Wskaznik jest pod adresem: %p\n", wsk);

	for (i = 0; i < SIZE; i++)
		printf("%d ", *(tab+i));
	printf("Wskaznik jest pod adresem: %p\n", tab);

	for (i = 0; i < SIZE; i++)
		printf("%d ", *wsk++);
	printf("Wskaznik jest pod adresem: %p\n", wsk);
/*
	for (i = 0; i < SIZE; i++)
		printf("%d ", *tab++);		// Blad
	printf("Wskaznik jest pod adresem: %p\n", tab);
*/
	return 0;
}
```

Listing 5.4.4 Róznicy między zmienną wskaźnikową a nazwą tablicy

We wszystkich pętlach drukujemy zawartość tablicy. Różnicą jest pozycja wskaźnika. Po zakończeniu pierwszej pętli for wskaźnik będzie dalej na pierwszym elemencie tablicy, ponieważ wyświetlając nie zmienialiśmy jego wartości, tylko wyświetlaliśmy wartość przesuniętą o i względem punktu bazowego. Działanie drugiej pętli for jest analogiczne – używamy nie wskaźnika, a nazwy tablicy więc możemy tak samo wyświetlać element przesunięty o i względem pierwszego elementu tablicy. Trzecia pętla poza tym, że wyświetla elementy tablicy przesuwa wskaźnik po każdej iteracji o 1 (w ramach przypomnienia wsk++ jest równoważne z wsk = wsk + 1). Tak więc po zakończeniu pętli wskaźnik znajdzie się na pozycji, która już nie należy do tablicy tab. W czwartej pętli konstrukcja *tab++ jest niedozwolona, tak więc jeśli usuniemy znaki komentarza i spróbujemy skompilować program dostaniemy błąd mówiący o tym, że L-wartość[1] (*ang. L-value*) wymaga operatora, który może być zwiększony. Tak samo przypisania do tablicy takie jak tab = i, tudzież tab = &i są niedozwolone.

5.5 Operacje na wskaźnikach

Istnieją pewne operacje na wskaźnikach, które są dozwolone i takie, których używać nie można,

1 Jednostka stojąca po lewej stronie operatora przypisania

dlatego też w tym miejscu opiszę te, którymi możemy się posługiwać. Do dozwolonych operacji na wskaźnikach zaliczamy:

- Przypisanie wskaźników do obiektów tego samego typu
- Dodawanie i odejmowanie wskaźnika i liczby całkowitej
- Odejmowanie i porównywanie dwóch wskaźników do elementów tej samej tablicy
- Przypisanie do wskaźnika wartości zero
- Porównanie wskaźnika z zerem

Pozostałe operacje na wskaźnikach wliczając w to dodawanie, mnożenie i dzielenie dwóch wskaźników, oraz dodawanie liczb rzeczywistych do wskaźników są niedozwolone. Nie można też przypisywać do wskaźnika jednego typu adresu zmiennej innego typu (wyjątkiem jest void *, który jest opisany w następnym punkcie). Poniżej znajdują się przykłady wyżej wymienionych operacji. Podpunkt drugi zrealizowaliśmy już, chociażby na listingu 5.4.4. Reszta podpunktów została pokazana poniżej.

```
#include <stdio.h>
#define MAX 100
main (void)
{
	int tab[MAX];
	int *w1, *w2, *w3;
	w1 = &tab[MAX-1];
	w2 = tab;

	if (w2 > w1)
		printf("w2 jest wskaznikiem do dalszego elementu\n");
	else
	{
		printf("w2 jest wskaznikiem do blizszego elementu\n");
		w3 = NULL;
	}
	if (w3 == NULL)
		printf("Roznica wskaznikow: %d\n", (w1-w2));
	return 0;
}
```

Listing 5.5.1 Użycie operacji na wskaźnikach

Do wartości wskaźnika w3 moglibyśmy przypisać zero i sprawdzić w warunku czy w3 równa się zero. Nie mniej jednak utworzono specjalną stałą symboliczną NULL (która ma wartość 0) zdefiniowaną w nagłówku stdio.h w celu podkreślenia, że chodzi o specjalną wartość wskaźnika. 0 i NULL mogą być stosowane zamiennie. Niektóre funkcje zwracają wartość NULL.

5.6 Wskaźnik typu void

Wskaźnik typu void może wskazywać na dowolny obiekt (dowolny typ), natomiast nie można użyć operatora wyłuskania do niego. Poniżej znajduje się przykład, który pokazuje możliwości typu void.

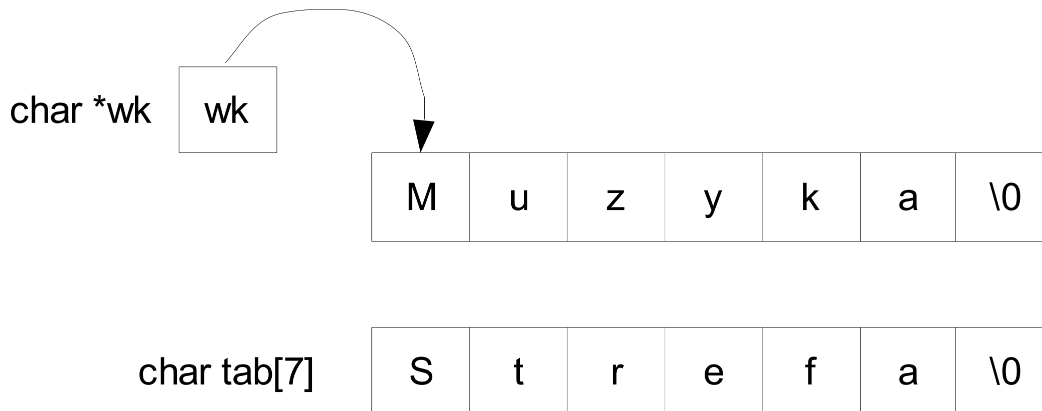
```
#include <stdio.h>

main (void)
{
	int i = 10, *wsk_i;
	double k = 15.5, *wsk_d;
	void *x;

	x = &i;
	printf("x: \t%p\n", x);
	// printf("x: %d\n", *x);

	wsk_i = x;
	printf("wsk_i: \t%p\t%d\n", wsk_i, *wsk_i);

	x = &k;
	printf("x: \t%p\n", x);
	// printf("x: %f\n", *x);

	wsk_d = x;
	printf("wsk_d: \t%p\t%f\n", wsk_d, *wsk_d);
	return 0;
}
```

Listing 5.6.1 Wskaźnik typu void

Jak widać do wskaźnika typu void można przypisać zarówno adres zmiennej typu int, jak i typu double. Natomiast nie możemy się odwołać do wartości – kompilator zaprotestuje. Jeśli wskaźnik typu void wskazuje na typ int, to możemy ten adres przypisać do wskaźnika typu int i wartość kryjącą się pod wskazywanym adresem wyświetlić. To samo obowiązuje dla typu double i innych.

5.7 Tablice znakowe

Typem tablic znakowych jest typ char. Tablice znakowe służą do przechowywania stałych napisowych. Przechowywać stałe napisowe można na dwa sposoby, albo za pomocą tablicy znaków, albo za pomocą wskaźnika. Różnica między nimi istnieje i na poniższym rysunku postram się ją wytłumaczyć.



Rysunek 5.7.1 Różnice między wskaźnikiem a tablicą

A teraz dla pełnego zrozumienia dwie deklaracje zmiennych z rysunku 5.7.1

```
char *wk = "Muzyka";
char tab[7] = "Strefa";
```

Zarówno wskaźnik wk jak i tablica tab wskazują na napis, który składa się z takiej samej ilości znaków. Wyobraźmy sobie to w sposób następujący. Podczas kompilacji programu, stała napisowa „Muzyka" zapisywana jest gdzieś w pamięci, a do wskaźnika wk zostaje przypisany adres pierwszego znaku, czyli adres litery M. Tekst jest odczytywany (np. przez funkcję printf) aż do znaku kończącego stałą napisową **\0**. Tablica tab podczas kompilacji uzupełniana jest znakami wyrazu „Strefa". Różnice między tymi dwiema formami zapisu są następujące:

- Elementów słowa „Muzyka" nie można zmieniać!
- Wskaźnik wk może w późniejszym czasie wskazywać na coś innego

- Elementy tablicy można zmieniać dowolnie
- Nazwa tab zawsze będzie odnosiła się do tablicy.

Zanim pokazany zostanie listing prezentujący różnice między wskaźnikiem, a tablicą znaków chciałbym jeszcze powiedzieć trochę właśnie o tablicy znaków. Tablica znaków podobnie jak tablica liczb całkowitych czy rzeczywistych posiada rozmiar i elementy. Elementy te może wpisywać normalnie tak jak wpisywaliśmy liczby, tylko, że litery ujmujemy w znaki apostrofu, lub wpisujemy w cudzysłowie ciąg znaków, co jest zdecydowanie łatwiejszym i szybszym sposobem. Należy pamiętać o ważnej rzeczy, jeśli wpisujemy znaki pojedyńczo oddzielone przecinkami, to musimy wstawić znak **\0** na ostatniej pozycji, tego nie trzeba robić jeśli wpisujemy ciąg znaków w cudzysłowie (podczas kompilacji znak ten zostanie dostawiony automatycznie).

Jeśli inicjujemy tablicę jakimiś znakami, to rozmiar nie musi być wpisany, ponieważ jest obliczany na etapie kompilacji. Najważniejszą rzeczą o której trzeba pamiętać jest fakt, że rozmiar tablicy to ilość znaków + 1 (**\0**). Czyli tak jak na naszym rysunku 5.7.1 tablica tab zajmuje siedem bajtów, bo ilość znaków stałej napisowej to sześć oraz jeden bajt na znak **\0**. Na poniższym listingu pokazane zostały różnice o których była mowa.

```
#include <stdio.h>

main (void)
{
      int i;
      char *wk = "Muzyka";
      char tab[7] = "Strefa";
      char tab2[7] = {'S', 't', 'r', 'e', 'f', 'a', '\0'};

      printf("wk: \t\t%s\n", wk);
      printf("tab: \t\t%s\n", tab);
      printf("tab2: \t\t%s\n", tab2);

      printf("wk[1]: \t\t%c\n", wk[1]);
      // wk[1] = 'U';       // Segmentation fault

      wk = tab;
      printf("wk: \t\t%s\n", wk);

      for (i = 0; i < sizeof (tab)/sizeof (tab[0]); i++)
            if (tab[i] == 'e')
                  tab[i] = 'E';
```

```
	printf("tab: \t\t%s\n", tab);
	// tab = wk;		// Blad
	return 0;
}
```

Listing 5.7.1 Różnice między wskaźnikiem a tablicą

Na powyższym listingu tab oraz tab2 posiadają taki sam ciąg znaków, aczkolwiek zapisane są inaczej. Można wyświetlić niektóre elementy stałej znakowej „Muzyka” wskazywanej przez wk, natomiast nie można ich zmienić. Wskaźnik wk, jeśli już nie jest potrzebny do wyświetlania słowa „Muzyka” może wskazywać na coś innego (wk = tab; wk wskazuje na pierwszy element tablicy tab). Tak jak było powiedziane, elementy tablicy można zmieniać, realizuje to nasza pętla w której przeszukujemy tablicę w celu znalezienia litery **e** i zastąpienia jej przez jej wielki odpowiednik. tab zawsze będzie wskazywać na tablicę, nie można tego zmienić.

Ciekawostką może być fakt, że znaki będące w tablicy znaków można by było wyświetlić za pomocą pętli, ale istnieje deskryptor formatu **%s**, który wyświetla znaki tablicy i kończy swoje działanie na znaku \0. Dla przykładu pokazano jeszcze jeden listing.

```
#include <stdio.h>
main (void)
{
	char *wsk = "Ala ma kota";
	char tab[] = "Lorem ipsum";
	int i, k, z;
	k = sizeof (tab) / sizeof (char);
	printf("Rozmiar tablicy: \t\t%d\n", sizeof (tab));
	printf("Rozmiar typu char: \t\t%d\n", sizeof (char));
	printf("Ilosc elementow: \t\t%d\n", k);
	printf("Zawartosc tablicy: \t\t");
	for (i = 0; i < k-1; i++)
		printf("%c", tab[i]);
	printf("\n\n");

	z = sizeof (wsk)/sizeof (char);
	printf("Rozmiar wskaznika: \t\t%d\n", sizeof (wsk));
	printf("Rozmiar typu char: \t\t%d\n", sizeof (char));
	printf("Ilosc elementow: \t\t%d\n", z);
	printf("Tekst wskazywany: \t\t");
	for (i = 0; wsk[i] != '\0'; i++)
		printf("%c", wsk[i]);
	printf("\n");
	return 0;
}
```

Listing 5.7.2 Rozmiar wskaźnika

Na powyższym przykładzie może nie ma nowych rzeczy, aczkolwiek jest jedna bardzo ważna rzecz, która przyda nam się w kolejnym punkcie. A mianowicie rozmiar wskaźnika. Rozmiar tablicy obliczany jest jako **ilość elementów · rozmiar typu tablicy**. Czyli wyświetliliśmy rozmiar tablicy tab, rozmiar typu char, a ilorazem ich jest ilość elementów tablicy. Skoro mamy rozmiar – wiemy ile jest znaków w tablicy – to możemy je wszystkie wydrukować. Ponieważ elementów w tablicy jest 12 (11 znaków bez **0**), a tablice indeksuje się od zera, czyli nasze drukowalne znaki są na pozycjach od 0 do 10 to warunek pętli jest i < k-1, czyli i < 11, czyli łącznie wydrukowane zostaną znaki od początku do dziesiątego włącznie. Z rozmiarem wskaźnika jest inaczej i tutaj mogą pojawić się pewne kłopoty. Nie jesteśmy w stanie powiedzieć ile elementów ma wskazywany tekst, ponieważ sizeof (wsk) pokazuje tylko rozmiar wskaźnika. Ale dzięki znakowi kończącemu stałą napisową jesteśmy w stanie wydrukować wszystkie znaki, a to zostało pokazane w drugiej pętli for, jeśli znak jest różny od **0** to go wydrukuj.

5.8 Przekazywanie tablicy do funkcji

Z punktu widzenia programu nie ma to znaczenia, czy do funkcji przekazujemy wskaźnik, tablicę, czy część tablicy. Niech naszą funkcją będzie funkcja duze, która zamienia wielkość liter swojego argumentu z małych na duże.

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

void duze (char *tab);

main (void)
{
	char tab1[] = "ala ma kota";
	char tab2[] = "lorem ipsum";
	char *napis = "ala nie ma kota";
	char *wsk;
	wsk = tab2;
	printf("%s\n", tab1);
	duze(tab1);
	printf("%s\n", tab1);

	printf("%s\n", wsk);
	duze(&wsk[5]);
	printf("%s\n", wsk);
	//duze(napis);
```

```
	return 0;
}

void duze (char *tab)
{
	while (*tab = toupper(*tab))
		tab++;
}
```

Listing 5.8.1 Przekazywanie tablicy do funkcji

Jak powiedziano w poprzednich rozdziałach tablice i wskaźniki mają bardzo wiele wspólnego, tak więc nie ma najmniejszego znaczenia czy funkcja duze przyjmuje tablicę, czy wskaźnik jako argument. Jeśli zapisalibyśmy, że przyjmuje tablicę to prototyp funkcji wyglądałby tak:

```
void duze (char tab[]);
```

A w funkcji moglibyśmy zostawic to co jest, lub zapisać to w następujący sposób:

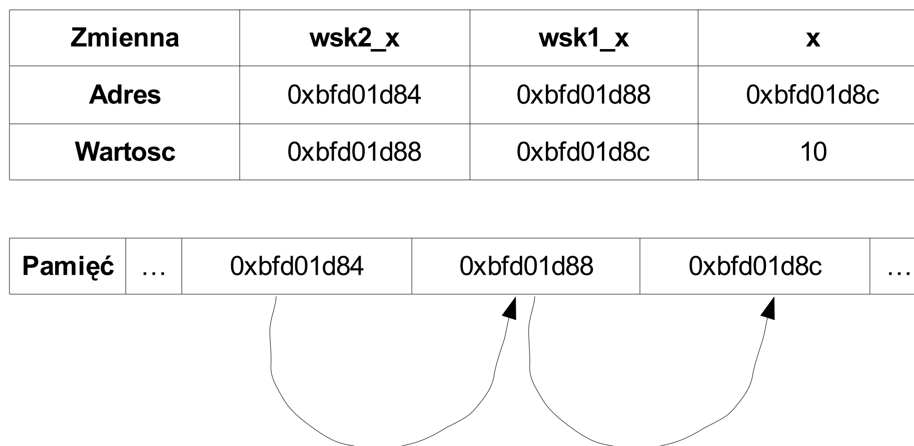
```
void duze (char tab[])
{
	int i = 0;
	while (tab[i] = toupper(tab[i]))
		i++;
}
```

Do funkcji duze przekazujemy adres tablicy znaków, nie ma różnicy czy przekazujemy całą tablicę czy tylko jej fragment. W pętli while zmieniamy wielkość znaku kryjącego się pod przekazanym adresem za pomocą funkcji toupper, jeśli znakiem tym nie był znak \0, a następnie zwiększamy adres i wykonujemy kolejną zmianę znaku. Jeśli znakiem jest znak końca tablicy, to wyrażenie while (0) jest fałszywe, a więc funkcja duze zakończy swoje działanie. Znak \0 w wyrażeniach warunkowych traktowany jest jak zero. Wskaźnikowi wsk przypisaliśmy adres pierwszego elementu tablicy tab2, tak więc używanie wsk odnosi się tak naprawdę do tablicy tab2. Przekazaliśmy część tablicy tab2 za pomocą wskaźnika deklaracją &wsk[5]. Czyli przekazujemy adres szóstego elementu i od tego elementu robiona jest operacja zamiany znaków, więc końcowym efektem będzie duży wyraz IPSUM, a pierwszy wyraz bez zmiany. Wywołanie funkcji duze z argumentem napis ujęte jest w komentarz, ponieważ jak wspomniano już nie można zmienić zawartości pamięci na którą wskazuje ten wskaźnik. W przeciwnym wypadku będzie błąd **segmentation fault**, który mówi o tym, że program chce odwołać się do pamięci do której nie ma dostępu. Nie którzy mogli by się zdziwić dlaczego przekazując do

funkcji duze tablicę możemy wykonać operację tab++; skoro była mowa w punkcie 5.4, że tak nie można. Tutaj ukłon w stronę funkcji i zmiennych lokalnych. Jeśli zmienna jest zadeklarowana jako parametr funkcji w ten sposób char *tab tzn, że wskaźnik na typ znakowy jest lokalny. Czyli przekazując do funkcji tablicę, czyli tak naprawdę adres jej pierwszej komórki zapisujemy ten adres w zmiennej lokalnej tab, na której już możemy wykonywać dozwolone operacje na wskaźnikach.

5.9 Wskaźniki do wskaźników

Jak już powiedziano wskaźniki to zmienne przetrzymujące adres innej zmiennej, ale jak każda zmienna tak również i wskaźnik musi być zapisany gdzieś w pamięci, czyli posiada swój adres. Można utworzyć inny wskaźnik, który będzie odwoływał się do adresu tego wskaźnika. Rysunek z kolejnej strony może okazać się pomocny.



Zmienna	wsk2_x	wsk1_x	x
Adres	0xbfd01d84	0xbfd01d88	0xbfd01d8c
Wartosc	0xbfd01d88	0xbfd01d8c	10

Pamięć	…	0xbfd01d84	0xbfd01d88	0xbfd01d8c	…

Rysunek 5.9.1 Ilustracja pamięci i zawartości wskaźników

Na rysunku zostało pokazane, że zmienna x posiada wartość 10 i jej adresem jest 0xbfd01d8c. Wskaźnik wsk1_x wskazuje na adres zmiennej x, czyli jako wartość ma zapisany adres zmiennej x, oraz posiada swój adres. Wskaźnik wsk2_x również posiada swój adres, a jako wartość ma wpisany adres wskaźnika wsk1_x. Listing poniższy pokazuje jak deklarować wskaźniki do wskaźników.

```
#include <stdio.h>
main (void)
{
```

```
	int x = 10;
	int *wsk1_x = &x;
	//wsk1_x = &x;
	int **wsk2_x = &wsk1_x;
	//wsk2_x = &wsk1_x;
	printf("x: \t\t%p\t%d\n", &x, x);
	printf("wsk1_x: \t%p\t%p\t%d\n", &wsk1_x, wsk1_x, *wsk1_x);
	printf("wsk2_x: \t%p\t%p\t%p\t%d\n", &wsk2_x, wsk2_x,
*wsk2_x,**wsk2_x);
	return 0;
}
```

Listing 5.9.1 Wskaźnik do wskaźnika

Najciekawszą linią może być ostatnia, w której instrukcja printf pobiera pięć argumentów. Zacznijmy od tego co zostanie przekazane do wyświetlenia.

- &wsk2_x – Adres wskaźnika wsk2_x
- wsk2_x – Adres wskaźnika wsk1_x
- *wsk2_x – Wyciągamy wartość kryjącą się pod wskaźnikiem wsk1_x – Adres zmiennej x
- **wsk2_x – Z wyżej wyciągniętego adresu wyciągamy wartość – 10

5.10 Tablica wskaźników

Jak już wspomniano wskaźniki to też zmienne, dlatego można utworzyć z nich tablicę do przechowywania większej ilości adresów tego samego typu. Sposób definiowania jest bardzo podobny do definicji pojedynczego wskaźnika z tą różnicą, że dodajemy rozmiar po nazwie. Tak więc dziesięcio elementowa tablica wskaźników na typ char wygląda następująco.

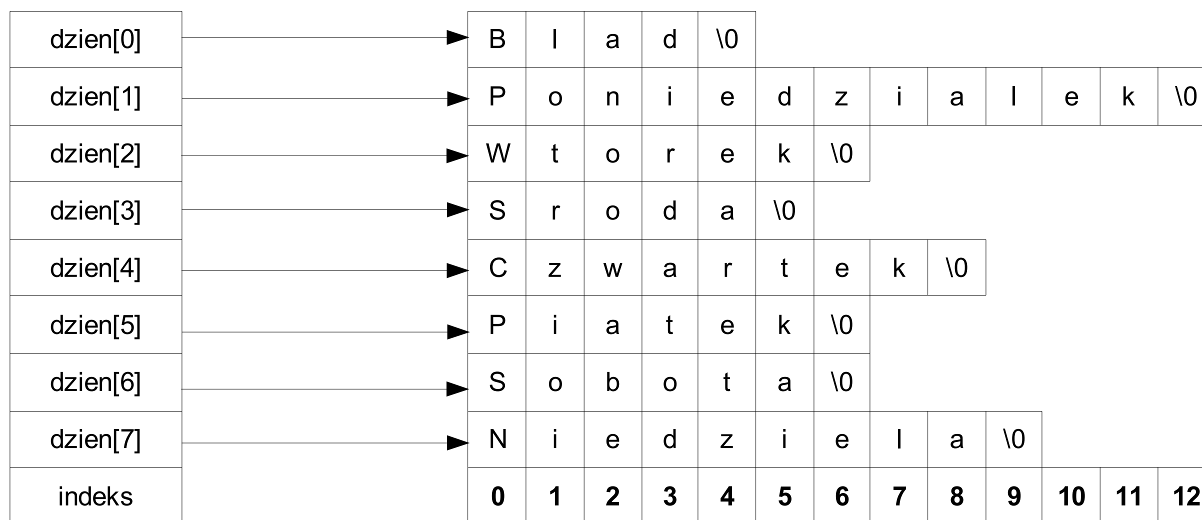
```
char *tabWsk[10];
```

Istnieje możliwość inicjowania tablic wskaźników i to chciałbym pokazać na poniższym przykładzie. Rysunek do zadania może wyjaśnić pewne nie jasności.

```
#include <stdio.h>

char *nazwa_dnia (int n);

int main (void)
```

```
{
      int nrDnia;
      char *wskDzien;
      printf("Podaj nr dnia: ");
      scanf("%d", &nrDnia);
      wskDzien = nazwa_dnia(nrDnia);
      printf("%s\n", wskDzien);

      return 0;
}
char *nazwa_dnia (int n)
{
      char *dzien[] = {
            "Blad: Nie ma takiego dnia",
            "Poniedzialek",
            "Wtorek",
            "Sroda",
            "Czwartek",
            "Piatek",
            "Sobota",
            "Niedziela",
      };
      return (n < 1 || n > 7) ? dzien[0] : dzien[n];
}
```

Listing 5.10.1 Tablica wskaźników



Rysunek 5.10.1 Miejsca, na które wskazują wskaźniki

A teraz pewne wyjaśnienie co do tego programu. Funcja nazwa_dnia zwraca wskaźnik do typu znakowego, a jako argument przyjmuje liczbę całkowitą co widać po deklaracji. W ciele tej funkcji jest zainicjowana tablica wskaźników do typu znakowego. Można by sobie to wyobrazić jak tablicę dwu wymiarową z czego każdy element jest napisem. Łatwiej jest zapisać to jako tablicę wskaźników, ponieważ rozmiar napisów jest różny. W funkcji main deklarujemy zmienną typu całkowitego do przechowywania liczby pobranej od użytkownika oraz wskaźnik na typ char – nasza funkcja zwraca taki wskaźnik, więc do tej zmiennej można go przypisać. Po reakcji użytkownika do funkcji przekazywana jest liczba i na podstawie warunku w instrukcji return zwracany jest odpowiedni wskaźnik do zerowego indeksu (pierwszy element) pewnego ciągu znaków. Jeśli liczba n jest spoza zakresu to zwracany jest wskaźnik do litery **B**, która występuje pod zerowym elementem tablicy dzien na pozycji zero (czyli dzien[0][0] to litera **B**). Jeśli liczba n jest z dozwolonego zakresu to zwracany jest adres do pierwszej litery (pierwszy element) n-tego elementu. Jak powiedziane było w poprzednich rozdziałach funkcja printf wyświetla znaki, aż osiągnie znak **\0**.

W instrukcji printf można wstawić bezpośrednio odwołanie do funkcji, czyli linijka

```
printf("%s\n", wskDzien);
```

mogłaby zostać zamieniona na

```
printf("%s\n", nazwa_dnia(nrDnia));
```

O jeszcze jednej sprawie trzeba napisać. Tablica dzien w funkcji nazwa_dnia zadeklarowana jest jako tablica wskaźników, każdy ze wskaźników może wskazywać na różny ciąg znaków. Rozmiarem tej tablicy wskaźników będzie **rozmiar typu wskaźnika · ilość elementów**, czyli sizeof (char *) · 8, czyli 32 bajty. Z drugiej strony jak chcielibyśmy zadeklarować tablicę dwuwymiarową, to pierwszy rozmiar można pominąc (ale będzie to osiem), a drugim rozmiarem musi być minimalna ilość znaków by przechować najdłuższy element, czyli w naszym przypadku 26. 26 · 8 = 208 bajtów.

```
#include <stdio.h>

int main (void)
{
      char dzien[][26] = {
            "Blad: Nie ma takiego dnia", "Poniedzialek", "Wtorek",
            "Sroda", "Czwartek", "Piatek", "Sobota", "Niedziela",
      };

      char *wskDzien[] = {
            "Blad: Nie ma takiego dnia", "Poniedzialek", "Wtorek",
```

```
            "Sroda", "Czwartek", "Piatek", "Sobota", "Niedziela",
      };

      printf("%s\n", dzien[0]);
      printf("%d\n", sizeof (dzien));

      printf("%s\n", wskDzien[0]);
      printf("%d\n", sizeof (wskDzien));

      return 0;
}
```

Listing 5.10.2 Różnica w rozmiarach tablicy dwuwymiarowej i tablicy wskaźników

Tworząc tablicę dwuwymiarową tracimy trochę pamięci, dlatego, że ciągi znaków nie są tego samego rozmiaru (no chyba, że są, ale to szczególny przypadek). Skoro każdy element ma zapewnione 26 znaków, a wykorzystuje mniej to pozostałe miejsce jest nie wykorzystane, a pamięć zarezerwowana. Poniższy rysunek wraz z listingiem mogą pomóc w zrozumieniu tego.

Ciąg znaków	Blad: Nie ma takiego dnia\0			Poniedzialek\0			…	Niedziela\0		
Rozmiar	1	…	26	27	…	52	…	183	…	208

Rysunek 5.10.2 Zajmowana pamięć przez tablicę dwuwymiarową

Poniższy kod pokazuje, ile miejsc zostało nie wykorzystanych, czyli ile pamięci straconej.

```
#include <stdio.h>

int main (void)
{
      char dzien[][26] = {
            "Blad: Nie ma takiego dnia", "Poniedzialek", "Wtorek",
            "Sroda", "Czwartek", "Piatek", "Sobota", "Niedziela",
      };

      int i, j;

      for (i = 0; i < 8; i++)
            for (j = 0; j < 26; j++)
                  printf("dzien[%d][%d]\t= %c\t%p\n", i, j, dzien[i][j],
&dzien[i][j]);

      printf("Rozmiar: %d\n", sizeof(dzien));
      return 0;
}
```

Listing 5.10.3 Zajmowane miejsce przez tablicę dwuwymiarową

Jeśli używamy tablicy wskaźników do odwoływania się do ciągu tekstów zapisanych gdzieś w pamięci, to tekst ten zapisywany jest bajt po bajcie w pewnym obszarze pamięci, a kolejne wskaźniki przypisywane są do adresów, w których zaczyna się „nowy” ciąg znaków, nowy w sensie ten po znaku \0. Poniższy rysunek i listing mogą pomóc w zrozumieniu tego.

Ciąg znaków	Blad: Nie ma takiego dnia\0			Poniedzialek\0			…	Niedziela\0		
Rozmiar	1	…	26	27	…	39	…	76	…	85

Rysunek 5.10.3 Zajmowana pamięć przez ciąg znaków do których odwołują wskaźniki

```
#include <stdio.h>

int main (void)
{
      char *wskDzien[] = {
            "Blad: Nie ma takiego dnia", "Poniedzialek", "Wtorek",
            "Sroda", "Czwartek", "Piatek", "Sobota", "Niedziela",
      };

      int i, j;

      for (i = 0; i < 8; i++)
      {
            for (j = 0; wskDzien[i][j] != '\0'; j++)
                  printf("dzien[%d][%d]\t= %c\t%p\n", i, j, wskDzien[i][j],
&wskDzien[i][j]);
            printf("dzien[%d][%d]\t= %c\t%p\n", i, j, '\0', &wskDzien[i]
[j]);
      }

      return 0;
}
```

Listing 5.10.4 Zajmowane miejsce przez ciągi znaków do których odwołują się wskaźniki

Skoro mowa jest o tablicach wskaźników, a na razie tylko zaprezentowane zostały tablice wskaźników do typu znakowego, to jest sens pokazać jak wyglądają tablice wskaźników np. do typu całkowitego. Może na tym przykładzie zostaną rozwiane wszelkie wątpliwości, jeśli jakieś zaistniały. Tablicę wskaźników do typu int definiuje się analogicznie, tylko zmienia się typ zmiennej. Poniższy listing pokazuje to.

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

```
int main (void)
{
	int x = 10, y = 100, z = 1000;
	int *tab[] = { &x, &y, &z };
	int i;
	int **wskTAB;
	wskTAB = tab;

	for (i = 0; i < 3; i++)
		printf("%p %p %d\n", (tab+i), *(tab+i), **(tab+i));

	for (i = 0; i < 3; i++)
		printf("%p %p %d\n", &tab[i], tab[i], *tab[i]);

	for (i = 0; i < 3; i++)
	{
		printf("%p %p %d\n", wskTAB, *wskTAB, **wskTAB);
		wskTAB++;
	}
}
```

Listing 5.10.5 Tablica wskaźników do typu całkowitego

Na tym przykładzie może widać „lepiej” tablicę wskaźników. Widzimy, że każdy element tablicy tab, jest wskaźnikiem (adresem) do jednej ze zmiennych zadeklarowanych powyżej. Deklaracja **wskTAB mówi o tym, że jest to podwójny wskaźnik, a linijkę niżej przypisujemy do niego adres zerowego elementu tablicy tab. Druga pętla for może wydawać się najbardziej zrozumiała. W pierwszej kolumnie zostanie wydrukowany adres i-tego elementu tablicy, w drugiej kolumnie wartość kryjąca się pod i-tym elementem (czyli adres zmiennej x, y lub z), a w trzeciej kolumnie zostanie wydrukowana wartość kryjąca się pod adresem wyciągniętym z drugiej kolumny. Trzecia pętla różni się od pierwszej tym, że przesuwamy w niej pozycję wskaźnika, czego w pierwszej zrobić nie możemy (omówione w punkcie 5.4).

6 Argumenty funkcji main

Każda funkcja może przyjmować jakieś, bądź nie przyjmować żadnych argumentów. Funkcje przyjmujące argumenty już były, ale funkcja main do tej pory nie przyjmowała żadnych argumentów. Dlatego w tym miejscu opiszę za pomocą jakiego narzędzia można przekazać do funkcji main pewne wartości. Dopiero teraz zostanie to omówione, bowiem narzędziem tym jest tablica wskaźników.

W nawiasach funkcji main wpisujemy dwa parametry. Typem pierwszego jest liczba całkowita, drugi natomiast to tablica wskaźników na typ znakowy. Domyślnie nazwano je argc (*ang. argument count*), czyli ilość podanych argumentów oraz argv (*ang. argument vector*), który jest tablicą wskaźników indeksowanych od zera do argc-1. Oczywiście nazwy parametrów można nazwać według własnego uznania, ale te przyjęto za standardowe. Poniższy rysunek pokazuje ten mechanizm.

Rysunek 6.1 Tablica wskaźników do argumentów funkcji main

A więc tak, uruchamiając program z argumentami (o tym za chwilę) program rezerwuje pamięć na nazwę programu oraz występujące argumenty. W zależności od długości poszczególnych argumentów taka ilość bajtów jest rezerwowana oczywiście plus jeden bajt na znak **\0**. Do tablicy wskaźników przypisywane są adresy początku ciągu znaków. Na pozycji zerowej w tablicy wskaźników argv jest adres pierwszego znaku nazwy programu, na pozycji pierwszej – adres pierwszego znaku pierwszego argumentu, itd. Na rysunku 6.1 widać, że argumenty są dwa, natomiast argc (ilość argumentów) równa się trzy, ponieważ nazwa programu też jest traktowana jako argument. Nasz program wyświetlający tekst przekazany jako argument można zapisać w następujący sposób.

```
#include <stdio.h>
int main (int argc, char *argv[])
{
	int i;
	for (i = 1; i < argc; i++)
		printf("%s ", argv[i]);
	printf("\n");
	return 0;
}
```

Listing 6.1 Argumenty funkcji main

Aby nasz program był taki sam jak schemat przyjęty na rysunku 6.1 musimy go skompilować i podać odpowiednią nazwę, a później uruchomić z dwoma argumentami.

```
$ gcc nazwa_programu.c -o powitanie
```

```
$ ./powitanie Dzien dobry
```

Program należy uruchomić tak jak pokazano powyżej. Do programu przekazaliśmy dwa argumenty i oba zostały wydrukowane. Powiedziano, że nazwa programu też jest argumentem, a nie została wydrukowana – owszem, pętla for zaczyna się od pierwszego elementu, nie zerowego, który wskazuje na nazwę programu.

Kolejny przykład może być trochę ciekawszy. Napiszemy kalkulator, który będzie przyjmował argumenty w następującej kolejności: argument pierwszy, operator, argument drugi, a wynik zostanie wyświetlony. Jako operator mam na myśli **+** (dodawanie), **-** (odejmowanie), **x** (mnożenie), / (dzielenie). Pod listingiem znajduje się opis poszczególnych części programu.

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

int main (int argc, char *argv[])
{
	double tab[2];

	if (argc != 4)
	{
		printf("Uzycie: %s arg1 op arg2\n", argv[0]);
		return -1;
	}
	else
	{
```

```
            tab[0] = atof(argv[1]);
            tab[1] = atof(argv[3]);

            switch(argv[2][0])
            {
                  case '+' : printf("%.2f\n", tab[0] + tab[1]);
                             break;
                  case '-' :      printf("%.2f\n", tab[0] - tab[1]);
                             break;
                  case 'x' :      printf("%.2f\n", tab[0] * tab[1]);
                             break;
                  case '/' : if (!tab[1])
                             {
                                  printf("Blad: Dzielenie przez zero\n");
                                  return -1;
                             }
                             else
                                  printf("%.2f\n", tab[0] / tab[1]);
                             break;
            }
      }
      return 0;
}
```

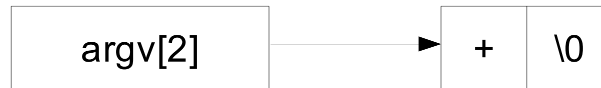
Listing 6.2 Prosty kalkulator

A więc na samym początku tworzymy dwu elementową tablicę liczb typu double, a następnie sprawdzamy, czy argc jest różne od 4 (pamiętamy, nazwa programu to też argument) jeśli tak jest to wydrukuj informację o sposobie używania programu i zakończ jego działanie. Do funkcji printf przekazujemy nazwę programu do wyświetlenia i tak jak pokazane było na rysunku, argv[0] jest wskaźnikiem do nazwy programu, po czym wypisana jest kolejność przyjmowanych argumentów podczas uruchamiania programu. Jeśli użytkownik podczas uruchamiania programu nie wpisze wszystkich argumentów lub wpisze ich za dużo to dostanie informację następującą.

```
Uzycie: ./kalkulator arg1 op arg2
```

Funkcja atof, przyjmuje wskaźnik do tekstu i konwertuje występujące w tym tekście znaki na liczbę typu double. atof akceptuje znak minus na początku ciągu znaków. Jeśli jako parametr atof podamy taki ciąg znaków: **-3aa3** to skonwertowane do typu double zostaną tylko pierwsze dwa znaki, czyli **-3**, pozostałe zostaną odrzucone. Funkcja ta kończy działanie na pierwszym znaku nie będącym liczbą. Wyjątkiem są litery **e**, **E**, które tworzą notację wykładniczą (opisane w punkcie 2.2.4). Jako argumenty funkcji atof podajemy drugi argument wywołania funkcji main (**arg1**) oraz czwarty argument (**arg2**).

Te wartości przypisujemy do elementów tablicy. Teraz może najciekawsza część, jeśli nie wiesz dlaczego argument switch jest zapisany tak argv[2][0], a nie w ten sposób argv[2] to poniższy obrazek może Ci to wyjaśnić.



Rysunek 6.2 Ciąg znaków wskazywanych przez argv[2]

Instrukcja switch jako swój argument chce pojedynczy znak, a nie ciąg znaków. Pomimo tego, iż wpisalismy jako operator tylko znak dodawania, bądź innej operacji to i tak dostawiony został znak **\0**. Aby wpisać tylko jeden znak, musimy odwołać się do zerowego znaku wskazywanego przez ten wskaźnik, czyli dostawić drugi indeks, argv[2][0] odnosi się do znaku plus z rysunku 6.2. Dalsze części instrukcji switch zostały omówione przy omawianiu tejże instrukcji.

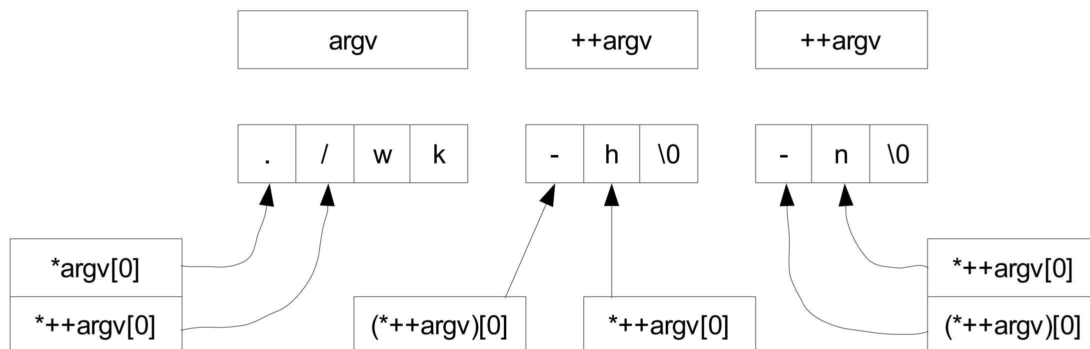
Większość programów w środowisku Linuksa posiada opcję **--help**, która wyświetla pomoc do programu. Ograniczę się tutaj do opcji **-h**, która też będzie wyświetlała pseudo pomoc, a opcja **-h -n**, bądź **-hn** będzie wyświetlała coś innego. Najpierw pokazany zostanie listing, a poniżej opis poszczególnych części.

```
#include <stdio.h>

int main (int argc, char *argv[])
{
       int h = 0, n = 0;
      int inny = 0;
      int znak;

      while (--argc > 0 && (*++argv)[0] == '-')
            while (znak = *++argv[0])
                  switch (znak)
                  {
                        case 'h' : ++h;
                                 break;
                        case 'n' : ++n;
                                 break;
                        default : ++inny;
                                break;
                  }
      if (h == 1 && !inny && !n)
```

```
            printf("Pomoc - sama opcja -h\n");
      else if (h == 1 && n == 1 && !inny)
            printf("Inna pomoc - opcja -hn / -h -n\n");
      else if (!h && !inny && n == 1)
            printf("Inna opcja - opcja -n\n");
      else
            printf("Zla opcja\n");

      return 0;
}
```

Listing 6.3 Pomoc za pomocą opcji

Z pomocą może przyjść poniższy obrazek, na którym zaznaczone są ważne części programu.



Rysunek 6.3 Wskaźniki do nazwy oraz parametru

Zmienne h oraz n zainicjowane są wartością zero, jeśli wystąpiła któraś z tych liter w argumencie, to zostanie zwiększony ich licznik. Zmienna inny będzie zliczała wystąpienia innych argumentów. Pętla while działa dopóki ilość argumentów jest większa od zera. Kolejną część warunku warto rozpatrzeć bardziej szczegółowo. Spójrzmy na rysunek 6.3, który pokazuje, że na nazwę programu wskazuje argv. Operator [] ma najwyższy priorytet, więc jak wykonamy instrukcję *++argv[0] to przesuniemy się wzdłuż ciągu znaków. Jeśli zastosujemy instrukcję (*++argv)[0] to najpierw zostanie zwiększony wskaźnik, czyli zostanie przesunięty do kolejnego ciągu znaków, czyli naszego następnego argumentu, a ponieważ w nawiasie mamy podany indeks zero, to sprawdzamy, czy pierwszym znakiem jest minus, co jest zapisane w warunku. Jeśli tak jest to przypisujemy do zmiennej znak następny znak występujący zaraz po minusie. Następnie wykonuje się instrukcja switch i z powrotem do zmiennej znak przypisywany jest kolejny znak, jeśli ten znak jest zerem to pętla nie jest wykonywana

i wykonywana jest zewnętrzna pętla while, zwiększany jest wskaźnik (przesuwamy go do następnego ciągu znaków) i jeśli na zerowej pozycji występuje minus to sprawdzamy kolejne znaki, jeśli nie to przechodzimy do sprawdzania warunków. Jeśli zmienna h równa się jeden (w argumencie występuje litera **h** tylko jeden raz) oraz litera **n** oraz inne znaki nie występują, to wydrukuj poniższy tekst. Jeśli h równa się jeden i n równa się jeden i nie było innych znaków to wydrukuj poniższy tekst. Jeśli parametrem była tylko litera n, to wydrukuj odpowiednią informację. W przeciwnym wypadku poinformuj o złej opcji. Operator pre – inkrementacji został użyty z tego względu, żeby nie sprawdzać nazwy plików tylko „przejść" od razu na pierwszy argument.

7 Struktury

Struktury w języku C jak i w innych językach programowania służą do przechowywania danych różnych typów, które są ze sobą ściśle powiązane (np. informacje o pracowniku/studencie, współrzędne punktu, długość boków prostokąta, itp) w celu zgrupowania ich pod wspólną nazwą oraz przechowywania w jednym wyznaczonym obszarze pamięci.

7.1 Podstawowe informacje o strukturach

Aby zadeklarować strukturę trzeba użyć słowa kluczowego struct, po którym występuje bądź nie etykieta struktury (nazwa typu strukturowego), następnie pomiędzy nawiasami klamrowymi występują składowe struktury (definicje zmiennych, odwołania do innych struktur, itp.). Po nawiasie klamrowym zamykającym może lecz nie musi występować lista zmiennych (zmienne typu strukturowego). W gruncie rzeczy jeśli zmienne nie występują to zdefiniowaliśmy jedynie typ strukturowy. Przykładowe deklaracje typów strukturowych oraz zmiennych typu strukturowego zostały pokazane na poniższym listingu. Opis poszczególnych mechanizmów został przedstawiony poniżej.

```
#include <stdio.h>
struct punkty {
	int x, y;
};
struct prostokat {
	int a;
	int b;
} pr1 = {3, 12}, pr2;
struct {
	float r;
} kolo;
int main (void)
{
	struct punkty pk1;
	struct prostokat pr3;
	pk1.x = 10;
	pk1.y = 15;
	printf("x: %d\ty: %d\n", pk1.x, pk1.y);
	printf("a: %d\tb: %d\n", pr1.a, pr1.b);
	pr2.a = 8;
	pr2.b = 4;
	printf("a: %d\tb: %d\n", pr2.a, pr2.b);
	pr3.a = 10;
	pr3.b = 12;
```

```
	printf("a: %d\tb: %d\n", pr3.a, pr3.b);
	kolo.r = 3.45;
	printf("r: %.2f\n", kolo.r);
	return 0;
}
```

Listing 7.1.1 Przykłady struktur

Pierwsza struktura, która ma etykietę nazwaną punkty zawiera jak widać dwie składowe - zmienne typu całkowitego. Nie ma zadeklarowanych żadnych zmiennych typu strukturowego, tak więc aby skorzystać z tej struktury trzeba taką zmienną utworzyć. Definicja zmiennej do typu strukturowego punkty jest pierwszą instrukcją w funkcji main, podajemy słowo kluczowe struct nazwę etykiety i nazwę zmiennej. Stworzyliśmy zmienną pk1 typu punkty i z jej pomocą możemy zmieniać wartości zmiennych x i y. Do elementów struktury (składowych struktury) odwołujemy się za pomocą operatora "." (kropka), który występuje pomiędzy zmienną typu strukturowego (pk1) a elementem struktury (x, y). Druga struktura którą zadeklarowaliśmy ma etykietę prostokat, w której również występują dwie składowe – zmienne typu int, lecz ważniejszą rzeczą jest to, iż występują zmienne strukturowe pr1 oraz pr2, za pomocą których można odwoływać się do zmiennych a i b (nie trzeba tworzyć nowej zmiennej, jak to było w poprzednim przypadku), zmienne te są globalne. Jak widać i co jest kolejną właściwością możemy zainicjować wartości początkowe elementów struktury w taki sposób jak zostały zapisane w zmiennej pr1 (dlaczego tak, a nie w standardowy sposób np. int a = 3 zostanie wyjaśnione później). Kolejna struktura, która nie ma etykiety pokazuje następną właściwość, nie trzeba podawać nazwy etykiety wystarczy podać nazwę zmiennej po kończącym składowe nawiasie klamrowym. Lecz w tym przypadku nie zadeklarujemy więcej zmiennych tego typu strukturowego, ponieważ nie ma etykiety z pomocą której taką zmienną moglibyśmy utworzyć.

Teraz pewne informacje odnośnie zajmowanego miejsca przez struktury. Struktura punkty dopóki nie zostanie utworzona zmienna typu punkty nie zajmuje żadnego miejsca w pamięci. Po deklaracji zmiennej pk1 struktura zajmuje sumaryczną ilość bajtów, które zajmują składowe, czyli w naszym przypadku 8 bajtów (dwie zmienne typu całkowitego cztero bajtowego). Rysunek może pomóc w zrozumieniu tego.

Nazwa	pk1		…
	pk1.x	pk1.y	…
Adres	0xbff84718	0xbff8471c	0xbff84720
Rozmiar	4	4	…

Rysunek 7.1.1 Rozmiar struktury

Jak widać adresem początku struktury jest adres pierwszej zmiennej, czyli zmiennej x. Zmienna ta zajmuje cztery bajty, tak więc następny element będzie pod adresem o cztery większym od poprzedniego, co widać na rysunku. Zmienna y kończy się pod adresem 0xbff84720, dlatego też cała struktura kończy się pod tym samym adresem. Struktura zajmuje łącznie 8 bajtów. Może rysunek ten przybliżył trochę sprawę dlaczego nie można przypisywać wartości zmiennym będącym składowymi struktury w dobrze znany nam sposób. Jeśli jest to jeszcze nie jasne to już tłumaczę. Deklaracja struktury do której nie odnosi się żadna zmienna opisuje tylko wzorzec struktury, czyli jak będzie ona reprezentowana później w pamięci. Do typu prostokat zadeklarowaliśmy dwie zmienne (pr1, pr2) i obie te zmienne posiadają dostęp do zmiennych a i b, z tym że zmienna pr1.a jest pod zupełnie innym adresem niż pr2.a. Ogólnie rzecz biorąc struktura bez zmiennej typu strukturowego jest bez użyteczna, staje się użyteczna w momencie gdy stworzymy taką zmienną, co za tym idzie zarezerwujemy pamięć dla wszystkich jej składowych. O nazwach struktur, zmiennych oraz składowych nic nie wspomniano jeszcze, dlatego chcę teraz o tym powiedzieć. Strukturę można nazwać dowolnie, zmienna typu strukturowego może mieć taką samą nazwę jak etykieta struktury oraz jedna z jej składowych. Poniższy listing jest poprawny.

```
#include <stdio.h>
struct info {
      int info;
} info;
int main (void)
{
      info.info = 10;
      printf("%d\n", info.info);
      return 0;
}
```

Listing 7.1.2 Trzy takie same nazwy

7.2 Operacje na elementach struktury

Operacje na elementach struktury są takie same jak na zwykłych zmiennych, jakby nie było są to właściwie zwykłe zmienne, które są zgrupowane w jednym miejscu i odwołujemy się do nich za pomocą specjalnego operatora.

7.3 Przekazywanie struktur do funkcji

O operacjach na elemetach struktury powiedziano, natomiast nie powiedziano o operacjach na całych strukturach, co można robić, a czego nie. Dozwolonymi operacjami dla struktury są:

- Przypisanie do struktury w całości innej struktury
- Pobranie adresu za pomocą operatora &
- Odwołanie się do elementów struktury

Operacją która jest zabroniona to porównywanie struktur. Jeśli chodzi o manipulowanie danymi zawartymi w strukturze za pomocą funkcji to można to zrobić na kilka sposobów. Po pierwsze można przekazać do funkcji wartości i przypisać je do elementów struktury. Po drugie można przekazać całą strukturę i zwrócić całą strukturę za pomocą return (patrz punkt pierwszy dozwolonych operacji). A po trzecie można przekazać wskaźnik do struktury. Wymienione operacja zostały przedstawione na poniższym listingu.

```
#include <stdio.h>
struct informacja {
      int x, y, z;
};
struct informacja przypiszWartosci (int x, int y, int z);
struct informacja dodajWartosci (struct informacja info, int px, int py,
int pz);
void odejmij (struct informacja *info, int px, int py, int pz);
int main (void)
{
      struct informacja info1, info2;
      info1 = przypiszWartosci(10, 100, 1000);
      printf("%d %d %d\n", info1.x, info1.y, info1.z);
      info2 = dodajWartosci(info1, 40, 50, 60);
      printf("%d %d %d\n", info2.x, info2.y, info2.z);
```

```
      odejmij(&info2, 100, 300, 500);
      printf("%d %d %d\n", info2.x, info2.y, info2.z);
      printf("%p %p %p %p\n", &info1, &info1.x, &info1.y, &info1.z);
      printf("%p %p %p %p\n", &info2, &info2.x, &info2.y, &info2.z);
      return 0;
}
struct informacja przypiszWartosci (int x, int y, int z)
{
      struct informacja tmp;
      tmp.x = x;
      tmp.y = y;
      tmp.z = z;
      return tmp;
}
struct informacja dodajWartosci (struct informacja info, int px, int py,
int pz)
{
      info.x += px;
      info.y += py;
      info.z += pz;
      return info;
}
void odejmij (struct informacja *info, int px, int py, int pz)
{
      info->x -= px;
      info->y -= py;
      info->z -= pz;
}
```

Listing 7.3.1 Pewne operacje z użyciem struktur

Pamiętamy, że funkcje przed nazwą miały typ zwracanej wartości, dlatego też w prototypach tych dwóch funkcji jako typ zwracanej wartości jest struct informacja, co oznacza, że funkcja zwraca strukturę typu informacja, czyli taką jak zdefiniowana jest powyżej. Funkcja przypiszWartosci pobiera trzy argumenty całkowite. W ciele tej funkcji musimy zadeklarować zmienną typu informacja (tmp), w której przechowamy tymczasowo przypisane dane do momentu zwrócenia całej struktury i przypisania jej do konkretnej zmiennej (info1). Skoro funkcja zwraca strukturę typu informacja, to możemy wywołać tę funkcję przypisując ją do zmiennej, która odnosi się do struktury informacja i tym sposobem przypisaliśmy jedną strukturę do drugiej. Funkcja dodajWartosci też zwraca strukturę typu informacja, ale jako pierwszy argument przyjmuje całą strukturę, kolejnymi argumentami są wartości, które zostają dodane. Widać w ciele tej funkcji, że nie musimy deklarować pomocniczej zmiennej do tej struktury, skoro przekazaliśmy strukturę, to wystarczy odwołać się do jej pól. Zwracamy znowu całą strukturę i tym razem przypisujemy ją do info2. Funkcja odejmij przyjmuje

wskaźnik do struktury, jak się łatwo domyśleć nie trzeba zwracać wartości, by zmiany były widoczne. Tutaj chyba najciekawsza część, bowiem został użyty dotąd nie używany operator -> (minus nawias trójkątny zamykający), który ma najwyższy priorytet. Co ciekawe postać (*info).x -= px jest równoważna, nawiasy są konieczne ponieważ operator "." (kropka) ma wyższy priorytet niż operator * (operator wyłuskania / dereferencji), gdybyśmy opuścili nawiasy to dostalibyśmy taki komunikat.

```
error: request for member 'x' in something not a structure or union
```

Dwie dotąd nie omówione linie listingu 7.1.2 to te drukujące adresy. Można sprawdzić jaki adres ma struktura i jej poszczególne elementy. Adres struktury i pierwszego elementu będą takie same, co zostało pokazane na rysunku 7.1.1.

Pomimo iż pokazane zostały operacje związane ze wskaźnikami do struktur to należy jednak o tym powiedzieć dokładniej, dla przykładu pokazany został poniższy kod.

```
#include <stdio.h>
struct info {
	char imie[30];
	int wiek;
	float waga;
} dane = {"Kasia", 19, 53.3};
void x (void);
int main (void)
{
	struct info dk, *wsk, *wsk_dk;
	dk = dane;
	wsk = &dane;
	wsk_dk = &dk;
	printf("%s\t%d\t%.2fkg\n", wsk_dk->imie, wsk_dk->wiek, wsk_dk-
>waga);
	printf("%s\t%d\t%.2fkg\n", (*wsk_dk).imie, (*wsk_dk).wiek,
(*wsk_dk).waga);
	printf("%s\t%d\t%.2fkg\n", wsk->imie, wsk->wiek, wsk->waga);
	printf("%s\t%d\t%.2fkg\n", (*wsk).imie, (*wsk).wiek, (*wsk).waga);
	x();
	printf("%s\t%d\t%.2fkg\n", wsk_dk->imie, wsk_dk->wiek, wsk_dk-
>waga);
	printf("%s\t%d\t%.2fkg\n", (*wsk_dk).imie, (*wsk_dk).wiek,
(*wsk_dk).waga);
	printf("%s\t%d\t%.2fkg\n", wsk->imie, wsk->wiek, wsk->waga);
	printf("%s\t%d\t%.2fkg\n", (*wsk).imie, (*wsk).wiek, (*wsk).waga);
	return 0;
}
void x (void)
{
```

```
	dane.wiek = 0;
	dane.waga = 0.0;
}
```

Listing 7.3.2 Wskaźnik do struktury

A więc stworzyliśmy strukturę info, do której odwołuje się zmienna dane z zainicjowanymi wartościami (tak przy okazji tablicę znaków inicjuje się właśnie w ten sposób). W funkcji main definiujemy zmienną dk oraz dwa wskaźniki do typu strukturowego. Do tych wskaźników trzeba przypisać adres zmiennych. Do wskaźnika wsk przypisujemy adres zmiennej dane (zmienna ta jak się później okaże jest zmienną globalną), do wskaźnika wsk_dk adres zmiennej dk, do której dwie linie wcześniej skopiowaliśmy zawartość zmiennej dane. Jak widać operacje wsk_dk->imie oraz (*wsk_dk).imie są równoważne. Analogicznie jest ze wskaźnikiem wsk. Wywołanie funkcji x zeruje dwie zmienne liczbowe ze struktury (zmienna dane jest globalna co widać po wykonaniu dwóch ostatnich instrukcji printf). Dwie instrukcje printf zaraz po wywołaniu funkcji x wyświetlą dalej te same wartości, ponieważ wskaźnik wskazuje na zmienną lokalną, do której tylko skopiowaliśmy zawartość zmiennej dane.

Poniższy przykład może i jest trochę fikcyjny, ale pokazuje pewne dotąd nie omówione właściwości wskaźników.

```
#include <stdio.h>
struct dane {
	int wiek;
	int *wsk_wiek;
};
struct info {
	struct dane *z_dan;
};
int main (void)
{
	struct info inf1, *wsk_inf;
	wsk_inf = &inf1;
	struct dane dan1;
	dan1.wsk_wiek = &dan1.wiek;
	wsk_inf->z_dan = &dan1;
	wsk_inf->z_dan->wiek = 80;
	printf("%d\n", wsk_inf->z_dan->wiek);
	printf("%d\n", *wsk_inf->z_dan->wsk_wiek);
	return 0;
}
```

Listing 7.3.3 Kolejne użycie wskaźników

Tworzymy dwa typy strukturowe, typ dane oraz typ info. W pierwszym z nich składowymi są liczba typu int i wskaźnik na taką liczbę, w drugim natomiast wskaźnik na typ dane. W funkcji main deklarujemy zmienną typu info oraz wskaźnik tego samego typu. Do wskaźnika przypisujemy adres tej zmiennej, żeby móc się odwoływać do elementów struktury za pomocą wskaźnika. Zmienna dan1 typu dane też musi zostać utworzona, bo będzie potrzebny jej adres. Do wskaźnika wsk_wiek (struktura dane) przypisujemy adres składowej struktury dane - wiek. Teraz za pomocą wskaźnika na strukturę info (wsk_inf) przypisujemy adres zmiennej strukturowej dan1 do wskaźnika na taką strukturę (z_dan). A następnie możemy zmienić wartość zmiennej wiek za pomocą wskaźników. Druga instrukcja printf wydrukuje wartość zmiennej wskaźnikowej wsk_wiek, ponieważ zapis *wsk_inf->z_dan->wsk_wiek jest równoważny zapisowi *(wsk_inf->z_dan->wsk_wiek) dlatego, że operator -> ma większy prioryter niż operator *. Czyli najpierw dostaniemy się do adresu, a później wyciągniemy to co pod nim siedzi.

7.4 Zagnieżdżone struktury

Struktury można zagnieżdżać to znaczy działa to w ten sposób, że definiujemy jedną strukturę, nie musimy tworzyć do niej zmiennej, a następnie definiujemy drugą strukturę, w której jako składową deklarujemy zmienną do tej pierwszej. Mam nadzieję, że na poniższym przykładzie zostanie to wyjaśnione. Tabela przedstawia dane, które zostaną zapisane w strukturze.

Imię	Nazwisko	Wzrost	Waga	Data urodzenia		
				Dzień	Miesiąc	Rok
Jan	Kowalski	185	89	12	9	1979

Tabela 7.4.1 Dane zawodnika

```
#include <stdio.h>
#include <string.h>
struct dataUrodzenia {
	int dzien;
	int miesiac;
	int rok;
};
struct daneZawodnika {
	char imie[20];
	char nazwisko[30];
```

```
	int wzrost, waga;
	struct dataUrodzenia urodziny;
};
int main (void)
{
	struct daneZawodnika dane;
	strncpy(dane.imie, "Jan", sizeof (dane.imie));
	strncpy(dane.nazwisko, "Kowalski", sizeof (dane.nazwisko));
	dane.wzrost = 185;
	dane.waga = 89;
	dane.urodziny.dzien = 12;
	dane.urodziny.miesiac = 9;
	dane.urodziny.rok = 1979;
	printf("%s\t%s\t%d\t%d\t%d\t%d\t%d\n", dane.imie, dane.nazwisko,
dane.wzrost, dane.waga, dane.urodziny.dzien, dane.urodziny.miesiac,
dane.urodziny.rok);
	return 0;
}
```

Listing 7.4.1 Struktury zagnieżdżone

Najpierw deklarujemy typ strukturowy dataUrodzenia, który będzie służył do przechowywania dnia, miesiąca oraz roku urodzenia zawodnika. Widać z tabeli, że nadaje się to do rozdzielenia, żeby w danych zawodnika nie było zmiennej opisującej dzień, miesiąc czy rok, tylko odpowiednia struktura. W strukturze daneZawodnika są ogólne informacje o zawodniku i jest tworzona zmienna urodziny, która odnosi się do pierwszej struktury. W funkcji main tworzymy zmienną dane, która odwołuje się do typu daneZawodnika, a następnie kopiujemy ciąg znaków „Jan" za pomocą funkcji strncpy (biblioteki standardowej – string.h) do tablicy znaków imie podając jako trzeci argument maksymalną ilość kopiowanych znaków. Analogicznie dzieje się z nazwiskiem. Kolejne przypisania znane są już z poprzedniego punktu. Ciekawe może być odwołanie się do pól struktury dataUrodzenia. A mianowicie robi się to tak, najpierw podaje się nazwę struktury, stawia się kropkę i podaje nazwę pola (analogicznie jak w poprzednim punkcie), tylko teraz naszym polem jest kolejna struktura, tak więc po kropce stawiamy nazwę zmiennej, która odwołuje się do tamtej struktury, stawiamy kolejną kropkę i podajemy wreszcie to pole, do którego chcemy się odwołać. Analogicznie byłoby, gdybyśmy stworzyli więcej zagnieżdżeń.

7.5 Tablice struktur

Tablice struktur deklaruje się analogicznie do zwykłych tablic. Właściwie to deklaruje się je tak samo jak zwykłą zmienną typu strukturowego, tylko dodaje się rozmiar. Czyli deklaracja tablicy do struktury daneZawodnika będzie wyglądała następująco.

```
struct daneZawodnika tab[10];
```

```
#include <math.h>
#include <stdio.h>
#define MAX 10
struct dane {
	int k;
	float x;
};
int main (void)
{
	struct dane tab[MAX];
	int i;
	for (i = 0; i < MAX; i++)
	{
		tab[i].k = pow(2, i);
		tab[i].x = i + 3.5 * (i + 2);
	}
	for (i = 0; i < MAX; i++)
	{
		printf("tab[%d].k = %d\t", i, tab[i].k);
		printf("tab[%d].x = %.2f\n", i, tab[i].x);
	}
	return 0;
}
```

Listing 7.5.1 Tablica struktur

Deklaracja tablicy jest pierwszą instrukcją funkcji main, której rozmiar zadeklarowany jest jako stała symboliczna MAX. W pętli for uzupełniamy elementy tab[i].k oraz tab[i].x, a następnie drukujemy wszystkie wartości. Program miał na celu jedynie pokazać w jaki sposób odwołujemy się do poszczególnych elementów tablicy struktur.

Kolejnym przykładem będzie tablica struktur wraz z inicjalizacją. Program będzie drukował wszystkie wiersze struktury, chyba, że dostanie podczas uruchomienia jako argument numer wiersza, jeśli będzie on poprawny, to wydrukuje tylko ten wiersz. Opis znajduje się poniżej listingu.

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

```
struct ludzie {
      char *imie;
      char *nazwisko;
      int wiek;
} id[] = {
      {"Jan", "Kowalski", 18},
      {"Tadeusz", "Nowak", 55},
      {"Marcin", "Maly", 23},
      {"Karol", "Biegacz", 45},
      {"Tomasz", "Smialy", 20},
      {"Kamil", "Mlody", 22},
      {"Tymon", "Kowalewski", 28},
};

int main (int argc, char *argv[])
{
      int i, k, kontrola = 0;
      int ilElem = sizeof (id) / sizeof (struct ludzie);

      if (argc == 2)
      {
            k = atoi(argv[1]);
            if (k > 0 && k <= ilElem)
                  kontrola = 1;
      }

      if (kontrola)
      printf("%s %s %d\n", id[k-1].imie, id[k-1].nazwisko, id[k-1].wiek);
      else
            for (i = 0; i < ilElem; i++)
            printf("%s %s %d\n", id[i].imie, id[i].nazwisko, id[i].wiek);
      return 0;
}
```

Listing 7.5.2 Tablice struktur i argumenty funkcji

Jak widać na listingu tablicę struktur inicjuje się w taki właśnie sposób. Dla lepszej czytelności każdy „wiersz” struktury został ujęty w nawiasy klamrowe (które mogły by zostać pominięte) oraz został zapisany w osobnej linii, dlatego też dużo łatwiej jest powiedzieć, jakie nazwisko kryje się pod id[3].nazwisko. W funkcji main deklarujemy zmienną kontrolną kontrola z zainicjowaną wartością zero, zmienną k służy do przetrzymywania argumentu wywołania programu. Zmienna ilElem zawiera ilość elementów, która jest obliczona za pomocą znanej już metody (przedstawionej w punkcie 5.4) dla przypomnienia **rozmiar tablicy / rozmiar typu = ilość elementów**. Jeśli program został wywołany z parametrem to przekształcamy go do typu int i przypisujemy do zmiennej k. Poniższy warunek sprawdza czy liczba ta jest większa od zera i mniejsza lub równa od ilości elementów, jeśli tak jest to

zmienna kontrola przyjmuje wartość jeden. Po ustaleniu, czy program został wywołany z argumentem i czy argument był z odpowiedniego zakresu sprawdzamy stan zmiennej kontrolnej. Jeśli jest różna od zera to drukujemy tylko wiersz, który podaliśmy jako argument. Zapis id[k-1], a konkretnie k-1 jest niezbędne, ponieważ jak wiadomo tablice są indeksowane od zera, a dzięki temu, że warunek na przedział liczby był taki jaki był musimy odjąć jeden, żeby indeks się zgadzał. Zazwyczaj dla ludzi pierwszy występujący wiersz nazywany jest pierwszym nie zerowym, tak więc można wpisać jako argument wywołania programu 1, aby uzyskać dostęp do zerowego wiersza. Ma to też inną zaletę, jeśli wpisalibyśmy jako argument np. literę, to funkcja atoi przypisała by do zmiennej k wartość zero i tak wpisując literę dostalibyśmy zerowy wiersz co raczej nie byłoby zamierzonym efektem.

7.6 Słowo kluczowe typedef

Za pomocą słowa kluczowego typedef tworzy się nowe nazwy dla dobrze już znanych nam typów danych. Co to znaczy i do czego to może się przydać? Na poniższym listingu zastosowano ten mechanizm, opis znajduje się poniżej.

```
#include <stdio.h>
typedef struct pole_ab {
      int a, b;
} Bok;
int f_pole (Bok pole);
int main (void)
{
      typedef int Licznik;
      typedef int Dlugosc;
      Bok pole;               // struct pole_ab pole;
      Licznik i;              // int i;
      Dlugosc max = 15;       // int max = 15;
      for (i = 0; i < max; i++)
      {
            pole.a = i + 3;
            pole.b = pole.a + 7;
            printf("Pole (%d, %d) = %d\n", pole.a, pole.b, f_pole(pole));
      }
      return 0;
}
int f_pole (Bok pole)
{
      return pole.a * pole.b;
}
```

Listing 7.6.1 Użycie typedef

Zacznijmy więc od miejsca, w którym definiowana jest struktura. Za pomocą słowa typedef stworzyliśmy nazwę Bok, która może być pisana w zastępstwie struct pole_ab. Znaczy to tyle, że Bok pole; oznacza dokładnie to samo co struct pole_ab pole; jest to definicja nowej nazwy odnoszącej się do „starego” typu danych. Analogicznie jest z dwiema zmiennymi typu całkowitego, pomimo iż nie jest to skrócony zapis, to może okazać się pomocny. Jeśli zadeklarowalibyśmy kilka zmiennych za pomocą Licznik, to wiemy (bynajmniej takie jest założenie), że zmienne te odnoszą się do liczenia czegoś, np. ilości wykonań pętli. Jak powiedziane było w jednym z poprzednich punktów, aby przekazać strukturę do funkcji musimy zrobić to tak:

```
int nasza_funkcja (struct nazwa_struktury nazwa_zmiennej)
```

I my zrobiliśmy dokładnie to samo, tylko za pomocą zdefiniowanej prędzej nazwy Bok. Używanie typedef ma swoje korzyści, ponieważ zazwyczaj skraca zapis, np. podczas przekazywania argumentów do funkcji. Należy zwrócić uwagę na jedną ważną rzecz, składnia użycia typedef ma się następująco:

```
typedef nazwa_typu_danych nowa_nazwa
```

Gdyby nie było słowa typedef to nowa_nazwa byłaby nazwą zmiennej podanego typu, tak samo jest w przypadku poniższym

```
typedef struct pole_ab {
      int a, b;
} Bok;
```

Gdyby nie typedef to Bok byłby zmienną odnoszącą się do typu strukturowego pole_ab. Dla wyróżnienia nazwy odnoszącej się do jakiegoś typu zapisujemy ją rozpoczynając wielką literą, aczkolwiek nie jest to żaden wymóg.

7.7 Unie

Unie bynajmniej ze sposobu zapisu podobne są do struktur, nie mniej jednak są trochę inne. Co to znaczy? Przypomnijmy sobie ile miejsca zajmowała struktura w pamięci komputera. Struktura zajmuje sumaryczną ilość bajtów zajmowanych przez jej elementy. Z unią jest trochę inaczej, ponieważ unia zajmuje tyle miejsca ile zajmuje największe z jej pól. W jednym czasie może przechowywać wartość jednego typu. Poniższy rysunek może pomóc w zrozumieniu tego, a poniższy listing pokazuje działanie unii.

int					x	x	x	x	x	x	x	x	x	x	x	x
double									x	x	x	x	x	x	x	x
int tab[4]																
Rozmiar unii	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Adres	0xbfebf100			…										0xbfebf110		

Rysunek 7.7.1 Schemat zajmowanej pamięci przez unię

Na rysunku widać, że typ int zajmuje 4 bajty, typ double 8 bajtów, a cztero elementowa tablica liczb typu int zajmuje 16 bajtów. Cała unia zajmuje tyle, żeby pomieścić największy z typów, czyli w tym przypadku 16 bajtów. Unia w jednej chwili może posiadać tylko jedną wartość konkretnego typu, co zostało pokazane na poniższym listingu.

```
#include <stdio.h>

union dane {
	int i;
	double x;
	int tab[4];
};

int main (void)
{
	int i;
	union dane u_data = {10};
	printf("Addr: %p Rozm: %d\n", &u_data, sizeof (u_data));
	printf("Addr: %p Rozm: %d\n", &u_data.i, sizeof (u_data.i));
	printf("Addr: %p Rozm: %d\n", &u_data.x, sizeof (u_data.x));
	printf("Addr: %p Rozm: %d\n", &u_data.tab, sizeof (u_data.tab));

	printf("u_data.i = %d\n", u_data.i);
	u_data.x = 19.9;
	printf("u_data.x = %.2f\n", u_data.x);
	printf("u_data.i = %d\n", u_data.i);

	u_data.tab[0] = 4;
	u_data.tab[1] = 9;
	u_data.tab[2] = 13;
	u_data.tab[3] = 159;

	for (i = 0; i < sizeof (u_data.tab)/sizeof (u_data.tab[0]); i++)
		printf("u_data.tab[%d] = %d\n", i, u_data.tab[i]);

	printf("u_data.x = %.2f\n", u_data.x);
	u_data.x = 19.9;
```

```
	printf("u_data.x = %.2f\n", u_data.x);

	printf("u_data.i = %d\n", u_data.i);
	printf("u_data.i = %d\n", *(&u_data.i+1));
	printf("u_data.i = %d\n", *(&u_data.i+2));
	printf("u_data.i = %d\n", *(&u_data.i+3));
	return 0;
}
```

Listing 7.7.1 Użycie unii

Na listingu widać, że unię definiuje się i używa tak samo jak strukturę. W funkcji main tworzymy zmienną odwołującą się do unii, możemy zainicjować wartość, lecz tylko dla jednego (pierwszego) elementu unii (czyli tutaj dla zmiennej i). Pierwsza instrukcja printf drukuje adres początku unii oraz jej rozmiar. Kolejne trzy instrukcje printf drukują informacje o adresie, w którym zaczynają się zmienne (czyli początek unii) oraz rozmiar jaki dany typ zajmuje. Ponieważ do zmiennej i przypisaliśmy wartość podczas inicjalizacji to możemy tę wartość wydrukować. W następnej linii przypisujemy do zmiennej typu double pewną wartość. Wartość, która była pod zmienną i zostaje „zasłonięta” przez co śmieci zostają wydrukowane, gdy chcemy odwołać się do tej zmiennej w kolejnej linii. Kolejne instrukcje przypisują wartości do tablicy. W pętli drukujemy te wartości, w warunku występuje konstrukcja omówiona w rozdziale 5.4. Próba wydrukowania zmiennej x jest nie udana, ponieważ wartość ta została już zasłonięta. Po przypisaniu wartości do zmiennej x przysłaniamy dwa pierwsze elementy tablicy i co jest ważne, nie zasłoniliśmy pozostałych, czyli możemy się do nich odwołać. Żeby tego dokonać trzeba użyć wskaźników i przesunąć wskaźnik na odpowiednią pozycję. Wskaźnik ten został przesunięty w instrukcji printf podany jako drugi argument. Najpierw staramy się wydrukować wartość, która znajduje się pod zmienną i – dostajemy śmieci co jest zrozumiałe. W następnej linii za pomocą instrukcji &u_data.i wyciągamy adres początku unii (oraz początku zmiennej i) i dodajemy 1 czyli przesuwamy wskaźnik o **1 · rozmiar typu int**, czyli 4 bajty i za pomocą gwiazdki wyciągamy wartość, która jest pod tym adresem. Rysunkiem można się wesprzeć i dzięki niemu wiemy, że rozmiar double zawiera 8 bajtów czyli pod tym adresem, na który przesuneliśmy wskaźnik też będą znajdować się śmieci, ponieważ przypisując wartość do zmiennej x zasłoniliśmy pierwsze 8 bajtów. Przesunięcie wskaźnika o dwie i trzy pozycje, co zostało zrobione w kolejnych instrukcjach wyświetla poprawne dane, czyli te które były zapisane w tablicy, ponieważ dane te nie zostały zasłonięte. Pewna ważna kwestia odnośnie przesuwania wskaźnika, w tych instrukcjach wydrukowaliśmy tylko to co jest pod kolejnymi adresami, samego wskaźnika nie ruszyliśmy –

wskaźnik dalej wskazuje na początek unii.

7.8 Pola bitowe

Pomimo iż obecne komputery raczej nie mają problemu z ilością pamięci, to jednak można ją zaoszczędzić za pomocą pól bitowych. Znaczy to tyle, że jeśli pewne informacje przetrzymują wartości 1 lub 0 (czyli np. włączony/wyłaczony przełącznik, stan diody, itp.) to po co rezerwować dla każdej takiej zmiennej dużo więcej pamięci niż jest potrzebne? Dla przykładu weźmy sytuację (rozwiązanie tego zadania składało by się ze znacznie większej ilości operacji, niż te przedstawione tutaj), w której mamy namiastkę komputera pokładowego w samochodzie. Czujniki przesyłają informacje na temat zasilania samochodu, poziomu paliwa oraz wody w zbiornikach, stanu całej elektryczności i stanu hamulców. Na listingu 7.8.1 znajduje się program wykorzystujący pola bitowe, a pod nim pomocny rysunek wraz z opisem.

```
#include <stdio.h>
struct {
	unsigned int zasilanie	: 1;
	unsigned int paliwo	: 3;
	unsigned int woda	: 2;
	unsigned int elektr	: 1;
	unsigned int hamulce	: 1;
} samochod;

void init (void);
int main (void)
{
	printf("Rozmiar: %d\n", sizeof (samochod));
	init();
	if (samochod.zasilanie == 1)
		printf("Zasilanie poprawne\n");
	else
		printf("Zasilanie nie dziala poprawnie\n");
	printf("Ilosc paliwa: %.2f%%\n", samochod.paliwo / 4.0 * 100);
	printf("Ilosc wody:   %.2f%%\n", samochod.woda / 2.0 * 100);
	if (samochod.elektr == 1)
		printf("Elektryka dziala poprawnie\n");
	else
		printf("Elektryka nie dziala poprawnie\n");
	if (samochod.hamulce == 1)
		printf("Hamulce dzialaja poprawnie\n");
	else
		printf("Hamulce nie dzialaja poprawnie\n");
```

```
	return 0;
}
void init (void)
{
	samochod.zasilanie = 1;
	samochod.paliwo = 3;
	samochod.woda = 1;
	samochod.elektr = 1;
	samochod.hamulce = 1;
}
```

Listing 7.8.1 Pola bitowe

Nazwa	-	-	-	-	h	e	w		p			z
Nr bitu	31	…			7	6	5	4	3	2	1	0

h	-	samochod.hamulce
e	-	samochod.elektr
w	-	samochod.woda
p	-	samochod.paliwo
z	-	samochod.zasilanie

Rysunek 7.8.1 Pola bitowe

Teraz jeszcze pewną rzecz odnośnie liczb binarnych muszę napomknąć. Na jednym bicie mogą zostać zapisane wartości 0 lub 1. Możliwe wartości zapisane na dwóch i trzech bitach znajdują się w tabeli 7.8.1.

3 bity			W
	2 bity		
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Tabela 7.8.1 Możliwe wartości zapisane na dwóch i trzech bitach

Zmienna typu strukturowego samochod ma rozmiar czterech bajtów, a możemy zapisać 5 różnych informacji do naszego zadania, co w przypadku zwykłych zmiennych wiązało by się z liczbą **rozmiar typu · ilość różnych informacji** (czyli 20 bajtów). Zeby zadeklarować zmienną odnoszącą się do konkretnej ilości bitów należy w strukturze po nazwie zmiennej postawić dwukropek oraz podać liczbę bitów, co pokazano na listingu. Informacja na temat zasilania (zasilanie) wymaga jednego bitu, jeśli jest zasilanie to ustaw wartość na 1, jeśli nie ma to 0, tak samo jest ze zmiennymi przyjmującymi informacje o elektryczności (elektr) i o stanie hamulców (hamulce). Informacje o zawartości płynów w zbiorniku wymaga większej ilości bitów. Informacje o stanie paliwa mają być wyświetlane w pięcio stopniowej skali (tj. 0, 25, 50, 75, 100 [%]). Spójrz teraz na tabelę 7.8.1 i zwróć uwagę na to, że na dwóch bitach możemy zapisać cztery wartości (0, 1, 2, 3), czyli potrzebujemy trzech bitów, aby nasza skala pokazywała dobry poziom. Dla poziomu wody wystarczą dwa bity, ponieważ nasza trzy stopniowa skala pokazuje 0, 50, 100%. Odwołania do pól bitowych są analogiczne jak do zwykłych zmiennych będącymi składowymi struktury. W funkcji init przypisujemy wartości tym właśnie zmiennym. Należy pamiętać o bardzo ważnej rzeczy, nie możemy przypisać do zmiennej, która odwołuje się do dwóch bitów wartości np. cztery. Jeśli spróbujemy zrobić coś takiego jak na poniższym listingu to otrzymamy ostrzeżenie od kompilatora, program się skompiluje, ale wynik nie będzie tym, którego oczekiwaliśmy.

```
warning: large integer implicitly truncated to unsigned type
```

```
#include <stdio.h>

struct {
	unsigned int bl : 2;
} x;

int main (void)
{
	x.bl = 4;
	printf("%d\n", x.bl);
}
```

Listing 7.8.2 Przypisanie większej liczby, niż dozwolona

Kolejną rzeczą o której trzeba pamiętać jest to, że zmienna która odwołuje się do pola bitowego musi być liczbą całkowitą ze znakiem (signed) lub bez znaku (unsigned). Pola te traktowane są jako malutkie zmienne typu całkowitego, tak więc wszystkie operacje arytmetyczne mogą zostać wykonane, co zresztą widać w instrukcjach printf, które wyświetlają poziom zawartości płynów w zbiornikach. Jeśli chcemy zapełnić dziury (nie wykorzystane bity pomiędzy polami) możemy to zrobić, nie wpisując nazwy zmiennej, czyli struktura wyglądała by tak:

```
struct {
	unsigned int zasilanie	: 1;
	unsigned int paliwo	: 3;
	unsigned int woda	: 2;
	unsigned int elektr	: 1;
	unsigned int hamulce	: 1;
	unsigned int		: 24;
} samochod;
```

Za pomocą tej operacji wypełniliśmy dziurę, która była pomiędzy wykorzystanymi bitami, a resztą do długości słowa, czyli do długości rozmiaru typu int. Dopóki rozmiar tej struktury będzie się równał cztery bajty, to tak jakbyśmy mieli jedną zwykłą zmienną zadeklarowaną jako składową struktury. Jeśli rozmiar będzie większy np. osiem bajtów to analogicznie dwie zmienne typu int. Za pomocą specjalnego rozmiaru 0 możemy przesunąć kolejne pola bitowe do granicy następnego słowa, poniższy przykład pokazuje to.

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

```
struct {
      unsigned int a : 1;
      unsigned int b : 2;
      unsigned int c : 1;
      unsigned int   : 0;
      unsigned int d : 10;
      unsigned int   : 22;
} x;

int main (void)
{
      printf("%d\n", sizeof (x));
      return 0;
}
```

Listing 7.8.3 Uzycie rozmiaru 0

Użycie rozmiaru 0 powoduje, że pole bitowe d będzie zaczynało się od nowego słowa (czyli tak jakby druga zmienna zwykła), dlatego też rozmiarem struktury będzie osiem bajtów.

Jeśli piszemy programy na maszynie, w której bity pola umieszcza się od prawej do lewej, jak to pokazano na rysunku 7.8.1 to program nie będzie działał poprawnie na maszynach w której umieszcza się bity w odwrotnej kolejności (od lewej do prawej).

8 Operacje wejścia i wyjścia

Operacje wejścia i wyjścia były już używane w poprzednich rozdziałach. Operacje wejścia może mniej, natomiast funkcję printf znamy już dość dobrze. W kolejnych podpunktach zawarte są informacje na temat funkcji wyświetlających dane (bardziej szczegółowo) oraz na temat tych, które pobierają dane od użytkownika, wraz z obsługą błędów. Znajdują się też informacje o obsłudze plików oraz zmiennej ilości argumentów.

8.1 Funkcja getchar i putchar

Funkcja getchar była już używana, chociażby w punkcie 3.3.2 podczas omawiania pętli while. W tamtym miejscu zapis warunku był ok, jeśli chodzi o wczytywanie tylko jednego wiersza. W tym miejscu pokażę jak za pomocą właściwie tego samego kodu wydrukować całą zawartość pliku (o drukowaniu zawartości pliku też jeszcze nie było). Funkcja ta została opisana w punkcie 3.3.2, stała EOF została opisana w punkcie 4.4. Tak więc połączmy te dwie rzeczy i napiszmy program, który drukuje pobrany tekst. Funkcja przestaje działać jak osiągnie koniec pliku (tudzież wczytywanie danych z klawiatury zostanie przerwane).

```
#include <stdio.h>
int main (void)
{
	int c;
	while ((c = getchar()) != EOF)
			putchar(c);
	return 0;
}
```

Listing 8.1.1 Drukowanie wczytanego tekstu

Po skompilowaniu i uruchomieniu programu wpisujemy teskt, po wciśnięciu klawisza **enter** przechodzimy do nowej linii, lecz tekst, który został pobrany zostaje wydrukowany. Aby zakończyć wciskamy kombinację **ctrl-c** lub **ctrl-d**.

Aby wyświetlić zawartość całego pliku za pomocą naszego programu trzeba ten plik przekazać do programu i wtedy funkcja getchar zamiast oczekiwać na znaki wpisane z klawiatury pobiera je z pliku. Żeby przekazać plik tekstowy do naszego programu musimy wpisać następujące polecenie.

```
$ ./main < main.c
```

Zakładając, że plik wykonywalny nazywa się main, a plik z kodem źródłowym main.c. Jeśli wszystko poszło dobrze, to kod źródłowy zostanie wydrukowany na ekranie.

Można to też zrobić za pomocą potoku. Potok jest to mechanizm komunikacji międzyprocesowej, lecz w to wgłębiać się nie będziemy. Użycie tego jest następujące

```
$ cat main.c | ./main
```

Chodzi o to, że to co otrzymujemy na wyjściu pierwszego polecenia (przed znakiem „|”) przekazywane jest na wejście drugiego polecenia. W tym przypadku jest to działanie na około, ponieważ samo polecenie cat wyświetla zawartość pliku, lecz chodziło o pokazanie alternatywy, swoją drogą używanie potoków jest bardzo użyteczne.

Skoro można przekazać plik tekstowy na wejście, to może można zapisać do pliku (przekazać na wyjście) coś co wpisujemy, bądź zawartość innego pliku. Za pomocą poniższego polecenia wszystko co wpiszemy zostanie przekazane do pliku plik.txt i nic nie zostanie wyświetlone na ekranie.

```
$ ./main > plik.c
```

A za pomocą poniższego polecenia cały plik main.c zostanie przekazany do pliku plik.c

```
$ ./main < main.c > plik.c
```

Funkcja putchar podobnie do funkcji printf wyświetla informacje na standardowym wyjściu (ekran monitora) z tą różnicą, że putchar nie może formatować wyjścia.

8.2 Funkcja printf i sprintf

Z funkcją printf mieliśmy styczność właściwie w każdym programie, podstawowe deskryptory formatu dla liczb całkowitych i rzeczywistych są już znane, lecz wszystkie wraz z różnymi kombinacjami zostały przedstawione w tym podpunkcie. Ogólna deklaracja funkcji printf ma się następująco:

```
int printf (char *format, arg1, arg2, ...)
```

Wielokropek oznacza, że funkcja nie ma sztywnej ilości parametrów – ma zmienną (więcej informacji o zmiennej ilości parametrów znajduje się w podpunkcie 8.4). Funkcja printf zwraca liczbę typu int z ilością wyświetlonych znaków, oraz drukuje tekst wskazywany przez format, który zazwyczaj ujęty jest w znaki cudzysłowu, znakami mogą być dowolne drukowalne znaki oraz specjalne znaki zaczynające się od znaku **%**, które nazwane są deskryptorami formatu. W miejsce tych deskryptorów podczas drukowania zostają wstawione kolejne argumenty funkcji printf. Deskyptor formatu składa się ze znaku **%** oraz jednego ze znaków przekształcenia wyszczególnionych w tabeli 8.2.1, pomiędzy znakiem procenta, a znakiem przekształcenia mogą lecz nie muszą wystąpić w następującej kolejności:

- – (minus) – Wyrównanie argumentu do lewej strony jego pola
- Liczba określająca minimalny rozmiar pola
- . (kropka) – Oddzielająca rozmiar pola od precyzji
- Prezycja
    - Maksymalna liczba znaków dla tekstu
    - Liczba cyfr po kropce dzisiętnej dla liczb rzeczywistych
    - Minimalna liczba cyfr dla wartości całkowitej
- Jedna z liter: **h** – jeśli argument całkowity należy wypisać jako short, lub l – jeśli long

Poniżej znajduje się tabela, która zawiera znaki, które mogą być użyte do formatowania tekstu.

Znak przekształcenia	Typ argumentu	Wyjście
d, i	int	Liczba dziesiętna
o	int	Liczba ósemkowa bez znaku, bez wiodącego zera
x, X	int	Liczba szesnastkowa bez znaku, bez wiodących 0x, 0X
u	int	Liczba dziesiętna bez znaku (unsigned)
c	int	Jeden znak
s	char *	Ciąg znaków wyświetlany aż do znaku kończącego \0, lub przekroczenie liczby określonej przez precyzję
f	double	[-]m.dddddd, ilość **d** określa precyzja, domyślnie 6
e, E	double	[-]m.dddddde±xx, [-]m.ddddddE±xx

g, G	double	Liczba wypisana w formacie %e, %E jeśli wykładnik jest mniejszy niż -4 albo większy lub równy precyzji, w przeciwnym wypadku liczba wypisana w formacie %f
p	void *	Wskaźnik
%	-	Wypisanie po prostu znaku %

Tabela 8.2.1 Deskryptory formatu funkcji printf

Przykładowe użycia deskryptorów formatu znajdują się na poniższym listingu. Każdy deskryptor z pierwszych czterech instrukcji printf został zapisany pomiędzy znakami „|” aby lepiej widać było wyrównania, długości pól itp.

```
#include <stdio.h>

int main (void)
{
	int kk1 = 145E5;
	double pi = 3.14159265;
	double xx = 3.1E-7;
	char *tab = "Ala ma kota";
	long int el = 10L;
	unsigned int ui = 1E4;
	unsigned long ul = 10E5L;
	int max = 6;

	printf("|%d| \t|%f| \t|%s|\n\n", kk1, pi, tab);
	printf("|%15d| \t|%15f| \t|%15s|\n\n", kk1, pi, tab);
	printf("|%-15d| \t|%-15.8f| \t|%-15.8s|\n\n", kk1, pi, tab);
	printf("|%-15.10d| \t|%-.2f| \t|%.10s|\n\n", kk1, pi, tab);

	printf("%d %i %o %x\n\n", kk1, kk1, kk1, kk1);
	printf("%ld %u %lu %c\n\n", el, ui, ul, tab[0]);
	printf("%.2e %.10g %.7f %p %%\n", xx, xx, xx, &pi);
	printf("|%15.*s|\n", max, tab);
	printf("|%*.*s|\n", max, max, tab);

	return 0;
}
```

Listing 8.2.1 Użycie deskryptorów formatu

Ciekawostką mogą być dwie ostatnie instrukcje printf. W pierwszej precyzja posiada gwiazdkę. Za pomocą zmiennej, która jest argumentem na odpowiedniej pozycji ustala się odpowiednią wartość precyzji, w tym przypadku będzie ona miała wartość, która jest pod zmienną max, czyli 6. W drugiej rozmiar pola oraz precyzja jest ustalana na podstawie zmiennej max. Precyzja dla liczb rzeczywistych

określa ilość miejsc po przecinku, dla liczb całkowitych minimalną ilość wyświetlanych cyfr, co widać po wydruku niektórych funkcji printf, że wyświetlane są wiodące zera, a dla tekstu ilość wyświetlanych liter, jeśli wartość prezycji jest mniejsza niż ilość liter, to ciąg znaków nie zostanie wyświetlony w całości. Zwróć uwagę na wyświetlanie liczb typu long, dodajemy literkę **l** przed deskryptorem dla konkretnego typu **%ld – long int**, **%lu – unsigned long**, **%llu – unsigned long long**, itp.

W tym miejscu chcę wspomnieć o czymś co nazywa się sekwencje ucieczki (*ang. escape sequences*). W gruncie rzeczy mieliśmy z nimi styczność nie jednokrotnie, lecz nie wszystkie zostały pokazane na przykładzie. Jest to znak (znaki) poprzedzone znakiem ukośnika, które nie są drukowane (np. na ekranie monitora) w normalny sposób, tylko spełniają pewną określoną funkcję (np. znak nowej linii). Tak więc poniższa tabela, to zbiór tych znaków, a pod tabelką znajduje się przykład użycia.

Sekwencja ucieczki	**Reprezentacja**
\a	Dźwięk (alarm)
\b	Backspace
\f	Wysunięcie strony
\n	Nowa linia
\r	Powrót karetki (*ang. carriage return*)
\t	Tabulacja pozioma
\v	Tabulacja pionowa
\'	Apostrof
\”	Cudzysłów
\\	Ukośnik
\?	Znak zapytania
\ooo	Znak ASCII podany jako liczba ósemkowa
\xhh	Znak ASCII podany jako liczba szesnastkowa

Tabela 8.2.2 Sekwencje ucieczki

```
#include <stdio.h>

int main (void)
{
	char *napis = "\x50\x72\x7a\x79\x67\x6f\x64\x79 \x6b\x6f\x74\x61
\x46\x69\x6c\x65\x6d\x6f\x6e\x61";

	printf("%s\n", napis);
```

```
	printf(" \b\? \' \" \\\n");
	return 0;
}
```

Listing 8.2.2 Użycie niektórych sekwencji ucieczek

Funkcja sprintf jest prawie, że identyczna jak funkcja printf, z tą różnicą, że zamiast drukować sformatowany tekst na standardowe wyjście (monitor) to zapisuje ten ciąg znaków w tablicy podanej jako pierwszy argument. Tablica musi być odpowiednio duża. Prototyp funkcji wygląda następująco:

```
int sprintf(char *string, char *format, arg1, arg2, ...);
```

Przykład użycia funkcji sprintf został pokazany na poniższym listingu. Obie funkcje zadeklarowane są w nagłówku stdio.h.

```
#include <stdio.h>

int main (void)
{
	char tab[50];
	int wiek = 20;

	sprintf(tab, "Czesc jestem Tomek i mam %d lat", wiek);
	printf("%s\n", tab);

	return 0;
}
```

Listing 8.2.3 Użycie funkcji sprintf

8.3 Funkcja scanf i sscanf

Funkcja scanf służy do wczytywania danych z klawiatury. W jednym z poprzednich rozdziałów była użyta, lecz nie była omówiona zbyt dobrze. W tym miejscu zostanie omówiona dokładniej wraz z pokazanymi przykładami zastosowania. Prototyp funkcji wygląda następująco:

```
int scanf (char *format, arg1, arg2, ...);
```

Funkcja zwraca wartość całkowitą, która odpowiada poprawnej ilości wczytanych znaków. Argumenty muszą być wskaźnikami, czyli jako argument trzeba przekazać adres zmiennej. Funkcji scanf używa

się bardzo podobnie do funkcji printf, tzn deskryptory formatu tworzy się tak samo, a listę znaków przekształcenia przedstawia poniższa tabela.

Znak przekształcenia	Typ argumentu	Wejście
d	int *	Liczba całkowita dziesiętna
i	int *	Liczba całkowita; może wystąpić w postaci ósemkowej z wiodącym zerem, lub szesnastkowej z wiądącymi 0x, 0X
o	int *	Liczba całkowita w postaci ósemkowej z wiodącym zerem lub bez
u	unsigned int *	Liczba całkowita dziesiętna bez znaku
x	int *	Liczba całkowita w postaci szesnastkowej z wiodącym 0x, 0X lub bez
c	char *	Znak
s	char *	Tekst
e, f, g	float *	Liczba zmiennopozycyjna z opcjonalnym znakiem, opcjonalną kropką dziesiętną i opcjonalnym wykładnikiem
%	-	Znak %, nie ma żadnego przypisania

Listing 8.3.1 Deskryptory formatu funkcji scanf

Przed znakami przekształcenia **d**, **i**, **o**, **u**, **x** można wstawić literę **h**, jeśli chodzi nam o wczytanie liczby short, lub **l**, jeśli liczba ma być typu long. Dla znaków **e**, **f**, **g** litera **l** oznacza, że typem wczytywanych danych jest double, a nie float.

Przykład użycia funkcji scanf w różnych wariantach pokazuje poniższy listing.

```
#include <stdio.h>
int main (void)
{
	int x;
	double y;
	printf("Podaj liczbe calkowita: ");
	scanf("%d", &x);
	printf("Podaj liczbe rzeczywista: ");
	scanf("%lf", &y);
	printf("%d %.2f\n", x, y);

	return 0;
}
```

Listing 8.3.1 Użycie funkcji scanf

Tak zapisane użycie funkcji scanf nie jest bezpieczne, ponieważ jeśli wpiszemy literę podczas wprowadzania liczby całkowitej to wychodzą dość ciekawe rzeczy, co zresztą warto przetestować, lub wpiszemy liczbę rzeczywistą to zostanie wzięta część całkowita, a kropka i kolejne cyfry zostaną przypisane do kolejnego wywołania funkcji, czyli tego, które pobiera liczbę rzeczywistą. Dzieje się tak ponieważ funkcja scanf kończy działanie wtedy, kiedy natrafi na pierwszy nie należący znak do zbioru danego deskryptora formatu, czyli podczas wczytywania liczb całkowitych, jak wpiszemy **3.4**, to pierwsze wywołanie funkcji weźmie tylko część całkowitą. Trójka została zabrana, ale **.4** zostanie w buforze, tak więc kolejnymi znakami pobranymi przez funkcję scanf będą **.4**, ponieważ pobieramy liczbę rzeczywistą, wartość ta jest traktowana poprawnie. Ważną rzeczą jest to i o tym należy pamiętać, że scanf jako argument przyjmuje adres do zmiennej, a nie jej nazwę!

Za pomocą funkcji scanf można pobierać teksty, ale nie jest to najlepsza metoda ponieważ funkcja ta skopiuje do tablicy tylko czarne znaki, tzn, że jeśli wpiszemy spację pomiędzy wyrazami, to na spacji poprzestanie wczytywać, czyli tylko pierwsze słowo zostanie wczytane. Nie mniej jednak pojedyncze wyrazy można pobierać za pomocą scanf. Poniższy przykład pokazuje to i pewną inna rzecz.

```
#include <stdio.h>

int main (void)
{
      int dzien, rok;
      char miesiac[20];

      printf("Podaj date w formacie: Dzien Miesiac Rok\n");
      scanf("%d %s %d", &dzien, miesiac, &rok);

      printf("Dzien: %d\n", dzien);
      printf("Miesiac: %s\n", miesiac);
      printf("Rok: %d\n", rok);
      return 0;
}
```

Listing 8.3.2 Użycie scanf do pobierania słowa

W funkcji scanf jako pierwszy parametr podane jest **%d %s %d**, czyli funkcja oczekuje liczby całkowitej, białego znaku (spacja, tabulatura), słowa, białego znaku, liczby całkowitej. (Jeśli zastanawiasz się dlaczego przekazalismy adresy zmiennych dzien oraz rok, a miesiac po prostu jako nazwę, to dla przypomnienia nazwa tablicy jest wskaźnikiem). Zamiast tej spacji można wstawić inny znak, np. ukośnik, dzięki czemu wpiszemy datę w formacie RRRR/MM/DD, poniższy przykład

pokazuje to.

```
#include <stdio.h>

int main (void)
{
	int dzien, miesiac, rok;

	printf("Podaj date w formacie: RRRR/MM/DD\n");
	scanf("%d/%d/%d", &rok, &miesiac, &dzien);

	printf("Dzien: %d\n", dzien);
	printf("Miesiac: %d\n", miesiac);
	printf("Rok: %d\n", rok);

	return 0;
}
```

Listing 8.3.3 Kolejne użycie scanf – inny format daty

Przykład wczytywania liczb w innych systemach liczbowych znajduje się poniżej.

```
#include <stdio.h>

int main (void)
{
	int ld;
	printf("Podaj liczbe dziesietna: ");
	scanf("%d", &ld);
	printf("Podales liczbe: %d\n", ld);

	printf("Podaj liczbe osemkowa: ");
	scanf("%o", &ld);
	printf("Osemkowo %o to dziesietnie %d\n", ld, ld);

	printf("Podaj liczbe szesnatkowa: ");
	scanf("%x", &ld);
	printf("Hex: %x, Oct: %o, Dec: %d\n", ld, ld, ld);
	return 0;
}
```

Listing 8.3.4 Wczytywanie liczb w innych systemach liczbowych

Funkcja scanf nadaje się do pobierania liczb, lecz z tekstami sobie nie radzi. Biorąc pod uwagę fakt, że wpisanie złej liczby, bez mechanizmów zabezpieczających może wykrzaczyć cały program, lepszym rozwiązaniem jest pobieranie liczb do tablicy znaków, a następnie za pomocą odpowiedniej funkcji przekształcenie tego ciągu do liczby. Pokazane zostanie to w punkcie 8.6.

Funkcja sscanf jest podobna do scanf, różni się tylko tym, że zamiast ze standardowego wejścia czyta znaki wskazywane przez jej pierwszy argument. Deklaracja funkcji wygląda następująco.

```
int sscanf (char *string, char *format, arg1, arg2, ...);
```

Dla przykładu weźmy następujący program.

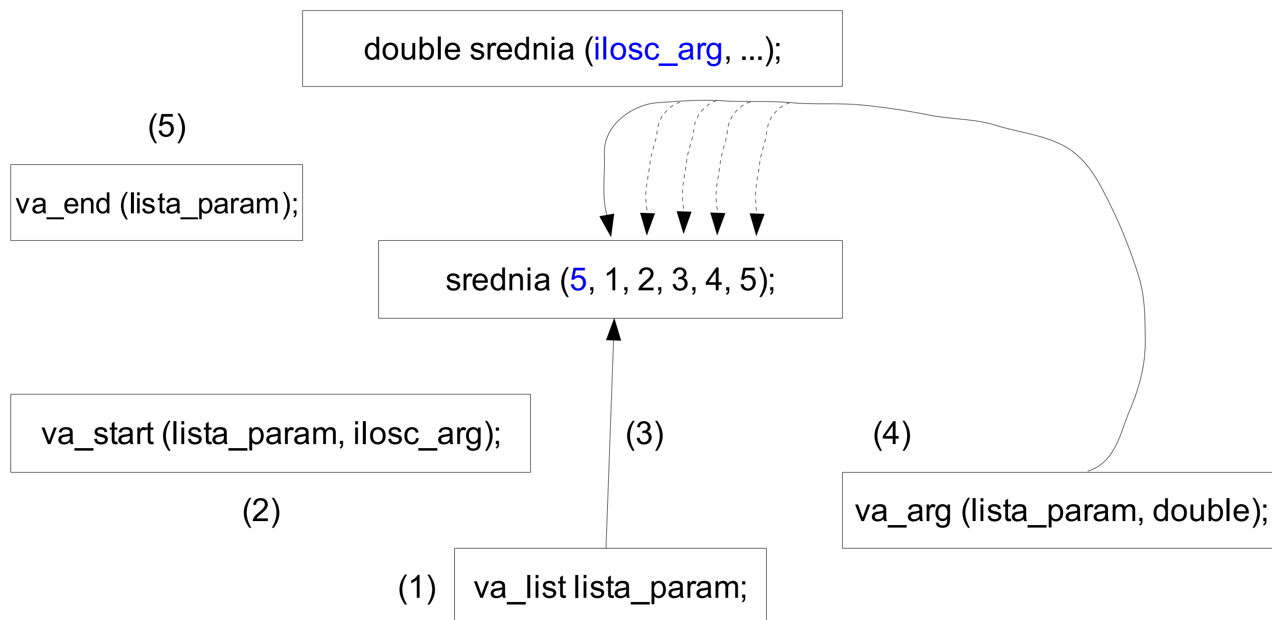
```
#include <stdio.h>

int main (void)
{
	char *napis = "02 Sierpien 2010";
	int dzien, rok;
	char miesiac[20];

	sscanf(napis, "%d %s %d", &dzien, miesiac, &rok);
	printf("Dzien: %d\n", dzien);
	printf("Miesiac: %s\n", miesiac);
	printf("Rok: %d\n", rok);

	return 0;
}
```

Listing 8.3.5 Użycie funkcji sscanf

Widać, że funkcja sscanf pobiera tekst i przekształca go nie ze standardowego wejścia, a z ciągu znaków wskazywanego przez napis, równie dobrze zamiast wskaźnika do tekstu mogłaby być tablica znaków z zainicjowanymi wartościami.

8.4 Zmienna ilość argumentów

Ze zmienną ilością argumentów mieliśmy, a właściwie mamy styczność cały czas, funkcje printf i scanf pobierają różną ilość argumentów. Z ich pomocą możemy wydrukować lub pobrać dowolną ilość wartości. Generalnie deklaracja funkcji o zmiennej ilości parametrów posiada trzy kropki (...) jako ostatni parametr. Wywołując funkcję z różną ilością argumentów funkcja nie ma ich nazw, skoro nie ma nazw, to jak się do nich odwołuje? Specjalne makra zdefiniowane w pliku nagłówkowym stdarg.h umożliwiają poruszanie się po takiej liście argumentów i zaraz pokaże jak się za to zabrać. Poniższy rysunek może zobrazować dane zagadnienie.



Rysunek 8.4.1 Zmienna ilość argumentów

Aby odwoływać się do nie nazwanych argumentów musimy utworzyć zmienną typu va_list, w naszym przypadku nazywa się ona lista_param. Makro va_start odpowiedzialne jest za to, aby zainicjować wartość zmiennej lista_param na pierwszy nie nazwany argument, dlatego też jako pierwszy argument przyjmuje zmienną typu va_list, a drugim jest ostatni nazwany parametr, w naszym przypadku jest to jedyny nazwany parametr i nazywa się ilosc_arg. Wywołanie makra va_arg wyciąga wartość, która kryje się pod aktualnie wskazywanym argumentem i przesuwa zmienną lista_param na następny argument. Aby wiedzieć o ile przesunąć zmienną lista_param oraz jaki jest typ szukanej wartości, drugim argumentem makra va_arg jest typ zmiennej. Po zakończeniu działań trzeba użyć makra va_end, które czyści wszystko to co związane było z poprzednimi wywołaniami. Kod do tego zadania jest następujący.

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

double srednia (int ilosc_arg, ...);

int main (void)
{
     printf("%.2f\n", srednia(3, 1.0, 4.0, 5.0));
     printf("%.2f\n", srednia(4, 2.0, 4.5, 5.5, 3.5));
     printf("%.2f\n", srednia(3, 1.5, 4.0, 5.5));
```

```
	printf("%.2f\n", srednia(4, 1.78, 4.34, 5.11));
	return 0;
}

double srednia (int ilosc_arg, ...)
{
	va_list lista_param;
	va_start (lista_param, ilosc_arg);
	int i;
	double suma = 0;

	for (i = 0; i < ilosc_arg; i++)
		suma += va_arg(lista_param, double);
	va_end(lista_param);
	return suma / (double) ilosc_arg;
}
```

Listing 8.4.1 Zmienna ilość parametrów

Pierwszym argumentem funkcji srednia jest ilość argumentów, będzie ona odpowiedzialna za sterowanie pętlą, żeby można było zsumować wszystkie argumenty no i oczywiście podzielić przez ich ilość, by otrzymać średnią arytmetyczną. Każde wywołanie makra va_arg wyciąga wartość wskazywanego argumentu i przesuwa zmienną lista_param na następny argument w kolejności.

8.5 Obsługa plików

Do tej pory jedyne co miało wspolnego z plikami to było przekierowanie standardowego wejścia, tudzież wyjścia przy pomocy linii poleceń, za pomocą czego mogliśmy odczytać wszystko co jest w pliku, lub zapisać coś do niego. W tym miejscu pokazane zostanie, jak za pomocą funkcji otworzyć plik, przeczytać jego zawartość, zapisać w nim coś, itp.

Przed każdą operacją na pliku plik ten trzeba otworzyć, służy do tego funkcji fopen, której deklaracja wygląda następująco.

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

Widać, że funkcja zwraca wskaźnik, ale do czego? Jakiego typu? Otóż FILE jest to struktura, która zawiera informacje o buforze, pozycji znaku, rodzaju dostępu, błędach itp. W gruncie rzeczy nie musimy się tym interesować, ważne natomiast jest jak się do tego zabrać. Załóżmy, że zadeklarujemy coś takiego.

```
FILE *wsk_plik;
```

Stworzyliśmy wskaźnik do pliku, teraz wystarczy przypisać do niego funkcję fopen z wypełnionymi parametrami, tj. z ścieżką oraz trybem dostępu do pliku. Zanim pokazana zostanie gotowa instrukcja parę słów wyjaśnienia czym jest tryb dostępu. Tryb dostępu jest informacją o tym co będzie z plikiem robione, tzn czy będzie tylko czytana zawartość pliku, czy czytana i zapisywana, w każdym bądź razie jeden z trybów trzeba podać, są one wyszczególnione w poniższej tabeli.

Tryb dostępu	Funkcja	Dodatkowe informacje
"r"	Czytanie	Otwarcie pliku tylko do czytania
"r+"	Czytanie i nadpisywanie	Otwarcie pliku do czytania oraz nadpisywania
"w"	Zapisywanie	Otwarcie pliku do zapisywania (zamazuje poprzednie dane)
"w+"	Nadpisywanie i czytanie	Otwarcie pliku do nadpisywania i czytania
"a"	Dopisywanie	Otwarcie pliku w celu dopisywania (jeśli plik nie istnieje, to jest tworzony – jeśli się da)
"a+"	Dopisywanie i czytanie	Otwarcie pliku w celu dopisywania (jeśli plik nie istnieje, to jest tworzony – jeśli się da) i czytania
"t"	Tryb tekstowy	Otwarcie pliku w trybie tekstowym
"b"	Tryb binarny	Otwarcie pliku w trybie binarnym

Tabela 8.5.1 Tryby dostępu

Mała uwaga, tryby można łączyć, tzn aby otworzyć plik do czytania, zapisu i w trybie binarnym to należy jako tryb podać "rwb".

Jeśli w katalogu, w którym jest plik wykonywalny znajduje się plik **sport.txt** to aby go otworzyć w trybie zapisu należy wpisać taką instrukcję:

```
wsk_plik = fopen("sport.txt", "w");
```

Oczywiście wpisywanie na sztywno nazwy pliku w kodzie źródłowym nie jest dobrym pomysłem, bo za każdym razem musielibyśmy kompilować program na nowo, jeśli nazwa pliku by się zmieniła. Dobrym, a zarazem wygodnym sposobem jest podanie nazwy pliku jako argument wywołania programu. Poniższy listing prezentuje podstawowe użycie omówionych instrukcji.

```
#include <stdio.h>

int main (int argc, char *argv[])
{
```

```
	FILE *wsk_plik;

	if (argc != 2)
	{
		printf("Uzycie: %s nazwa_pliku\n", argv[0]);
		return -1;
	}

	if ((wsk_plik = fopen(argv[1], "w")) != NULL)
	{
		printf("Plik otwarto.\n");
		fprintf(wsk_plik, "Ala ma kota, a kot ma %d lat\n", 5);
		fclose(wsk_plik);
	}
	return 0;
}
```

Listing 8.5.1 Zapisanie informacji w pliku

Funkcja fopen w razie nie powodzenia zwraca wartość NULL, dlatego też sprawdzamy warunek, jeśli otwarto plik poprawnie, to zostanie wyświetlona informacja o tym, oraz za pomocą dotąd nie omawianej funkcji fprintf tekst zostaje zapisany w pliku. Obowiązkowo po skończeniu operacji na pliku trzeba go zamknąć za pomocą funkcji fclose, w której jako argument podajemy wskaźnik do pliku.

Funkcja fprintf jak już zresztą pewnie widzisz jest analogiczna jak printf, z małą różnicą. Jako pierwszy argument podajemy wskaźnik do pliku, deklaracja tej funkcji wygląda następująco.

```
int fprintf (FILE *wsk_file, char *format, ...);
```

Wywołanie funkcji fprintf z pierwszym argumentem jako stdout (standard out) tj.

```
fprintf(stdout, "%s %s\n", tab[0], tab[1]);
```

daje efekt dokładnie taki sam jak printf, czyli informacje zostaną wydrukowane na ekranie.

Istnieje funkcja fscanf, która jest analogiczna do scanf z tą różnicą, że pobiera tekst z pliku, czyli tak samo jak w przypadku poprzedniej funkcji jako pierwszy argument podajemy wskaźnik do pliku. Deklaracja jej wygląda tak.

```
int fscanf(FILE *wsk_file, char *format, ...);
```

A użycie może być następujące.

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

```
int main (int argc, char *argv[])
{
      FILE *wsk_plik;
      char dane[3][10];

      if (argc != 2)
      {
            printf("Uzycie: %s nazwa_pliku\n", argv[0]);
            return -1;
      }

      if ((wsk_plik = fopen(argv[1], "r")) != NULL)
      {
            fscanf(wsk_plik, "%s %s %s", dane[0], dane[1], dane[2]);
            fclose (wsk_plik);
            printf("%s %s %s\n", dane[0], dane[1], dane[2]);
      }
      return 0;
}
```

Listing 8.5.2 Użycie fscanf

Analogicznie i dla tej funkcji z drobną różnicą można wymusić, aby wczytywała znaki z klawiatury, różnica polega na tym, że zamiast wskaźnika do pliku podajemy stdin (standard in). A więc poniższe polecenie wczyta liczbę całkowitą z klawiatury.

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

stdin oraz stdout zwane są strumieniami, pierwszy odnosi się do wejścia, drugi do wyjścia. Jest jeszcze trzeci, nazwany stderr, który to jest wyjściem błędów. Zarówno stdout i stderr drukują informacje na standardowe wyjście, lecz jest między nimi subtelna różnica, którą na poniższych przykładach pokażę.

```
#include <stdio.h>

int main (int argc, char *argv[])
{
      FILE *wsk_plik;

      if (argc != 2)
      {
            printf("Uzycie: %s nazwa_pliku\n", argv[0]);
            return -1;
      }

      if ((wsk_plik = fopen(argv[1], "r")) != NULL)
```

```
    {
        printf("Plik zostal pomyslnie otwarty\n");
        fclose(wsk_plik);
    }
    return 0;
}
```

Listing 8.5.3 Wyświetlanie błędów na stdout

```
#include <stdio.h>

int main (int argc, char *argv[])
{
    FILE *wsk_plik;

    if (argc != 2)
    {
        fprintf(stderr, "Uzycie: %s nazwa_pliku\n", argv[0]);
        return -1;
    }

    if ((wsk_plik = fopen(argv[1], "r")) != NULL)
    {
        fprintf(stderr, "Plik zostal pomyslnie otwarty\n");
        fclose(wsk_plik);
    }
    return 0;
}
```

Listing 8.5.4 Wyświetlanie błędów na stderr

Te dwa listingi jak widać różnią się tylko instrukcją drukującą komunikat o błędzie i powodzeniu. Skoro obie funkcje drukują informacje na ekranie to co za różnica, której się używa? Otóż jest różnica i zaraz to zasymulujemy. Program podczas uruchamiania pobiera argumenty, a konkretnie jeden i powinna to być nazwa pliku. Tak więc po kompilacji uruchom programy w następujący sposób.

```
$ ./pierwszy plik1.txt > info.txt
```

A następnie wpisz

```
$ cat info.txt
```

Jeśli plik1.txt istniał to otrzymasz wiadomość, że plik został pomyślnie otwarty, jeśli nie istniał to info.txt będzie puste. W przypadku drugiego programu, jeśli otwierany plik nie istniał, to info.txt

będzie puste, jeśli plik istniał informacja zostanie wyświetlona na ekranie, nie w pliku – to jest właśnie ta różnica. Niech kolejny przypadek dobitnie pokaże różnicę między tymi strumieniami oraz uświadomi nam, który lepiej używać do sygnalizacji błędów. Program sprawdza czy były podane dwa parametry, jeśli podamy trzy, to zasygnalizuje błąd, tak więc wpisz:

```
$ ./pierwszy plik1.txt plik2.txt > info.txt
```

I wyświetl zawartość pliku info.txt, tak samo jak w poprzednim przykładzie. Widać, że dostaniemy komunikat **Uzycie: ./pierwszy nazwa_pliku**. To jest błąd, a nie informacja, która może nam się przydać, więc nie powinna znaleźć się w pliku. Uruchamiając drugi program w ten sam sposób dostaniemy informację taką samą, lecz wyświetloną na ekranie, a plik info.txt będzie pusty. Tym się właśnie różni strumień stderr od stdout.

8.6 Pobieranie i wyświetlanie całych wierszy tekstów – funkcje: fgets, fputs

Do pobierania oraz drukowania całych wierszy można użyć funkcji fgets oraz fputs zdefiniowanych w bibliotece standardowej. Deklaracja funkcji fgets wygląda następująco:

```
char *fgets (char *line, int maxline, FILE *wsk_plik);
```

Jako trzeci argument podajemy wskaźnik do pliku, z którego funkcja ta czyta po kolei wiersze. Po przeczytaniu jednego wiersza wstawia ten ciąg znaków do tablicy znakowej line łącznie ze znakiem nowej linii **\n**. fgets czyta maxline-1 znaków dlatego, że każdy przeczytany i wstawiony do tablicy znakowej wiersz zakończony jest znakiem **\0**. Jako wskaźnik do pliku można podać również stdin, co wiąże się z tym, że funkcja będzie czytała wiersz ze standardowego wejścia (czyli klawiatury). Użycie fgets można zobaczyć w poniższym programie, który przyjmuje jako argument wywołania funkcji nazwę pliku i drukuje zawartość pliku na standardowe wyjście numerując wiersze.

```
#include <stdio.h>
#define MAX 1000

int main (int argc, char *argv[])
{
      FILE *wsk_plik;
      char *progName, *fileName;
      unsigned int counter = 1;
      char line[MAX];
```

```
	progName = argv[0];
	fileName = argv[1];

	if (argc != 2)
	{
		fprintf(stderr, "Uzycie: %s nazwa_pliku\n", progName);
		return -1;
	}

	if ((wsk_plik = fopen(fileName, "r")) != NULL)
	{
		while (fgets(line, MAX, wsk_plik) != NULL)
		{
			printf("%2d: %s", counter, line);
			counter++;
		}
		fclose(wsk_plik);
	}
	else
	{
		fprintf(stderr, "%s: Nie moge otworzyc pliku: %s\n", progName, 
fileName);
		return -1;
	}
	return 0;
}
```

Listing 8.6.1 Użycie fgets

Deklarujemy wskaźnik do pliku za pomocą pierwszej instrukcji w funkcji main, następnie co nie było konieczne, lecz może ułatwić czytanie kodu deklarujemy dwa wskaźniki do tekstu, które odpowiednio będą przechowywać nazwę programu i nazwę pliku. Trzeba utworzyć licznik, który będzie przechowywał ilość wierszy, deklarujemy tę zmienną jako zmienną bez znaku, wierszy ujemnych przecież nie ma, a unsigned jak wiadomo podwaja zakres zmiennej. Potrzebujemy tablicę znaków do której wiersz zostanie skopiowany, tak więc tworzymy tablicę line o rozmiarze MAX, który jest stałą symboliczną. Funkcja fgets po wykryciu konca pliku zwraca wartość NULL, tak więc pętla while będzie wykonywać się dopóki, dopóty koniec pliku nie zostanie wykryty. W przypadku sukcesu funkcja zwraca wskaźnik do tablicy line. Po wykryciu końca pliku, plik jest zamykany, a program kończy swoje działanie. Należy zwrócić uwagę na linię:

```
printf("%2d: %s", counter, line);
```

Jak widać, nie trzeba wpisywać znaku nowego wiersza, ponieważ jak wspomniano fgets pobiera znak nowego wiersza i przypisuje go do tablicy. Prototyp funkcji fputs wygląda tak:

```
int fputs (char *line, FILE *wsk_plik);
```

Funkcja zwraca zero, jeśli wszystko poszło dobrze i NULL w razie nie powodzenia. Jako pierwszy argument funkcja przyjmuje tablicę znaków, którą drukuje do pliku wskazywanego przez drugi argument. Podobnie jak przy poprzedniej funkcji, drugim argumentem może być strumień stdout, który wydrukuje daną tablicę na ekran monitora. Przykład użycia został pokazany na poniższym listingu.

```
#include <stdio.h>
#define MAX 1000

int main (int argc, char *argv[])
{
      FILE *wsk_plik;
      char *progName = argv[0], *fileName = argv[1];
      char line[MAX];

      if (argc != 2)
      {
            fprintf(stderr, "Uzycie: %s nazwa pliku\n", progName);
            return -1;
      }

      if ((wsk_plik = fopen(fileName, "a")) != NULL)
      {
            printf("Wpisz wiersz: ");
            fgets(line, MAX, stdin);
            if (fputs(line, wsk_plik) != EOF)
                  fprintf(stderr, "Wiersz pomyslnie zapisany w pliku: %s\n",
fileName);
            else
                  fprintf(stderr, "Nie udalo mi sie zapisac wiersza w pliku:
%s\n", fileName);

            fclose(wsk_plik);
      }
      else
      {
            fprintf(stderr, "%s: Nie moge otworzyc pliku: %s\n", progName,
fileName);
            return -1;
      }
      return 0;
}
```

Listing 8.6.2 Użycie fputs

Początek jest w zasadzie taki sam, tak więc omówienie zacznę od miejsca w którym wpisujemy tekst do tablicy. Tak jak powiedziane było, jeśli trzecim argumentem funkcji fgets będzie stdin, to zamiast pobierać tekst z pliku pobieramy go ze standardowego wejścia. Po pobraniu wiersza sprawdzamy czy fputs zwraca EOF, który oznacza błąd. Jeśli błędu nie było, to na stderr wysyłamy informację o pomyślnym zapisaniu wiersza. Jeśli jednak jakiś błąd wystąpił to wyświetlamy stosowny do takiej sytuacji komunikat. Teraz pytanie, jak sprowokować błąd zapisu, aby zobaczyć informację o nie udanym zapisie oraz błąd otwarcia pliku (skoro tyb „a” tworzy plik, jeśli go nie ma)? Najpierw uruchomimy program podając jako parametr nazwę pliku, który istnieje, bądź nie istnieje i w takim wypadku zostanie utworzony, czyli na przykład:

```
$ ./nazwa_programu plik1.txt
```

Plik został utworzony, wpisany wiersz został dopisany, czyli program działa tak jak powinien. Teraz mała uwaga, w systemach Linux występują prawa dostępu do pliku, co w bardzo podstawowym stopniu opisano w dodatku A, podpunkt 4 i 6. Aby otwarcie pliku się nie powiodło musimy usunąć prawa dostępu do pliku, wpisz w konsoli poniższe polecenie:

```
$ chmod 0 plik1.txt
```

Teraz po uruchomieniu programu z argumentem **plik1.txt** dostaniemy poniższy komunikat, ponieważ system operacyjny nie pozwoli na otwarcie pliku:

```
./nazwa_programu: Nie moge otworzyc pliku: plik1.txt
```

Jeśli chcemy z powrotem nadać prawa dostepu do pliku wpisz np. tak:

```
$ chmod 750 plik1.txt
```

Aby otrzymać komunikat o nie powodzeniu podczas zapisywania wiersza do pliku można na przykład spróbować otworzyć plik w trybie „r”, próba zapisu do pliku, kiedy otwarty jest on w trybie czytania kończy się błędem.

9 Dynamicznie przydzielana pamięć

Do dynamicznie przydzielanej pamięci stosuje się dwie funkcje zdefiniowane w pliku nagłówkowym stdlib.h, a mianowicie malloc, oraz calloc, w zasadzie obie zwracają wskaźnik do pewnego obszaru, aczkolwiek różnią się małym szczegółem, o którym zaraz powiem. Deklaracja funkcji malloc wygląda następująco:

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

Funkcja ta w przypadku powodzenia zwraca wskaźnik do n bajtów nie zainicjowanej pamięci. Jeśli nie udało się zarezerwować miejsca, to funkcja zwraca NULL. Deklaracja funkcji calloc wygląda następująco:

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

Funkcja calloc zwraca wskaźnik do n · size bajtów, czyli tyle miejsca, aby pomieścić tablicę n elementów o rozmiarze size. W razie nie powodzenia, analogicznie do poprzedniej funkcji – zwraca NULL. size_t jest odwołaniem do pewnego typu danych (typ całkowity bez znaku) do określania długości, rozmiaru itp. Nic nie stoi na przeszkodzie by używać podstawowego typu danych jak np. unsigned int. Przydzielona pamięć przez funkcję calloc inicjowana jest zerami.

Jeszcze jedna uwaga do obu tych funkcji. Wartość wskaźnika trzeba zrzutować na konkretny typ danych. Poniższy przykład pokazuje jak za pomocą funkcji malloc zarezerwować n miejsc dla pseudolosowych liczb typu int.

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

int main (int argc, char *argv[])
{
	size_t iloscElementow;		// unsigned int iloscElementow;
	int *wsk, i;

	if (argc != 2)
	{
		fprintf(stderr, "Uzycie: %s ilosc_elementow_tablicy\n", 
argv[0]);
		return -1;
	}

	srand(time(0));
```

```
    iloscElementow = atoi(argv[1]);
    if ((wsk = (int *)malloc (iloscElementow * sizeof (int))) == NULL)
    {
        fprintf(stderr, "Pamieci nie udalo sie zarezerwowac\n");
        return -1;
    }

    for (i = 0; i < iloscElementow; i++)
        wsk[i] = rand() % 100;

    for (i = 0; i < iloscElementow; i++)
        printf("%d: %d\n", i, wsk[i]);
    free(wsk);
    return 0;
}
```

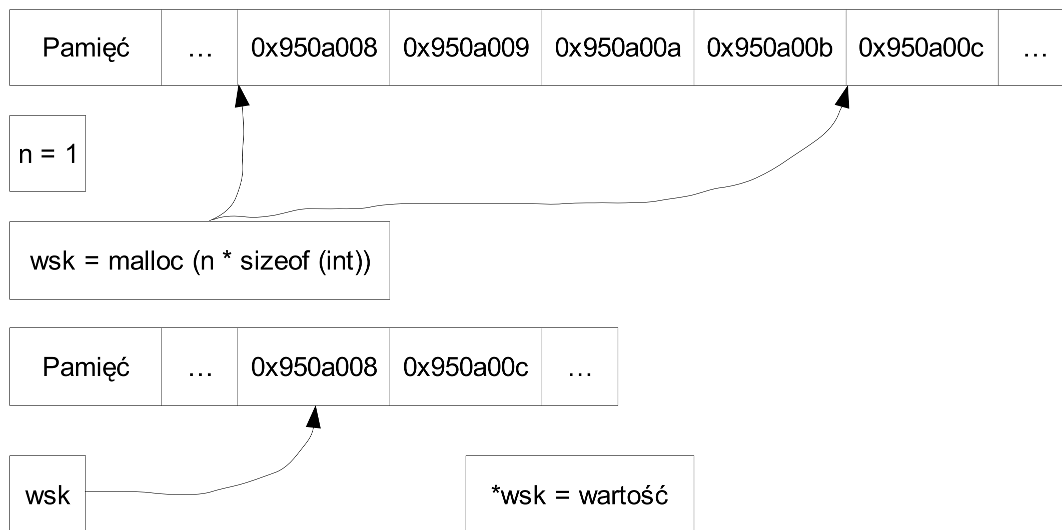
Listing 9.1 Użycie funkcji malloc

O funkcji srand, rand i time powiem za chwilę, najpierw to co nas interesuje najbardziej. Tworzymy wskaźnik do typu całkowitego i następnie przypisujemy do niego wywołanie funkcji malloc, która jako argument przyjmuje ilość bajtów, które ma zarezerwować. Ilością tą jest ilość elementów podanych jako argument wywołania programu razy rozmiar typu int. Całość musimy zrzutować na odpowiedni typ, czyli w naszym przypadku wskaźnik na int (int *). Następnie sprawdzamy, czy wskaźnik równa się NULL, jeśli tak, to trzeba przerwać działanie programu, bo nie można się odwoływać do nie zaalokowanej pamięci. Absolutnym obowiązkiem jest zwolnienie pamięci wtedy, kiedy już nie jest używana. W tak małym programie jak ten z listingu 9.1 nic się nie stanie jeśli nie użyjemy funkcji free, ponieważ program zakończy się po wydrukowaniu wartości, a wraz z końcem programu pamięć jest zwalniana. Nie mniej jednak czasem rezerwacja pamięci może odbywać się w innych funkcjach i nie zwolnienie pamięci wiąże się z tym, że dopóki program działa, pamięć ta nie może być po raz drugi zaalokowana. Pamięć zwalnia się za pomocą funkcji free, której argumentem jest wskaźnik do zaalokowanej wcześniej pamięci przez funkcję malloc lub calloc. Nie można zwalniać pamięci, która nie była prędzej zaalokowana oraz odwoływać się do pamięci, która została zwolniona. Jak widać, w pętli odwołujemy się do elementów jak do tablicy i nie jest to przypadek, ponieważ ilością zaalokowanej pamięci jest **ilość elementów · rozmiar typu**, a skoro wszystkie elementy są obok siebie w pamięci, to możemy przesuwać wskaźnik o konkretną ilość miejsc, tzn o rozmiar elementu, w gruncie rzeczy na tej samej zasadzie działają tablice, dlatego możemy indeksować zmienną wskaźnikową. Alternatywnym (w tym wypadku może bardziej zrozumiałym) sposobem było by

przesuwanie wskaźnika, np. w ten sposób *(wsk+i). Obie operacje są sobie równoważne. Użyliśmy funkcji malloc do zaalokowania miejsca dla tablicy, równie dobrze można by było wywołać funkcję calloc w następujący sposób:

```
wsk = (int *)calloc (iloscElementow, sizeof (int));
```

Rysunek może zobrazuje sposób alokacji pamięci przez funkcję z rodziny alloc.



Rysunek 9.1 Obrazowe przedstawienie sposobu alokacji pamięci

Funkcja rand zwraca pseudolosową liczbę, jeśli użylibyśmy tylko funkcji rand to raz skompilowany program wyświetlałby zawsze te same liczby, więc to nas nie zadowala. Do losowania różnych liczb używa się mechanizmu, który nazywa się zarodkiem, realizuje to funkcja srand, ale i tym razem jeśli podamy w argumencie stałą liczbę, to funkcja rand wylosuje nam pewne liczby i przy każdym uruchomieniu będzie pokazywała te same wartości, to też nas nie zadowala. W tej sytuacji ratuje nas funkcja time. Funkcja time zwraca informację o czasie, wywołanie funkcji z parametrem 0 daje ilość sekund, które upłynęły od 1 stycznia 1970 roku, więc przy każdym uruchomieniu ilość ta jest inna, tak więc dostajemy liczby pseudolosowe. Wywołanie instrukcji rand() % 100 tworzy przedział liczb losowych od 0 do 99.

10 Biblioteka standardowa

W rozdziale tym przedstawione zostaną funkcje z biblioteki standardowej. Biblioteka standardowa jest to zestaw plików nagłówkowych, w których zawarte są definicje poszczególnych funkcji, typów oraz makr. Do tej pory zdarzyło nam się już wykorzystywać np. funkcje printf, scanf, tudzież malloc. Deklaracje tych funkcji (oraz wielu innych) znajdują się w plikach nagłówkowych, które trzeba dołączyć za pomocą include. Plikami nagłówkowymi opisanymi w tej książce wchodzącymi w skład biblioteki standardowej są:

assert.h	complex.h	ctype.h	errno.h	iso646.h	limits.h
locale.h	math.h	setjmp.h	signal.h	stdarg.h	stdbool.h
stdio.h	stdlib.h	string.h	time.h		

W kolejnych podpunktach niniejszego rozdziału zostaną omówione wyżej wymienione pliki nagłówkowe zawierające bardzo użyteczne funkcje. Pomimo wielkich starań materiał przedstawiony w tym punkcie nie stanowi kompletnego kompedium wiedzy z zakresu biblioteki standardowej, aby taką uzyskać można wejść na stronę http://www.gnu.org/software/libc/manual/ na której w kilku różnych formatach przedstawiona jest biblioteka języka C (*ang. The GNU C Library*) bardzo szczegółowo (wersja pdf ma ponad 1000 stron).

10.1 assert.h

W pliku nagłówkowym assert.h znajduje się zdefiniowane makro assert, które służy do testowania programu, w celu wykrycia (jeśli są jakieś) błędów. assert zdefiniowane jest następująco:

```
void assert (int expression);
```

Jako parametr podajemy wyrażenie, które po obliczeniu przyjmuje wartość niezerową, bądź zero. Jeśli expression przyjmuje wartość zero, to za pomocą assert działanie programu zostaje przerwane i na standardowe wyjście błędów (stderr) zostaje wyświetlona informacje zgodna z następującym schematem. Jeśli argument ma wartość niezerową to nic się nie dzieje.

```
file: source.c:linenum: function: Assertion `expression' failed.
Aborted
```

Gdzie:

- file – nazwa skompilowanego programu
- source.c – nazwa pliku źródłowego
- linenum – numer linii, w którym występuje assert
- function – nazwa funkcji
- expression – argument assert

Przykład użycia został pokazany na poniższym listingu.

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

int main (void)
{
      int x = 1, y = 0;
      int w1, w2;
      w1 = x || y;
      w2 = x && y;

      assert(w1);
      assert(w2);
      return 0;
}
```

Do zmiennych w1 i w2 przypisujemy wynik operacji logicznych na zmiennych x i y. Zmienna w1 będzie miała wartość jeden (zgodnie z tabelą 2.3.6), a zmienna w2 wartość zero (zgodnie z tabelą 2.3.3). Pierwsze wywołanie assert nie wywołuje żadnej akcji, ponieważ w1 ma wartość jeden. Natomiast drugie wywołanie, gdzie argument ma wartość zerową zakończy program ze stosownym komunikatem. Funkcja assert powinina być używana raczej do kontrolowania programu, a nie do informowania użytkownika, że coś poszło nie tak. Dajmy na to przykład taki, użytkownik podaje dwie liczby, z których ma zostać wyświetlony iloraz. Jeśli jako drugą liczbę poda wartość zero, to lepiej, żeby program napisany był w taki sposób, aby użytkownik mógł podać jeszcze raz inną liczbę aniżeli program by całkowicie kończył działanie z komunikatem, który dla zwykłego użytkownika będzie najprawdopodobniej nie zrozumiały.

10.2 complex.h

Nagłówek complex.h zawiera definicje funkcji używanych do arytmetyki liczb zespolonych. Funkcji tych jest dość sporo i zostały wypisane w poniższej tabeli. Przykład zawierający operacje na liczbach zespolonych został pokazany pod tabelą. Mała uwaga, każda z funkcji posiada trzy wersje, które różnią się typem zwracanych wartości. W tabeli zostały wypisane wersje, w których typem zwracanej wartości jest double, lub double complex. Dodatkowo są funkcje zwracające liczby typu float, float complex oraz long double, long double complex. Aby to osiągnąć do nazwy funkcji dodajemy literę odpowiednio f, bądź l (el). Na listingu zostało to pokazane.

Prototyp funkcji	Opis
double cabs (double complex)	Moduł liczby zespolonej
double carg (double complex)	Argument liczby zespolonej
double cimag (double complex)	Część urojona liczby zespolonej
double creal (double complex)	Część rzeczywista liczby zespolonej
double complex csqrt (double complex)	Pierwiastek z liczby zespolonej
double complex cacos (double complex)	Funkcja odwrotna do cosinusa
double complex cacosh (double complex)	Funkcja odwrotna do cosinusa hiperbolicznego
double complex casin (double complex)	Funkcja odwrotna do sinusa
double complex casinh (double complex)	Funkcja odwrotna do sinusa hiperbolicznego
double complex catan (double complex)	Funkcja odwrotna do tangensa
double complex catanh (double complex)	Funkcja odwrotna do tangensa hiperbolicznego
double complex ccos (double complex)	Funkcja cosinus
double complex ccosh (double complex)	Funkcja cosinus hiperboliczny
double complex cexp (double complex)	Funkcja eksponencjalna
double complex clog (double complex)	Logarytm
double complex conj (double complex)	Sprzężenie zespolone
double complex cpow (double complex, double complex)	Potęgowanie
double complex csin (double complex)	Funkcja sinus
double complex csinh (double complex)	Funkcja sinus hiperboliczny
double complex ctan (double complex)	Funkcja tangens
double complex ctanh (double complex)	Funkcja tangens hiperboliczny

Dajmy na to przykład następujący, dana jest liczba z określona wzorem w postaci algebraicznej:

$$z = 4 + 5i$$

Wyświetlić moduł, argument, część rzeczywistą i urojoną liczby z oraz sprzężonej liczby zespolonej z oraz kąty jakie obie liczby tworzą z dodatnim kierunkiem osi Re.

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

int main (void)
{
      double complex z = 4+5i;
      double complex z_ = conj(z);

      double mod = cabs(z);
      double re = creal(z);
      double im = cimag(z);
      double arg1, arg2;

      arg1 = atan(im/re);
      arg1 *= (180/M_PI);
      printf("Modul: %lf\n", mod);
      printf("Re{z}: %lf\n", re);
      printf("Im{z}: %lf\n", im);
      printf("Arg{z}: %lf\n", arg1);

      mod = cabs(z_);
      re = creal(z_);
      im = cimag(z_);
      arg2 = atan(im/re);
      arg2 *= (180/M_PI);
      printf("Modul:  %lf\n", mod);
      printf("Re{z_}: %lf\n", re);
      printf("Im{z_}: %lf\n", im);
      printf("Arg{z_}: %lf\n", arg2);

      return 0;
}
```

Nowością może okazać się sposób deklarowanie liczby zespolonej, po typie danych dodajemy słowo kluczowe complex. Liczbę zespoloną można zapiać jeszcze na takie sposoby:

```
double complex z = 4 + 5I;

double complex z = 4 + 5 * I;
```

Ponieważ funkcje cabs, carg, cimag oraz creal zwracają wartości typu double – przypisujemy je do zmiennych typu double. Funkcja conj zwraca wartość typu double complex, więc do takiej zmiennej musimy przypisać jej wywołanie. Argument (czyli kąt pomiędzy modułem, a dodatnim kierunkiem osi Re) obliczamy za pomocą funkcji odwrotnej do tangensa, czyli atan. Wynik jest w radianach, tak więc aby zamienić na stopnie użyto formuły:

$$\frac{radians \cdot 180^{\circ}}{\pi}$$

Analogicznie jest z liczbą sprzężoną do z. Do zmiennej z_ typu double complex przypisujemy wywołanie funkcji conj, która zwraca liczbę sprzężoną do z. Następnie wyciągamy z niej część rzeczywistą i urojoną za pomocą tych samych funkcji co poprzednio. Istotną informacją jest to, że jeśli zmienna z_ nie będzie typu double complex, to kompilator nie poinformuje nas o tym, a przypisaną liczbą nie będzie liczba zespolona, tylko część rzeczywista ze sprzężonej liczby zespolonej, w naszym przypadku 4. Więc jeśli będziemy chcieli wyświetlić część urojoną i argument takiej liczby, to dostaniemy w obydwu przypadkach zero.

Na początku powiedziane było, że każda z tych funkcji ma swoje odpowiedniki dla typu float, oraz long double. Tak więc na poniższym przykładzie będzie wyjaśniona zasada używania tych funkcji.

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

int main (void)
{
      double complex z = 5 + 8i;
      float arg = cargf(z);
      float abs = cabsf(z);
      long double complex asin = casinl(z);

      arg *= (180/M_PI);
      printf("cargf(z): %f\n", arg);
      printf("cabsf(z): %f\n", abs);

      printf("%.20Lf\n%.20Lfi\n", creall(asin), cimagl(asin));

      return 0;
}
```

Jak widać jedyną różnicą jest to, że po nazwie funkcji dodajemy literkę f w przypadku float, i l w przypadku long double. Trzeba pamiętać jeszcze o tym, że jeśli chcemy wyświetlić część

rzeczywistą i urojoną dla zmiennej asin, to musimy użyć odpowiedników long double (creall, cimagl).

10.3 ctype.h

Nagłówek ctype.h zawiera deklaracje funkcji sprawdzających i wyświetlających informacje na temat znaków. W poniższej tabeli znajduje się lista tych funkcji wraz z ogólnym opisem.

Nazwa funkcji	Opis
isalnum	Funkcja sprawdzająca czy znak jest znakiem alfanumeryczny
isalpha	Funkcja sprawdzająca czy znak jest znakiem z alfabetu
isblank	Funkcja sprawdzająca czy znak jest znakiem odstępu (spacja, tabulacja)
iscntrl	Funkcja sprawdzająca czy znak jest znakiem sterującym
isdigit	Funkcja sprawdzająca czy znak jest liczbą dziesiętną
isgraph	Funkcja sprawdzająca czy znak jest znakiem drukowalnym różnym od spacji
islower	Funkcja sprawdzająca czy znak jest małą literą
isprint	Funkcja sprawdzająca czy znak jest znakiem drukowalnym (razem ze spacją)
ispunct	Funkcja sprawdzająca czy znak jest znakiem przestankowym[1]
isspace	Funkcja sprawdzająca czy znak jest białym znakiem[2]
isupper	Funkcja sprawdzająca czy znak jest dużą literą
isxdigit	Funkcja sprawdzająca czy znak jest liczbą szesnastkową[3]

Każda z wyżej wymienionych funkcji zwraca wartość niezerową jeśli jest prawda i zero jeśli jest fałsz. Prototypy funkcji są takie same dla wszystkich, dlatego też pokażę tylko jeden, mianowicie:

```
int alnum (int c);
```

Istnieją jeszcze dwie funkcje zamieniające wielkość liter, a mianowicie tolower oraz toupper. Odpowiednio pierwsza zamienia duży znak na mały (jeśli był mały, to nic się nie dzieje), druga zamienia mały znak na duży (jeśli był duży to nic się nie dzieje). Obie funkcje przyjmują znak jako argument. Przykład użycia tych funkcji został pokazany na poniższym listingu.

1 Wszystkie znaki drukowalne dla których funkcje isspace oraz isalnum zwracają wartość zero.
2 Białe znaki: spacja, tabulacja pozioma (\t) , tabulacja pionowa (\v), przejście do nowej linii (\n), powrót karetki (\r), wysunięcie strony (\f).
3 Znaki: cyfry od 0 do 9, litery od a – f bez względu na wielkość.

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

int main (void)
{
	char msg[200];
	int i;
	int count[12];

	char *info[] = {"alnum: ", "alpha: ", "blank: ", "cntrl: ",
			"digit: ", "graph: ", "lower: ", "print: ",
			"punct: ", "space: ", "upper: ", "xdigit: "};

	for (i = 0; i < sizeof (count) / sizeof (int); i++)
		count[i] = 0;

	printf("> ");
	fgets(msg, sizeof (msg), stdin);

	for (i = 0; msg[i] != '\0'; i++)
	{
		if (isalnum(msg[i])) count[0]++;
		if (isalpha(msg[i])) count[1]++;
		if (isblank(msg[i])) count[2]++;
		if (iscntrl(msg[i])) count[3]++;
		if (isdigit(msg[i])) count[4]++;
		if (isgraph(msg[i])) count[5]++;
		if (islower(msg[i])) count[6]++;
		if (isprint(msg[i])) count[7]++;
		if (ispunct(msg[i])) count[8]++;
		if (isspace(msg[i])) count[9]++;
		if (isupper(msg[i])) count[10]++;
		if (isxdigit(msg[i])) count[11]++;
	}

	for (i = 0; i < sizeof (count) / sizeof (int); i++)
		printf("%s%d\n", info[i], count[i]);

	for (i = 0; msg[i] != '\0'; i++)
		msg[i] = tolower(msg[i]);
	printf("%s", msg);

	for (i = 0; msg[i] != '\0'; i++)
		msg[i] = toupper(msg[i]);
	printf("%s", msg);

	return 0;
}
```

Instrukcja for jako warunek przyjmuje wyrażenie, które sprawdza, czy dany znak jest różny od znaku

końca tablicy. Jak pamiętamy z poprzednich rozdziałów za pomocą funkcji fgets przypisujemy tekst do tablicy wraz ze znakiem końca \0, dzięki temu łatwo możemy ustalić kiedy pobrany tekst się skończył i przerwać sprawdzanie warunków. Warunki zamieszczone w pętli sprawdzają po kolei, czy znak z i-tej iteracji jest znakiem alfanumerycznym, znakiem z alfabetu itp. Jeśli tak jest to zwiększany jest licznik wystąpień, który jest tablicą dwunasto elementową. Dla zaoszczędzenia miejsca instrukcja inkrementacji została zapisana w tej samej linii co warunek. Na górze programu została utworzona tablica wskaźników do typu znakowego, aby podczas drukowania wypisywać za co dany licznik odpowiada w łatwiejszy sposób.

10.4 errno.h

Nagłówek errno.h zawiera zdefiniowane makro do informowania o błędach. Błędów tych jest dość dużo, wszystkich tutaj nie opiszę, natomiast pokażę sposób używania tego mechanizmu oraz jak wyświetlić wszystkie możliwe błędy. Do przechowywania numeru błędu służy specjalna zmienna errno przechowująca kod ostatniego błędu. Opis błędu spod tego numeru można otrzymać za pomocą funkcji strerror, której prototyp wygląda następująco.

```
char *strerror (int no_err);
```

Jak widać funkcja zwraca wskaźnik do tekstu, a jako argument przyjmuje numer błędu, czyli to co kryje się pod zmienną errno. Poniższy przykład pokazuje jak wyświetlić listę wszystkich błędów (numer błędu wraz z opisem).

```
#include <stdio.h>
#include <errno.h>
int main (void)
{
      int i;
      char *opis;

      for (i = 0; i < 132; i++)
      {
            errno = i;
            opis = (char *)strerror(errno);
            printf("Errno: %d - %s\n", errno, opis);
      }
      return 0;
}
```

Do wskaźnika opis przypisujemy otrzymany wskaźnik od funkcji strerror. W kolejnej linijce drukujemy numer błędu (zmienna errno), oraz opis błędu (zmienna opis). Aby pokazać jak to działa w praktyce rozważmy następujący przykład.

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

int main (int argc, char *argv[])
{
	char *progname = argv[0];
	char *filename = argv[1];
	FILE *wp;

	if (argc != 2)
	{
		fprintf(stderr, "Uzycie: %s filename\n", progname);
		return -1;
	}
	if ((wp = fopen(filename, "r")) != NULL)
	{
		fprintf(stderr, "Plik otwarty pomyslnie\n");
		fclose(wp);
	}
	else
		fprintf(stderr, "No:%d - %s\n",errno,(char *)strerror(errno));
	return 0;
}
```

Jeśli wskaźnik wp będzie posiadał wartość NULL – co jest równoważne (w tym kontekście) z sytuacją, w której nie uda się otworzyć pliku – to wykona się instrukcja else, po której na standardowe wyjście błędów zostanie wyświetlony numer oraz opis błedu.

10.5 iso646.h

Nagłówek iso646.h zawiera makra odnoszące się do operatorów relacji opisanych w tabeli 2.3.2 oraz operatorów bitowych z tabeli 2.3.5. Wprowadzono je po to, ażeby ułatwić wprowadzanie tych operatorów ludziom używającym klawiatury nie-QWERTY (układ liter). Nagłówek ten opisuje 11 makr, które zdefiniowane są następująco. Przykład 2.3.10 z użyciem makr z nagłówka iso646.h znajduje się na listingu.

<table>
<tr><th>Makro</th><th>Operator</th></tr>
<tr><td>and</td><td>&&</td></tr>
<tr><td>and_eq</td><td>&=</td></tr>
<tr><td>bitand</td><td>&</td></tr>
<tr><td>bitor</td><td>|</td></tr>
<tr><td>compl</td><td>~</td></tr>
<tr><td>not</td><td>!</td></tr>
<tr><td>not_eq</td><td>!=</td></tr>
<tr><td>or</td><td>||</td></tr>
<tr><td>or_eq</td><td>|=</td></tr>
<tr><td>xor</td><td>^</td></tr>
<tr><td>xor_eq</td><td>^=</td></tr>
</table>

Tabela 10.5.1 Lista makr z nagłówka iso646.h

```
#include <stdio.h>
#include <iso646.h>

unsigned setbits (unsigned x, int p, int n, unsigned y);

int main (void)
{
      unsigned x = 1023, y = 774;
      int p = 7, n = 5;
      printf("%u\n", setbits(x, p, n, y));   // 823
}

unsigned setbits (unsigned x, int p, int n, unsigned y)
{
      y = compl(y);                          // Krok 2
      y and_eq compl(compl(0) << n);         // Krok 3
      y <<= p + 1 - n;                       // Krok 4
      y = compl(y);                          // Krok 5
      return x bitand y;                     // Krok 6
}
```

10.6 limits.h

Nagłówek limits.h zawiera makra, które zwracają pewne wartości. Wartościami tymi są zakresy typów danych. W gruncie rzeczy zakres typu zależy od implementacji, lecz z pomocą tych makr można sprawdzić jaki zakres posiada konkretny typ danych na naszym sprzęcie. Makra te zostały zestawione

w tabeli 10.6.1. Przykład użycia znajduje się poniżej.

Nazwa	Opis
CHAR_BIT	Liczba bitów w typie char
SCHAR_MIN	Minimalna wartość dla typu char ze znakiem (signed)
SCHAR_MAX	Maksymalna wartość dla typu char ze znakiem (signed)
UCHAR_MAX	Maksymalna wartość dla typu char bez znaku (unsigned)
CHAR_MIN	Minimalna wartość dla typu char
CHAR_MAX	Maksymalna wartość dla typu char
SHRT_MIN	Minimalna wartość dla typu short int
SHRT_MAX	Maksymalna wartość dla typu short int
USHRT_MAX	Maksymalna wartość dla typu short int bez znaku (unsigned)
INT_MIN	Minimalna wartość dla typu int
INT_MAX	Maksymalna wartość dla typu int
UINT_MAX	Maksymalna wartość dla typu int bez znaku (unsigned)
LONG_MIN	Minimalna wartość dla typu long int
LONG_MAX	Maksymalna wartość dla typu long int
ULONG_MAX	Maksymalna wartość dla typu long int bez znaku (unsigned)
LLONG_MIN	Minimalna wartość dla typu long long int
LLONG_MAX	Maksymalna wartość dla typu long long int
ULLONG_MAX	Maksymalna wartość dla typu long long int bez znaku (unsigned)

Tabela 10.6.1 Spis makr z nagłówka limits.h

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

int main (void)
{
	printf("CHAR_BIT: %d\n", CHAR_BIT);
	printf("SCHAR_MIN: %d\n", SCHAR_MIN);
	printf("SCHAR_MAX: %d\n", SCHAR_MAX);
	printf("UCHAR_MAX: %d\n", UCHAR_MAX);
	printf("CHAR_MIN: %d\n", CHAR_MIN);
	printf("CHAR_MAX: %d\n", CHAR_MAX);
	printf("SHRT_MIN: %d\n", SHRT_MIN);
	printf("SHRT_MAX: %d\n", SHRT_MAX);
	printf("USHRT_MAX: %d\n", USHRT_MAX);
	printf("INT_MIN: %d\n", INT_MIN);
	printf("INT_MAX: %d\n", INT_MAX);
```

```
    printf("UINT_MAX: %u\n", UINT_MAX);
    printf("LONG_MIN: %ld\n", LONG_MIN);
    printf("LONG_MAX: %ld\n", LONG_MAX);
    printf("ULONG_MAX: %lu\n", ULONG_MAX);
    printf("LLONG_MIN: %lld\n", LLONG_MIN);
    printf("LLONG_MAX: %lld\n", LLONG_MAX);
    printf("ULLONG_MAX: %llu\n", ULLONG_MAX);

    return 0;
}
```

10.7 locale.h

Nagłówek locale.h zawiera dwie funkcje (setlocale, localeconv) i jeden typ danych (lconv) używanych do ustawień regionalnych. W zależności od kompilatora można wybrać różne ustawienia lokalne, ale przynajmniej dwa ustawienia są standardowe i dostępne we wszystkich implementacjach. Wybór ustawienia lokalnego następuje za pomocą funkcji setlocale, dlatego też o standardowych ustawieniach zostanie powiedziane podczas opisu tejże funkcji.

Zasób	Opis
setlocale	Funkcja ustawiająca lub pobierająca informacje o ustawieniach lokalnych
localeconv	Funkcja zwracająca strukturę typu lconv
lconv	Struktura danych zawierająca różne informacje o lokalnych ustawieniach

Tabela 10.7.1 Funkcje i typ danych z nagłówka locale.h

```
char *setlocale (int category, const char *locale)
```

Opis

Funkcja setlocale używana jest do ustawiania ustawień lokalnych, lub zmiany ich w całości lub częściowo w aktualnie wykonującym się programie. Jeśli jako locale podamy **NULL** to otrzymamy nazwę aktualnego ustawienia lokalnego. Na starcie wszystkie programy mają ustawione "C" jako locale. Aby użyć ustawień domyślnych dla danego środowiska musimy podać jako drugi argument "".

Parametry

Pierwszym parametrem jest informacja o tym co będzie zmieniane (część ustawień, lub całość), drugim natomiast na jaką wartość. Drugi parametr może mieć różne wartości, nie mniej jednak te dwie wymienione w tabeli powinny być dostępne we wszystkich implementacjach.

category	
Nazwa	**Zmiana**
LC_ALL	Całe ustawienia lokalne
LC_COLLATE	Wpływa na zachowanie funkcji strcoll i strxfrm
LC_CTYPE	Wpływa na zachowanie funkcji z ctype.h (z wyjątkiem isdigit, isxdigit)
LC_MONETARY	Wpływa na formatowanie wartości monetarnych
LC_NUMERIC	Wpływa na kropkę dziesiętną w formatowanym wejściu / wyjściu
LC_TIME	Wpływa na zachowanie funkcji strftime
locale	
Lokalna nazwa	**Opis**
"C"	Minimalne ustawienia lokalne "C"
""	Domyślne ustawienia środowiskowe

Zwracana wartość

Funkcja setlocale w przypadku powodzenia zwraca wskaźnik do tekstu (nazwa ustawień) identyfikujący aktualne ustawienia lokalne dla kategorii (`category`). W przypadku nie powodzenia funkcja zwraca **NULL**.

```
struct lconv *localeconv (void)
```

Opis

Funkcja localeconv jest używana do pobierania informacji o ustawieniach lokalnych. Informacje te zwraca w formie wskaźnika do struktury typu lconv. Trzeba mieć na uwadzę, że kolejne wywołania funkcji nadpisują zmienione wartości.

Parametry

Brak

Zwracana wartość

Funkcja zwraca wskaźnik do typu strukturowego lconv.

Struktura lconv przetrzymuje informacje o tym w jaki sposób mają być zapisane wartości monetarne i nie monetarne. Struktura ta posiada następujące pola.

Pole	Opis
char *decimal_point	Kropka dziesiętna używana w liczbach nie monetarnych
char *thousands_sep	Separator używany do ograniczenia grup cyfr w liczbach nie monetarnych
char *grouping	Ustawienie ilości cyfr, w które grupowane są liczby nie monetarne: "\3" grupuje tak: 1,000,000 "\1\2\3"grupuje tak: 1,000,00,0 "\3\1" grupuje tak: 1,0,0,0,000
char *int_curr_symbol	Międzynarodowy znak waluty, trzyliterowy, np. USD
char *currency_symbol	Lokalny znak waluty, np. $
char *mon_decimal_point	Kropka dziesiętna używana w liczbach monetarnych
char *mon_thousands_sep	Separator używany do ograniczenia grup cyfr w liczbach monetarnych
char *mon_grouping	Określa ilość cyfr każdej grupy, która ma być oddzielona w polu mon_thousands_sep
char *positive_sign	Znak używany dla dodatnich liczb monetarnych, oraz zera
char *negative_sign	Znak używany dla ujemnych liczb monetarnych
char int_frac_digits	Liczba miejsc po przecinku dla monetarnych wartości w międzynarodowym formacie
char frac_digits	Liczba miejsc po przecinku dla monetarnych wartości w lokalnym formacie
char p_cs_precedes	Jeśli wartość tego pola równa się 1, to znak waluty poprzedza liczby dodatnie (lub zero). Jeśli wartość to 0, to znak waluty jest po wartości liczbowej
char n_cs_precedes	Jeśli wartość tego pola równa się 1, to znak waluty poprzedza liczby ujemne. Jeśli wartość to 0, to znak waluty jest po wartości liczbowej
char p_sep_by_space	Jeśli wartość tego pola równa się 1, to spacja występuje pomiędzy znakiem waluty, a wartością liczbową (dodatnia, lub zero). Jeśli 0 to spacja nie występuje.
char n_sep_by_space	Jeśli wartość tego pola równa się 1, to spacja występuje pomiędzy znakiem waluty, a wartością liczbową (ujemna). Jeśli 0 to spacja nie występuje.
char p_sign_posn	Pozycja znaku dla dodatnich (lub zera) wartości monetarnych: 0 – Znak waluty i wartość liczbowa otoczone nawiasami 1 – Znak przed znakiem waluty i wartością 2 – Znak po znakiem waluty i wartością 3 – Znak bezpośrednio przed znakiem waluty 4 – Znak bezpośrednio po znaku waluty
char n_sign_posn	Pozycja znaku dla ujemnych wartości monetarnych (zobacz powyżej)

Przykład

```
#include <stdio.h>
#include <locale.h>
int main (void)
{
	struct lconv *lv;
	printf("Nazwa ustawien:  %s\n", setlocale(LC_ALL, NULL));
	lv = localeconv();
	printf("int_curr_symbol: %s\n", lv->int_curr_symbol);

	printf("Nazwa ustawien:  %s\n", setlocale(LC_ALL, ""));
	lv = localeconv();
	printf("int_curr_symbol: %s\n", lv->int_curr_symbol);

	return 0;
}
```

10.8 math.h

W punkcie 2.3.6 przedstawione zostały funkcje matematyczne, lecz nie wszystko z nagłówka math.h. Rzeczą o której tam nie wspomniano to wykaz stałych matematycznych, dlatego też zostały one umieszczone w poniższej tabeli.

Nazwa stałej	Opis
M_E	Podstawa logarytmu naturalnego – liczba e
M_LOG2E	Logarytm przy podstawie 2 z e
M_LOG10E	Logarytm przy podstawie 10 z e
M_LN2	Logarytm naturalny z 2
M_LN10	Logarytm naturalny z 10
M_PI	Liczba PI
M_PI_2	Liczba PI podzielona przez 2
M_PI_4	Liczba PI podzielona przez 4
M_1_PI	Odwrotność PI, czyli 1/PI
M_2_PI	Odwrotność PI pomnożona przez 2, czyli 2/PI
M_2_SQRTPI	Odwrotność liczby PI pod pierwiastkiem pomnożona przez 2, czyli 2/sqrt(PI)
M_SQRT2	Pierwiastek kwadratowy z 2
M_SQRT1_2	Pierwiastek kwadratowy z ½

Tabela 10.8.1 Stałe matematyczne

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

int main (void)
{
     printf("%f\n", M_E);
     printf("%f\n", M_LOG2E);
     printf("%f\n", M_LOG10E);
     printf("%f\n", M_LN2);
     printf("%f\n", M_LN10);
     printf("%f\n", M_PI);
     printf("%f\n", M_PI_2);
     printf("%f\n", M_PI_4);
     printf("%f\n", M_1_PI);
     printf("%f\n", M_2_PI);
     printf("%f\n", M_2_SQRTPI);
     printf("%f\n", M_SQRT2);
     printf("%f\n", M_SQRT1_2);
     return 0;
}
```

10.9 setjmp.h

Nagłówek setjmp.h zawiera informacje umożliwiające omijanie normalnych wywołań funkcji, tzn nie lokalne skoki (non-local goto). Nagłówek zawiera definicję funkcji longjmp, setjmp, oraz typu jmp_buf. Prototyp funkcji longjmp wygląda następująco.

```
void longjmp (jmp_buf env, int val)
```

Opis

Funkcji używa się do „cofnięcia" do ostatniego wywołania funkcji setjmp. Wszystkie potrzebne informacje znajdują się w zmiennej env. Funkcja ta odwołuje się do ostatniego wywołania setjmp. Wywołana funkcja „nakazuje" zwrócić funkcji setjmp wartość val.

Parametry

Pierwszym parametrem jest zmienna przechowująca informacje potrzebne do odtworzenia środowiska, które zostało zapisane w miejscu wywołania funkcji setjmp. Drugim parametrem jest liczba całkowita, która pomaga w sterowaniu skokami.

Zwracana wartość

Brak.

```
int setjmp (jmp_buf env)
```

Opis

Funkcja pobiera jako argument zmienną typu jmp_buf, w której przechowywane są informacje odnośnie środowiska, które w późniejszym czasie mogą zostać użyte w wywołaniu funkcji longjmp. Funkcja tworzy punkt bazowy, do którego można się „cofnąć”.

Parametry

Zmienna typu jmp_buf.

Zwracana wartość

Za pierwszym razem funkcja zwraca zawsze zero. Jeśli wywołamy funkcję longjmp ze zmienną env to setjmp zwraca wartość podaną jako drugi argument funkcji longjmp.

Typ jmp_buf jest to typ danych zdolny przetrzymywać informacje o środowisku, które w późniejszym czasie może być wykorzystane przez funkcje z nagłówka setjmp.h.

Przykład

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

int main (void)
{
	jmp_buf env;
	int kontrola;

	kontrola = setjmp(env);

	if (!kontrola)
		longjmp (env, 1);
	else
		printf("longjmp sie wykonalo\n");
	return 0;
}
```

Omówienie programu

Tworzymy zmienną env typu jmp_buf do przechowywania informacji o stanie środowiska. Następnie do zmiennej kontrola przypisujemy wywołanie setjmp, które w tym miejscu zwróci wartość 0, bo jest to pierwsze wywołanie tej funkcji. Warunek pętli jest spełniony, tak więc wykonuje się funkcja longjmp, która „oddaje” sterowanie do miejsca wywołania setjmp, oraz przekazuje wartość, która ma zostać zwrócona za pomocą setjmp, czyli 1. Dlatego też przy kolejnym wywołaniu instrukcji if,

warunek nie jest spełniony i zostaje wydrukowana informacja, która znajduje się w części else.

10.10 signal.h

Nagłówek signal.h definiuje dwie funkcje, które odpowiedzialne są za współpracę z sygnałami. Sygnał jest to między procesowa forma komunikacji używana w Linuksie (Unix). Takim sygnałem, może być na przykład ten, który jest używany do zakończenia działania programu, jeśli program się zapętlił, a nie użyliśmy żadnej instrukcji break.

```
int raise (signal sig);
```

Opis

Funkcja wysyła sygnał podany jako argument do aktualnie wykonywanego procesu.

Parametry

Parametrem jest sygnał, który ma zostać przekazany do programu. Możliwe sygnały zostały wypisane w tabeli (zdefiniowane jako makra).

sig	Pełna nazwa	Opis
SIGABRT	Signal Abort	Nieprawidłowe zakończenie programu (analogia do wywołania funkcji abort)
SIGFPE	Signal Floating-Point Exception	Błędna operacja matematyczna, np. dzielenie przez zero, albo operacja skotkująca przepełnieniem (niekoniecznie dla typów float).
SIGILL	Signal Illegal Instructions	Niepoprawne dane w funkcji, takie jak niedozwolone instrukcje
SIGINT	Signal Interrupt	Interaktywny sygnał ostrzeżenia, zazwyczaj generowany przez użytkownika
SIGSEGV	Signal Segmentation Violation	Niepoprawny dostęp do zasobu, np. wtedy, gdy program stara się odczytać, lub zapisać coś do pamięci, która nie może być użyta w tym celu.
SIGTERM	Signal Terminate	Żadanie zakończenia programu

Zwracana wartość

Zero jest zwracane w przypadku sukcesu. W przeciwnym wypadku wartość nie zerowa.

Przykład

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

void dowidzenia (void);
int main (void)
{
	int i;
	printf(": ");
	scanf("%d", &i);
	if (i)
		raise(SIGABRT);

	atexit(dowidzenia);
	return 0;
}

void dowidzenia (void)
{
	printf("Dowidzenia\n");
}
```

Omówienie programu

Jeśli podamy wartość różną od zera, to program wykona funkcję raise z argumentem kończącym działanie programu w sposób nieprawidłowy, przez co funkcja dowidzenia się nie wykona.

```
void (*signal(int sig, void (*func)(int)))(int)
```

Opis

Funkcja ustawia akcję, która ma zostać wykonana w momencie, gdy program otrzyma sygnał sig. Jeśli wartością func będzie **SIG_DFL** to nastąpi domyślna obsługa sygnału, jeśli **SIG_IGN** to sygnał zostanie zignorowany. Jeśli żadna z dwóch wymienionych wartości nie zostanie przekazana to parametrem tym jest wskaźnik do funkcji, która wykona się w przypadku wystąpienia sygnału. Jeśli nie użyjemy w ogóle funkcji signal, to sygnały wkonują się tak jakby został przekazany argument **SIG_DFL**.

Parametry

Jako sig rozumiemy jeden z sygnałów wymienionych w tabeli (funkcja raise). Jako func można podać **SIG_DFL**, **SIG_IGN**, lub wskaźnik do własnoręcznie napisanej funkcji. Jeśli func ma się odnosić do naszej funkcji, to jej prototyp powinien wyglądać następująco:

```
void funkcja (int parametr)
```

Zwracana wartość

W przypadku sukcesu funkcja signal zwraca wartość func (czyli sygnał będzie traktowany domyślnie, ignorowany, lub funkcja będzie go obsługiwać). W przypadku nie powodzenia funkcja zwraca **SIG_ERR** (makro zdefiniowane w nagłówku signal.h) oraz errno zostaje ustawione na odpowiednią wartość.

Przykład

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

void sgtm (int x);
int main (void)
{
	if (signal(SIGABRT, SIG_IGN) == SIG_ERR)
		fprintf(stderr, "%s\n", (char *)strerror(errno));
	raise(SIGABRT);

	printf("Funkcja jednak dalej dziala\n");

	signal(SIGTERM, sgtm);
	raise(SIGTERM);

	raise(SIGABRT);

	signal(SIGABRT, SIG_DFL);
	raise(SIGABRT);

	return 0;
}
void sgtm (int x)
{
	printf("Pojawil sie sygnal SIGTERM\n");
	return;
}
```

Omówienie programu

Sprawdzamy warunek, w którym porównujemy wywołanie funkcji signal z wartością **SIG_ERR**, jeśli są równe to wydrukowane zostaną informację o błędzie. Jeśli natomiast zwrócona wartość jest inna, to sygnał **SIGABRT** przy każdym następnym wywołaniu będzie ignorowany. Kolejne wywołanie signal, mówi o tym, że jeśli wystąpi sygnał **SIGTERM** to zostanie uruchomiona funkcja sgtm, co w kolejnej linii zostało pokazane. Kolejne wywołanie sygnału **SIGABRT** pokazuje, że dopóki nie zmienimy akcji

to będzie się wykonywać ostatnia ustawiona akcja, dlatego też sygnał jest po raz kolejny ignorowany. W kolejnej linii zmieniamy akcję na domyślną, dlatego też ostatnie wywołanie raise przerwie działanie programu.

10.11 stdarg.h

W nagłówku stdarg.h znajdują się makra, dzięki którym można używać zmiennej listy parametrów, tzn wywoływać funkcję z różną ilością argumentów. Mechanizm ten został szczegółowo opisany w punkcie 8.4. Nagłówek definiuje jeden typ danych va_list i trzy funkcję (makra) va_start, va_arg, va_end. Ponieważ opis użycia wraz z przykładem znajduje się we wspomnianym punkcie 8.4 w tym miejscu przedstawie tylko prototypy dla danych funkcji.

```
void va_start (va_list ap, ilosc_parametrow)
```

```
type va_arg (va_list ap, type)
```

Gdzie type oznacza konkretny typ danych, jeśli argumenty są typu int, to zamieniamy oba wystąpienia słowa type na int.

```
void va_end (va_list ap)
```

10.12 stdbool.h

W nagłówku stdbool.h znajdują się cztery makra, za pośrednictwem których możemy w sposób bardziej oczywisty (jeśli wartości 1 i 0 nie były oczywiste przy warunkach) sterować programem. Makrami tymi są:

- bool – typ danych
- true – odpowiada wartości 1
- false – odpowiada wartości 0
- __bool_true_false_are_defined – odpowiada wartości 1

Jeśli pojawiły się jakiekolwiek wątpliwości odnośnie sposobu użycia, to przykład poniższy powinien je zdecydowanie rozwiać.

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

int main (void)
{
	bool condition = true;

	if (condition)
	{
		printf("To jest prawda\n");
		condition = false;
	}
	if (!condition)
		printf("To jest falsz\n");

	return 0;
}
```

10.13 stdio.h

W nagłówku stdio.h znajdują się funkcje dotyczące wejścia i wyjścia. W rozdziale 8 została opisana część funkcji należących do tego nagłówka. W tym miejscu zostaną wymienione wszystkie funkcje, lecz opisane zostaną te, których w rozdziale 8 nie omówiono. Ponieważ funkcje te używa się zazwyczaj równolegle z innymi, tak więc przykład użycia może (lecz nie musi) ograniczyć się do jednego na grupę.

Nazwa funkcji	Opis
Operacje na plikach	
fopen	Otwarcie pliku
freopen	Otwarcie pliku ze związanym strumieniem
fflush	Czyszczenie zbuforowanych danych
fclose	Zamykanie pliku
remove	Usunięcie pliku
rename	Zmiana nazwy pliku
tmpfile	Tworzenie tymczasowego pliku
tmpnam	Tworzenie unikalnej nazwy pliku
setvbuf	Kontrolowanie buforowania danych
setbuf	Kontrolowanie buforowania danych
Formatowane wyjście	
fprintf	Wypisywanie danych do strumienia

printf	Wypisywanie danych na stdout
sprintf	Wypisywanie danych do tablicy znaków
vprintf	Odpowiedniki dla funkcji printf z tą różnicą, że podajemy arg, który został zainicjowany przez va_start
vfprintf	
vsprintf	
Formatowane wejście	
fscanf	Czytanie danych ze strumienia
scanf	Czytanie danych ze stdin
sscanf	Czytanie danych z tablicy znaków
Wejście i wyjście znakowe	
fgetc	Czytanie znaku ze strumienia
fgets	Czytanie większej ilości znaków ze strumienia
fputc	Wypisanie znaku do strumienia
fputs	Wypisanie większej ilosci znakow do strumienia
getc	Czytanie znaku ze stdin
getchar	Czytanie znaku ze stdin
gets	Wczytanie wiersza ze stdin i zapisanie go w tablicy
putc	Wypisanie znaku do strumienia
putchar	Wypisanie znaku do stdout
puts	Wypisanie tekstu z tablicy do stdout
Pozycja w pliku	
fseek	Wyznaczenie pozycji w strumieniu
ftell	Wartość bieżącej pozycji dla strumienia
rewind	Przewinięcie pozycji na początek pliku
fgetpos	Zapamiętanie pozycji pliku
fsetpos	Ustawienie pozycji w pliku z użyciem wartości otrzymanej przez funkcje fgetpos
Obsługa błędów	
clearerr	Kasowanie znacznika błędu i znacznika końca pliku dla strumienia
feof	Sprawdzanie znacznika końca pliku
ferror	Sprawdzanie znacznika błędu dla strumienia
perror	Wyświetlanie komunikatu o błędzie

10.13.1 Operacje na plikach

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

Opis

Funkcja szczegółowo opisana w punkcie 8.5

```
FILE *freopen (const char *filename, const char *mode, FILE *stream)
```

Opis

Funkcja freopen początkowo próbuje zamknąć (jeśli występują) pliki związane ze strumieniem stream. Następnie, jeśli pliki zostały zamknięte lub nie było takich plików, funkcja otwiera plik w konkretnym trybie (mode) wskazywany przez filename i powiązuje go ze strumieniem stream w taki sam sposób jak funkcja fopen.

Parametry

Pierwszym parametrem jest nazwa pliku, drugim tryb otwarcie pliku (zobacz funkcję fopen), trzecim jest strumień (FILE *, lub jeden z podstawowych – stdin, stdout).

Zwracana wartość

W przypadku sukcesu funkcja zwraca wskaźnik do pliku, jeśli operacja się nie powiodła to NULL jest zwracane.

Przykład

```
#include <stdio.h>

int main (void)
{
      FILE *wp;
      char *napis = "Napis do pliku 1";
      char *info = "Napis do pliku 2";

      wp = fopen ("my_first_file.txt", "w");
      fprintf(wp, "%s\n", napis);
      freopen ("my_second_file.txt", "w", wp);
      fprintf(wp, "%s\n", info);

      fclose(wp);

      return 0;
}
```

Omówienie programu

Najpierw do wskaźnika wp przypisujemy otwarcie pliku **my_first_file.txt** w trybie zapisu, drukujemy do tego pliku znaki wskazywane przez napis. Następnie za pomocą freopen otwieramy drugi plik, przypisując go do tego samego wskaźnika. Funkcja freopen zamyka połączenie z pierwszym plikiem przed otwarciem drugiego.

```
int fflush (FILE *stream)
```

Opis

Funkcję można kolokwialnie nazwać funkcją „odświeżającą zawartość" pliku. Chodzi o to, że jeśli otworzyliśmy jakiś plik do zapisu, to wszystkie wprowadzone dane do pliku nie zostaną zapisane w nim dopóki plik nie zostanie zamknięty poprawnie, lub funkcja fflush nie zostanie wywołana. Jeśli argumentem będzie NULL, to wszystkie otwarte pliki są „odświeżane" i pomimo iż plik nie został zamknięty zawartość zostaje wydrukowana do pliku.

Parametry

Parametrem jest wskaźnik do pliku, lub NULL.

Zwracana wartość

Zero jest zwracane w przypadku sukcesu. Jeśli wystąpił jakiś problem, to **EOF** jest zwracane.

Przykład

Przykład użycia tej funkcji został przedstawiony w kilku następnych funkcjach. Nie mniej jednak w tym miejscu pokażę co się stanie, jeśli argumentem będzie NULL.

```
#include <stdio.h>

int main (void)
{
      FILE *wp1, *wp2, *wp3;

      wp1 = fopen("plik1.txt", "w+");
      wp2 = fopen("plik2.txt", "w+");
      wp3 = fopen("plik3.txt", "w+");

      fprintf(wp1, "Ciag znakow do plik1.txt\n");
      fprintf(wp2, "Ciag znakow do plik2.txt\n");
      fprintf(wp3, "Ciag znakow do plik3.txt\n");

      getchar();
      fflush(NULL);
```

```
	getchar();

	fclose(wp1);
	fclose(wp2);
	fclose(wp3);
	return 0;
}
```

Omówienie programu

Otwieramy trzy pliki do zapisu i odczytu, następnie drukujemy trzy ciągi znaków do trzech różnych plików. W momencie gdy program prosi nas o wpisanie znaku otwieramy nowy terminal i sprawdzamy zawartość tych trzech plików za pomocą polecenia:

```
$ cat *.txt
```

W plikach nie ma nic, ponieważ użyte zostało buforowanie (opis tego znajduje się w funkcjach setbuf i setvbuf – po przeczytaniu tych funkcji zdecydowanie łatwiej będzie Ci zrozumieć ten mechanizm użyty w tym listingu). W momencie, gdy czyścimy bufor, a właściwie wszystkie bufory (bo są otwarte trzy pliki) zawartość zostaje przeniesiona do pliku. W tym momencie program prosi o podanie kolejnego znaku, jeśli przed naciśnięciem entera wyświetlisz zawartość plików jeszcze raz to ujrzysz już zawartość, która została przekazana do nich.

```
int fclose (FILE *stream)
```

Opis

Funkcja opisana w punkcie 8.5

```
int remove (const char *filename)
```

Opis

Funkcja usuwa plik o nazwie filename. Jeśli plik znajduje się w innym katalogu niż bieżący, to możemy podać ścieżkę do tego pliku.

Parametry

Jako parametr podajemy nazwę pliku, jeśli plik znajduje się w katalogu bieżącym, bądź ścieżkę do pliku, jeśli plik znajduje się w innym miejscu.

Zwracana wartość

Funkcja zwraca zero, jeśli plik udało się usunąć. W przeciwnym wypadku funkcja zwraca wartość nie zerową oraz ustawia zmienną errno na odpowiednią wartość.

Przykład

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

int main (int argc, char *argv[])
{
	if (argc != 2)
	{
		fprintf(stderr, "Uzycie: %s nazwa_pliku\n", argv[0]);
		return -1;
	}

	if (!remove(argv[1]))
		fprintf(stderr, "Usuniecie pliku: %s powiodlo sie\n", argv[1]);
	else
		fprintf(stderr, "%s : %s\n", argv[1], (char *)strerror(errno));
	return 0;
}
```

```
int rename (const char *oldname, const char *newname)
```

Opis

Funkcja rename zmienia nazwę pliku lub katalogu podanego jako parametr oldname na nową nazwę sprecyzowaną jako newname.

Parametry

Funkcja przyjmuje dwa argumenty. Pierwszym z nich jest aktualna nazwa pliku lub katalogu, drugim natomiast jest nowa nazwa.

Zwracana wartość

Funkcja zwraca zero, jeśli nazwa pliku została pomyślnie zmieniona, bądź wartość nie zerową jeśli operacja ta się nie powiodła. Jeśli operacja się nie powiedzie zmienna errno zostaje ustawiona na odpowiednią wartość.

Przykład

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

int main (int argc, char *argv[])
```

```
{
	if (argc != 3)
	{
		fprintf(stderr, "Uzycie: %s oldName newName\n", argv[0]);
		return -1;
	}

	if (!rename(argv[1], argv[2]))
		fprintf(stderr, "Plik o nazwie: %s teraz nazywa sie: %s\n",
argv[1], argv[2]);
	else
	{
		fprintf(stderr, "%s : %s\n", argv[1], (char *)strerror(errno));
		return -1;
	}
	return 0;
}
```

`FILE *tmpfile (void)`

Opis

Funkcja tworzy tymczasowy plik binarny, który otwarty jest w trybie **wb+** oraz gwarantuje, że nazwa tego pliku będzie inna niż jakiegokolwiek istniejącego pliku. Tymczasowy plik jest automatycznie usuwany w momencie zamknięcie strumienia (pliku), lub w przypadku, gdy program kończy swoje działanie normalnie.

Parametry

Brak.

Zwracana wartość

Funkcja zwraca wskaźnik do pliku w przypadku sukcesu i NULL jeśli pliku nie udało się utworzyć.

Przykład

```
#include <stdio.h>

int main (void)
{
	FILE *wp;
	int c;

	wp = tmpfile();

	fprintf(wp, "Ala ma kota\n");
	rewind(wp);

	while ((c = fgetc(wp)) != EOF)
```

```
            putchar(c);
      return 0;
}
```

Omówienie programu

Za pomocą tmpfile tworzymy tymczasowy plik, do którego drukujemy pewien ciąg znaków. Następnie za pomocą rewind przesuwamy kursor na początek pliku oraz drukujemy wszystkie znaki, aż do **EOF**. Po zamknięciu programu normalnie, plik jest zamykany i usuwany.

```
char *tmpnam (char s[L_tmpnam])
```

Opis

Funkcja tworzy losową nazwę różną od jakiegokolwiek istniejącego pliku. Ten ciąg znaków można wykorzystać do utworzenia pliku tymczasowego z pewnością, że nie nadpiszemy żadnego innego. Jeśli utworzymy plik za pomocą fopen z nazwą otrzymaną za pomocą tmpnam, to należy pamiętać, że po zakończeniu programu plik nie jest kasowany, można go usunąć za pomocą remove.

Parametry

Parametrem jest tablica znaków składająca się z **L_tmpnam** elementów. Wartość ta jest makrem, którego definicja znajduje się w nagłówku stdio.h. Jeśli parametrem jest **NULL** to nazwa jest zwracana.

Zwracana wartość

Jeśli parametrem jest tablica s, to ona jest zwracana. Jeśli parametrem jest **NULL**, to przypisanie funkcji powinno być do wewnętrznego wskaźnika (pokazane na przykładzie). Jeśli funkcji nie uda się utworzyć nazwy pliku to **NULL** jest zwracane.

Przykład

```
#include <stdio.h>

int main (void)
{
      FILE *wp1, *wp2;
      char bufor[L_tmpnam];
      char *buf;

      tmpnam(bufor);
      printf("Wygenerowana nazwa to: %s\n", bufor);

      buf = tmpnam(NULL);
```

```
    printf("Wygenerowana nazwa to: %s\n", buf);

    wp1 = fopen(bufor, "w");
    wp2 = fopen(buf, "w");

    fprintf(wp1, "Pewien ciag znakow 1\n");
    fprintf(wp2, "Pewien ciag znakow 2\n");

    fflush(wp1);
    fflush(wp2);

    getchar();

    remove(bufor);
    remove(buf);

    return 0;
}
```

Omówienie programu

W programie tym na początku definiujemy tablicę znaków o ilości elementów równej **L_tmpnam**, oraz wskaźnik na tym char. Następnie wywołujemy funkcję tmpnam z argumentem bufor. Wygenerowana nazwa zostanie zapisana w tej tablicy. Jeśli przypiszmy wywołanie z argumentem **NULL** do wskaźnika, to wskaźnik będzie wskazywał na wygenerowaną nazwę. Za pomocą funkcji fopen otwieramy (a właściwie tworzymy) pliki o podanych nazwach i drukujemy do nich ciągi znaków. Funkcja getchar została wpisana, ażeby można było zobaczyć, że faktycznie taki plik został utworzony i pewien ciąg znaków został do niego zapisany. Dlatego też otwórz kolejny terminal i wpisz poniższe polecenie:

```
$ ls /tmp
```

W katalogu tym powinny znajdować się jakieś pliki, mniejsza o to, nas interesują dwa pliki, które utworzyliśmy przed chwilą. U mnie nazywają się one, bynajmniej zaczynają się od liter **file** po których występują losowe znaki. Aby wyświetlić zawartość tych plików wpisz:

```
$ cat /tmp/file*
```

Najlepszym przykładem użycia funkcji fflush jest to właśnie miejsce. Gdybyśmy nie wywołali tych funkcji to nie zobaczylibyśmy zawartości pliku, tzn zobaczylibyśmy, lecz nie byłoby w nim tekstu, który chcieliśmy tam umieścić. Można by sobie to wyobrazić, że funkcja fflush „odświeża" zawartość

pliku. Po naciśnięciu klawisza **enter**, program zakończy swoje działanie, a pliki zostaną usunięte.

```
void setbuf (FILE *stream, char *bufor)
```

Opis

Informacje o buforze były w jakimś tam stopniu poruszone podczas omawiania pętli while. W tym miejscu powiem o tym szerzej, oraz pokażę zastosowanie oraz właściwości buforu. Funkcja setbuf włącza lub wyłącza pełne buforowanie dla danego strumienia. Funkcję wywołuje się raz, po otworzeniu pliku, a przed wykonaniem jakiejkolwiek operacji wejścia, lub wyjścia. Tablica, która będzie buforem powinna składać się z co najmniej **BUFSIZ** (stała zdefiniowane w nagłówku stdio.h) elementów. Buforowany strumień ma to do siebie, że informacje zostają zapisane w pliku dopiero po jego zamknięciu, lub po użyciu funkcji czyszczącej bufor (fflush). Po zakończeniu programu bufor jest automatycznie czyszczony. Jeśli jako bufor podamy **NULL** to wyłączamy buforowania dla strumienia stream. W tym przypadku drukowanie do pliku odbywa się natychmiastowo. Wszystkie otwarte pliki standardowo są buforowane. Za pomocą tej funkcji można zdefiniować swój własny bufor.

Parametry

Pierwszym parametrem jest wskaźnik do pliku, drugim tablica znaków o ilości elementów nie mniejszej niż **BUFSIZE**.

Zwracana wartość

Brak.

Przykład 1

```
#include <stdio.h>
int main (void)
{
      FILE *wp;
      char buff[BUFSIZ];
      wp = fopen ("moj_plik.txt", "w+");
      setbuf(wp, buff);
      fprintf(wp, "Pewien ciag znakow 1\n");
      fprintf(wp, "Pewien ciag znakow 2\n");
      fprintf(wp, "Pewien ciag znakow 3\n");
      fprintf(wp, "Pewien ciag znakow 4\n");

      printf("%s", buff);
      //fflush(wp);
      getchar();

      fclose(wp);
```

```
	return 0;
}
```

Omówienie programu 1

Definiujemy bufor buff o rozmiarze **BUFSIZ** otwieramy plik w trybie **w+** i ustawiamy buforowanie dla strumienia wp. Drukujemy do pliku pewne ciągi znaków, ponieważ używamy buforowanej operacji wejścia / wyjścia to dane te najpierw kopiowane są do tablicy buff, co zostało udowodnione podczas drukowania zawartości tablicy. Funkcja getchar została użyta po to, aby sprawdzić czy faktycznie dane nie zostały wydrukowane do pliku. Podczas gdy program oczekuje na podanie znaku możemy sprawdzić w innym terminalu zawartość pliku **moj_plik.txt**. Do czasu, aż plik nie zostanie zamknięty, lub funkcja „odświeżająca zawartość” pliku nie zostanie uruchomiona, w pliku nic się nie pojawi. Usuń komentarze sprzed funkcji fflush, by zobaczyć, że zawartość pokaże się w pliku nawet wtedy, gdy program oczekuje na naszą reakcję.

Przykład 2

```
#include <stdio.h>

int main (void)
{
	FILE *wp;
	wp = fopen ("moj_plik_2.txt", "w+");
	setbuf(wp, NULL);

	fprintf(wp, "Pewien ciag znakow 1\n");
	fprintf(wp, "Pewien ciag znakow 2\n");
	fprintf(wp, "Pewien ciag znakow 3\n");

	getchar();
	fclose(wp);
	return 0;
}
```

Omówienie programu 2

W tym programie wyłączamy buforowanie dla strumienia wp. Ciąg znaków został zapisany w pliku przed jego zamknięciem. Można to sprawdzić uruchamiając nowe okienko terminala, w którym obejrzymy zawartość pliku **moj_plik_2.txt**.

```
int setvbuf (FILE *stream, char *bufor, int mode, size_t size)
```

Opis

Funkcja ta ma wiele wspólnego z omówioną funkcją setbuf, lecz różni się pewnymi szczegółami o których zaraz powiem. Funkcja ta ustawia buforowanie w tryb pełnego, liniowego, oraz wyłączonego buforowania. setvbuf tak samo jak setbuf powinna być wywołana raz, po nawiązaniu połączenia z plikiem, oraz przed użyciem jakichkolwiek operacji wejścia / wyjścia. Rozmiar buforu jest definiowany w ostatnim parametrze jako ilość bajtów. Jeśli jako bufor podamy NULL to system sam dynamicznie zaalokuje pamięć (size bajtów) i będzie korzystał z niej jako buforu dla strumienia stream. Trzeci parametr odpowiada za tryb buforu, tabela z dostępnymi wartościami tego parametru została pokazana poniżej. Pełne buforowanie oraz brak buforowania zostały opisane podczas omawiania funkcji setbuf, natomiast buforowanie liniowe opiszę w tym miejscu. Jak sama nazwa mówi, więc łatwo można się domyśleć, że dane zapisywane są w pliku po wykryciu znaku nowej linii.

Parametry

Pierwszym parametrem jest wskaźnik do pliku (strumień), drugim tablica, która będzie służyła jako bufor o rozmiarze nie mniejszym niż size. Trzecim parametrem jest tryb buforowania (tabelka), czwarty parametr to ilość bajtów dla bufora. Jeśli drugi parametr będzie miał wartość NULL, to system zaalokuje automatycznie pamięć o rozmiarze size bajtów, który będzie służyć jako bufor.

Tryb	Opis
_IOFBF	Pełne buforowanie
_IOLBF	Buforowanie liniowe
_IONBF	Brak buforowania

Zwracana wartość

W przypadku poprawnego połączenia buforu z plikiem zero jest zwracane. W przeciwnym wypadku nie zerowa wartość jest zwracana.

Przykład

```
#include <stdio.h>

int main (void)
{
      int size = 1024;
      FILE *wp;
      char buff[size];
```

```
    wp = fopen("file1.txt", "w+");

    setvbuf(wp, buff, _IOLBF, size);
    fprintf(wp, "Linia bez znaku konca linii. ");
    getchar();
    fprintf(wp, "Druga linia bez znaku konca linii. ");
    getchar();
    fprintf(wp, "Linia ze znakiem konca linii\n");
    getchar();
    fprintf(wp, "Linia nr. 1\n");
    getchar();
    fprintf(wp, "Linia nr. 2\n");
    getchar();
    fprintf(wp, "Linia nr. 3\n");

    fclose(wp);
    return 0;
}
```

Omówienie programu

Najpierw opiszę co się stanie dla takiego trybu buforowania jaki został ustawiony (buforowanie liniowe), a następnie dla pozostałych dwóch. Tak więc dobrze by było, jakbyś zmienił i sprawdził jak się program zachowuje przy innym buforowaniu. Zróbmy pewną czynność, która nam udowodni, że działa to tak, jak zamierzono. Podczas każdego wywołania funkcji getchar sprawdź zawartość pliku **file1.txt**, a następnie naciśnij **enter**. Skoro mamy do czynienia z buforowaniem liniowym, które jak wspomniano wymaga znaku końca linii, by wydrukować ciąg znaków do pliku, domyślamy się, że będziemy musieli wcisnąć dwa razy **enter** zanim nasza zawartość pojawi się w pliku, bowiem dopiero po drugiej funkcji getchar następuje drukowanie do pliku ciągu znaków zakończonych znakiem nowej linii. Jeśli tryb zmienilibyśmy na **_IONBF**, to każdy ciąg znaków, nie zależnie od tego czy zakończony jest znakiem nowej linii, czy nie wydrukowany zostałby do pliku natychmiastowo. Jeśli trybem byłby **_IOFBF** to zawartość pliku uzupełniona by była dopiero po zamknięciu pliku, lub wywołaniu funkcji fflush.

10.13.2 Formatowane wyjście

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

Opis

Funkcja fprintf jest bardzo podobna do funkcji printf, która została szczegółowo opisana w punkcie 8.2. Funkcja różni się tym, że potrafi wysłać tekst również na inny strumien (stream) niż stdout.

Parametry

Funkcja jako pierwszy parametr przyjmuje wskaźnik do strumienia. Jeśli strumieniem jest standardowe wyjście, czyli wywołanie funkcji fprintf(stdout, …); wygląda następująco, to jest ona równoważna z wywołaniem funkcji printf.

Zwracana wartość

W razie powodzenia funkcja zwraca ilość znaków wyświetlonych (zapisanych). W przypadku nie powodzenia wartość ujemna jest zwracana.

Przykład

Jeden na grupe na zakończenie niniejszego podpunktu.

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

Opis

Funkcja szczegółowo opisana w punkcie 8.2

```
int sprintf (char *s, const char *format, ...)
```

Opis

Funkcja opisana szczegółowo w punkcie 8.2

```
int vprintf (const char *format, va_list arg)
```

Opis

Funkcja jest w gruncie rzeczy taką samą funkcją jak printf, lecz z tą różnicą, że zamiast parametrów wypisanych podczas wywołania funkcji przyjmuje listę argumentów arg (Zmienna długość parametrów – opisana w punkcie 8.4). Funkcja ta nie wywołuje automatycznie makra va_end.

Parametry

Funkcja jako pierwszy argument przyjmuje dokładnie to samo co jej odpowiednik bez literki v. Natomiast drugim argumentem jest lista parametrów.

Zwracana wartość

W razie powodzenia funkcja zwraca ilość znaków wyświetlonych (zapisanych). W przypadku nie

powodzenia wartość ujemna jest zwracana.

Przykład

Jeden na grupe na zakończenie niniejszego podpunktu.

```
int vfprintf (FILE *stream, const char *format, va_list arg)
```

Opis

Funkcja od swojego odpowiednika bez literki v na początku różni się tylko ostatnim argumentem (argumentami). Natomiast pierwsze dwa parametry są identyczne. Funkcja ta nie wywołuje automatycznie makra va_end.

Parametry

Ostatni argument jest to zmienna lista parametrów.

Zwracana wartość

W razie powodzenia funkcja zwraca ilość znaków wyświetlonych (zapisanych). W przypadku nie powodzenia wartość ujemna jest zwracana.

Przykład

Jeden na grupe na zakończenie niniejszego podpunktu.

```
int vsprintf (char *s, const char *format, va_list arg)
```

Opis

Funkcja od swojego odpowiednika bez literki v na początku różni się tylko ostatnim argumentem (argumentami). Natomiast pierwsze dwa parametry są identyczne. Funkcja ta nie wywołuje automatycznie makra va_end.

Parametry

Ostatni argument jest to zmienna lista długości.

Zwracana wartość

W razie powodzenia funkcja zwraca ilość znaków wyświetlonych (zapisanych). W przypadku nie powodzenia wartość ujemna jest zwracana.

Przykład

Przykład obejmuje działanie wszystkich wyżej wymienionych funkcji.

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

```
void newPrint (const char *format, ...);
void newPrint_s (char *tab, const char *format, ...);
void newPrint_v (FILE *stream, const char *format, ...);

int main (void)
{
	char tab[200];
	int ilosc;
	ilosc = sprintf(tab, "Lorem ipsum dolor sit amet");
	fprintf(stdout, "%s\n", tab);
	printf("Ilosc wyswietlonych znakow: %d\n", ilosc);

	newPrint("Ala ma kota, a kot ma %d lat\n", 5);
	newPrint_s (tab, "Ja mam %d lat. Ty masz %d lat\n", 20, 20);
	fprintf(stdout, "%s", tab);

	newPrint_v (stdout, "Ja mam %d lat. Ty masz %d lat\n", 20, 20);

	return 0;
}

void newPrint (const char *format, ...)
{
	va_list argument;
	va_start (argument, format);
	vprintf (format, argument);
	va_end (argument);
}

void newPrint_s (char *tab, const char *format, ...)
{
	va_list argument;
	va_start (argument, format);
	vsprintf (tab, format, argument);
	va_end (argument);
}

void newPrint_v (FILE *stream, const char *format, ...)
{
	va_list argument;
	va_start (argument, format);
	vfprintf (stream, format, argument);
	va_end (argument);
}
```

10.13.3 Formatowane wejście

```
int fscanf (FILE *stream, const char *format, ...)
```

Opis

Funkcja fscanf działa na tej samej zasadzie co te opisane w pukcie 8.3, z tą różnicą, że jako pierwszy argument przyjmuje wskaźnik do pliku, tudzież jeden ze standardowych strumieni.

Parametry

Pierwszy parametr to wskaźnik do pliku, lub jeden ze standardowych strumieni, natomiast pozostałe takie same jak w przypadku funkcji scanf.

Zwracana wartość

W przypadku powodzenia funkcja zwraca ilość wczytanych elementów. W przypadku gdy wystąpi błąd wejścia funkcja zwraca EOF.

Przykład

```
#include <stdio.h>

int main (int argc, char *argv[])
{
      FILE *wp;
      int tab[4];
      int i, k;

      if (argc != 2)
      {
            fprintf(stderr, "Uzycie: %s nazwa_pliku\n", argv[0]);
            return -1;
      }

      if ((wp = fopen(argv[1], "r")) != NULL)
      {
            k = fscanf(wp, "%d-%d-%d-%d", &tab[0], &tab[1], &tab[2],
&tab[3]);
            for (i = 0; i < k; i++)
                  printf("tab[%d] = %d\n", i, tab[i]);
            fclose (wp);
      }
      else
      {
            fprintf(stderr, "Pliku: %s nie udalo sie otworzyc\n", argv[1]);
            return -1;
      }
      return 0;
}
```

Takie małe wyjaśnienie odnośnie powyższego listingu. Trzeba uruchomić program z argumentem, tzn z nazwą pliku, którego zawartość będzie w takiej postaci:

```
X-Y-Z-K
```

Gdzie X, Y, Z i K są dowolnymi liczbami całkowitymi.

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

Opis

Funkcja opisana w punkcie 8.3

`int sscanf (char *s, const char *format, ...)`

Opis

Funkcja opisana w punkcie 8.3

10.13.4 Wejście i wyjście znakowe

`int fgetc (FILE *stream)`

Opis

Funkcja pobierająca jeden znak ze strumienia. Strumieniem może być stdin, lub np. plik.

Parametry

Parametrem jest wskaźnik do pliku, lub stdin.

Zwracana wartość

Funkcja zwraca znak, który pobrała ze strumienia.

Przykład

```
#include <stdio.h>

int main (void)
{
      int i = 0;
      do
      {
            if (i)
                  while (fgetc(stdin) != '\n')
                        ;
            printf("Zakonczyc? [y/n] ");
            i = 1;
      } while (fgetc(stdin) != 'y');
```

```
	return 0;
}
```

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

Opis

Funkcja opisana w pukcie 8.6

`int fputc (int c, FILE *stream)`

Opis

Funkcja ta drukuje znak do określonego strumienia.

Parametry

Pierwszym parametrem jest znak, drugim jest strumień. Strumieniem może być stdout, stderr, lub np. plik.

Zwracana wartość

W przypadku powodzenia pierwszy argument (znak) drukowany jest do określonego strumienia. Jeśli wystąpiły błędy EOF jest zwracane.

Przykład

```
#include <stdio.h>

int main (void)
{
	char info[] = "Bardzo wazna informacja\n";
	int i;

	for (i = 0; i < sizeof (info) / sizeof (info[0]); i++)
		fputc(info[i], stdout);

	return 0;
}
```

`int fputs (char *s, FILE *stream)`

Opis

Funkcja opisana w pukcie 8.6

`int getc (FILE *stream)`

Opis

Funkcja działa analogicznie do funkcji fgetc.

`int getchar (void)`

Opis

Funkcja opisana w pukcie 8.1

`char *gets (char *s)`

Opis

Funkcja gets pobiera znaki od użytkownika i zapisuje je do tablicy s. Funkcja ta jest podobna do funkcji fgets, lecz różni się paroma ważnymi szczegółami. Po pierwsze nie można sprecyzować strumienia, z którego pochodzą znaki, więc można je wprowadzić tylko ze standardowego wejścia (stdin) oraz nie ma możliwości poinformowania o ilości wprowadzonych znaków, co potencjalnie może być przyczyną błędów jeśli wprowadzimy więcej znaków, niż tablica posiada miejsc. Funkcja czyta znaki, aż do napotkania znaku nowej linii, którego nie umieszcza w tablicy. Raczej powinno używać się funkcji fgets, dzięki której mamy pewność, że skopiujemy tylko tyle ile potrzebujemy znaków, dzięki czemu obejdzie się bez pisania po pamięci.

Parametry

Parametrem jest tablica znaków, do której zapisujemy wprowadzony tekst, czyli funkcja zwraca wskaźnik do tablicy s. Jeśli wystąpił błąd to funkcja zwraca NULL.

Zwracana wartość

W przypadku sukcesu, znaki są zapisane w tablicy.

Przykład

```
#include <stdio.h>

int main (void)
{
	char s[40];

	printf("> ");
	gets(s);

	printf("%s\n", s);
	return 0;
}
```

`int putc (int c, FILE *stream)`

Opis

Funkcja działa analogicznie do funkcji fputc.

`int putchar (int c)`

Opis

Funkcja opisana w pukcie 8.1

`int puts (char *s)`

Opis

Funkcja drukuje na standardowe wyjście (stdout) znaki wskazywane przez s. puts kończy drukowanie po napotkaniu znaku końca (**0**). Na koniec drukowanego tekstu dodawany jest znak nowej linii.

Parametry

Parametrem jest wskaźnik do tekstu s.

Zwracana wartość

W przypadku sukcesu funkcja zwraca ilość wydrukowanych znaków.

Przykład

```
#include <stdio.h>

int main (void)
{
      int k;
      char *w = "Aloha";

      k = puts (w);
      printf("%d\n", k);
      return 0;
}
```

10.13.5 Pozycja w pliku

`int fseek (FILE *stream, long offset, int origin)`

Opis

Funkcja ustawia kursor (wskaźnik) związany z strumieniem na inne zdefiniowane miejsce. Robi to poprzez dodanie do pozycji kursora (origin) pewnej ustalonej wartości (offset).

Parametry

Pierwszym parametrem jest wskaźnik do strumienia, najczęściej wskaźnik do pliku. Drugim parametrem jest offset, jest to liczba bajtów dodawanych do pozycji referencyjnej origin, która jest trzecim parametrem. Zmienna origin może przyjmować następujące wartości:

- SEEK_SET – Początek pliku
- SEEK_CUR – Aktualna pozycja w pliku
- SEEK_END – Koniec pliku

Zwracana wartość

W przypadku sukcesu funkcja zwraca zero. Jeśli wystąpiły jakieś błędy funkcja zwraca wartość nie zerową.

Przykład

Jeden na grupę na zakończenie niniejszego podpunktu.

```
long int ftell (FILE *stream)
```

Opis

Funkcja zwraca aktualną pozycję kursora powiązanego z konkretnym strumieniem, zazwyczaj plikiem.

Parametry

Parametrem jest wskaźnik do strumienia, najczęściej pliku.

Zwracana wartość

W przypadku sukcesu funkcja zwraca aktualną wartość kursora. W razie nie powodzenia wartość -1L jest zwracana.

Przykład

Jeden na grupę na zakończenie niniejszego podpunktu.

```
void rewind (FILE *stream)
```

Opis

Funkcja przesuwa kursor na początek pliku.

Parametry

Parametrem jest wskaźnik do pliku.

Zwracana wartość

Brak.

Przykład

Jeden na grupe na zakończenie niniejszego podpunktu.

```
int fgetpos (FILE *stream, fpos_t *ptr)
```

Opis

Funkcja pobiera informacje o pozycji kursora i przechowuje tę informację w specjalnej zmiennej, która jest typu fpos_t. Zmienna ta może być użyta w funkcji fsetpos.

Parametry

Pierwszy parametr to wskaźnik do pliku, drugi parametr to obiekt w którym przechowuje się informacje o położeniu kursora w pliku.

Zwracana wartość

W przypadku sukcesu funkcja zwraca wartość zero, jeśli wystąpiły jakieś błędy to funkcja zwraca wartość nie zerową.

```
int fsetpos (FILE *stream, const fpos_t *ptr)
```

Opis

Funkcja zmienia pozycję kursora w pliku na wartość podaną jako drugi parametr, czyli ptr. Wartość ptr jest dostarczana za pomocą funkcji fgetpos.

Parametry

Pierwszym parametrem jest wskaźnik do pliku, drugim natomiast wskaźnik do obiektu typu fpos_t, którego uzyskano za pomocą funkcji fgetpos.

Zwracana wartość

W przypadku sukcesu funkcja zwraca zero, w razie nie powodzenia wartość nie zerowa zostaje zwrócona oraz zmienna errno zostaje ustawiona na konkretną wartość.

Przykład

```
#include <stdio.h>

int main (int argc, char *argv[])
{
	FILE *wp;
	long int rozmiar;
	fpos_t pp;

	if (argc != 2)
	{
		fprintf(stderr, "Uzycie: %s filename\n", argv[0]);
		return -1;
	}
```

```
	if ((wp = fopen(argv[1], "w")) != NULL)
	{
		fprintf(wp, "Ala ma kota, a kot ma %d lat\n", 5);
		fseek (wp, 22L, SEEK_SET);
		fprintf(wp, "%d", 9);
		fseek (wp, 0L, SEEK_END);
		rozmiar = ftell (wp);
		printf("Rozmiar pliku: %ld bytes\n", rozmiar);
		rewind (wp);
		rozmiar = ftell (wp);
		printf("Kursor znajduje sie na poczatku, dlatego rozmiar pliku 
rowna sie: %ld bytes\n", rozmiar);


		fseek (wp, 0, SEEK_END);
		fgetpos(wp, &pp);
		printf("Koniec pliku znajduje sie na pozycji: %d\n", pp);
		fsetpos (wp, &pp);
		fprintf(wp, "Tekst zostal dodany pozniej, wiec teraz koniec 
pliku jest gdzie indziej\n");

		fseek (wp, 0, SEEK_END);
		fgetpos(wp, &pp);
		printf("Nowy rozmiar pliku to: %d bytes\n", pp);

		fclose(wp);
	}
	return 0;
}
```

Omówienie programu

Po otwarciu pliku zapisujemy do niego zdanie, w którym występuje liczba lat równa 5. Następnie przesuwany kursor (czyli pozycję od której zaczynamy pisać) na 22. miejsce licząc od początku pliku (origin = SEEK_SET) a następnie wpisujemy do pliku na następną pozycję liczbę 9. Dlatego też, po wyświetleniu zawartości pliku liczbą lat kota będzie 9, a nie 5. Następnie przesuwamy kursor na koniec pliku i pobieramy aktualną pozycję kursora (ilość znaków) za pomocą ftell. Z użyciem rewind przesunęliśmy kursor na początek, dlatego też wywołanie ftell da wartość zero. W następnej linii przesuwamy kursor na koniec, a następnie zapamiętujemy tę pozycję za pomocą fgetpos w zmiennej pp. Ustawiamy pozycję w pliku na wartość, którą przed chwilą pobraliśmy i wpisujemy kolejny ciąg znaków. Poprzednie wywołania funkcji sprawdzających rozmiar pliku były dla pliku, który nie miał nowego ciągu znaków, dlatego faktycznie różni się on od poprzedniego, co udowadniają ostatnie wywołania funkcji.

10.13.6 Obsługa błędów

```
void clearerr (FILE *stream)
```

Opis

Kiedy funkcja działająca na strumieniach (pliku) zawiedzie z powodu wykrycia końca pliku, lub z powodu innego błędu wskaźnik błędu może zostać ustawiony. Funkcja ta czyści informacje o tych błędach.

Parametry

Parametrem jest wskaźnik do pliku.

Zwracana wartość

Brak.

Przykład

```
#include <stdio.h>

int main (void)
{
      FILE *wp;
      int znak;

      if ((wp = fopen("plik1", "r")) == NULL)
      {
            perror("Blad otwarcia");
            return -1;
      }

      fputc('A', wp);
      if (ferror(wp))
      {
            perror("Blad zapisu");
            clearerr(wp);
      }

      znak = fgetc(wp);
      if (!ferror(wp))
            printf("Pobrano znak: %c\n", znak);

      return 0;
}
```

Omówienie programu

Ponieważ funkcja ferror zwraca wartość nie zerową jeśli przy ostatniej operacji na pliku wystąpił błąd

to informacja o błędzie zostaje wyświetlona. Gdyby nie było wywołania funkcji clearerr to przy następnym wywołaniu funkcji ferror znów mielibyśmy wartość nie zerową dlatego też informacja o pobranym znaku nie została by wyświetlona. Funkcja clearerr czyści te informacje i dlatego też po pobraniu znaku nie ma żadnego błędu, a więc informacja zostaje wyświetlona.

```
int feof (FILE *stream)
```

Opis

Funkcja feof sprawdza czy kursor (wskaźnik pozycji) skojarzony z konkretnym strumieniem jest ustawiony na wartość EOF, czyli End-Of-File.

Parametry

Parametrem jest wskaźnik do pliku, lub strumienia stdin.

Zwracana wartość

Funkcja zwraca wartość nie zerową, jeśli kursor jest ustawiony na EOF. W przeciwnym wypadku zwraca wartość zero.

```
int ferror (FILE *stream)
```

Opis

Funkcja sprawdza czy ostatnia wykonana operacja na pliku nie powiodła się (czy wskaźnik błędu jest ustawiony). W gruncie rzeczy wskaźnik błędu ustawiany jest w momencie, kiedy ostatnia operacja na pliku zawiedzie.

Parametry

Parametrem jest wskaźnik do pliku.

Zwracana wartość

Jeśli poprzednia operacja na pliku nie powiodła się to funkcja zwraca wartość nie zerową. Jeśli wszystko poszło zgodnie z założeniami zero jest zwracane.

```
void perror (const char *s)
```

Opis

Funkcja służy do wyświetlania informacji o błędzie. Interpretuje wartość zmiennej errno i wyświetla stosowny komunikat, który związany jest z odpowiednią jej wartością. Funkcja drukuje komunikat

w następującej postaci:

```
Nie mozna otworzyc pliku: No such file or directory
```

Pierwsza część (do dwukropka) jest ciągiem znaków wskazywanych przez s, druga część jest to tekst związany z konkretną wartością errno.

Parametry

Funkcja pobiera wskaźnik do tekstu.

Zwracana wartość

Brak.

Przykład

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

void wpisz (char *filename);
void odczytaj (char *filename);

int main (int argc, char *argv[])
{
	if (argc != 2)
	{
		fprintf(stderr, "Uzycie: %s filename\n", argv[0]);
		return -1;
	}

	//wpisz (argv[1]);
	odczytaj (argv[1]);

	return 0;
}

void wpisz (char *filename)
{
	FILE *wp;

	if ((wp = fopen(filename, "r")) != NULL)
	{
		fputc('A', wp);
		if (ferror(wp))
			perror("Blad zapisu");
		fclose(wp);
	}
	else
	{
```

```
            perror ("Nie mozna otworzyc pliku");
            exit (-1);
      }
}

void odczytaj (char *filename)
{
      FILE *wp;
      int znak;

      if ((wp = fopen(filename, "r")) != NULL)
      {
            znak = fgetc(wp);

            while (znak != EOF)
            {
                  putchar(znak);
                  znak = fgetc(wp);
            }
            if (feof(wp))
                  fprintf(stderr, "-- KONIEC PLIKU --\n");

            fclose(wp);
      }
      else
      {
            perror ("Nie mozna otworzyc pliku");
            exit (-1);
      }
}
```

Omówienie programu

W funkcji wpisz celowo ustawiono tryb otwarcie jako **r**, żeby pokazać jaki błąd zostaje wyświetlony podczas próby wpisania do tak otwartego pliku czegokolwiek. Funkcja odczytaj ma za zadanie wyświetlić na ekranie wszystkie znaki z danego pliku, a na końcu poinformować użytkownika o tym, że nastąpił koniec pliku. Do zmiennej znak przypisujemy wywołanie funkcji pobierającej znak z pliku, a następnie sprawdzamy, czy ten znak jest różny od EOF (koniec pliku) jeśli tak jest, to drukujemy go, a następnie powtarzamy przypisanie, aby pętla działała do czasu, aż osiągnie EOF. Następnie sprawdzamy czy feof zwróciła wartość nie zerową (będzie tak, jeśli kursor zostanie ustawiony na EOF) i jeśli tak jest to drukujemy na stderr informacje o końcu pliku.

10.14 stdlib.h

W nagłówku stdlib.h znajdują się różne użyteczne funkcje, takie jak komunikacja ze środowiskiem (system operacyjny), dynamicznie przydzielana pamięć, przekształcanie liczb, sortowanie, itp. W poniższej tabeli zostały zestawione te funkcje, a pod tabelą opis wraz z przykładami.

Nazwa	Opis
Konwersja ciągu znaków na liczby	
atof	Przekształcenie znaków na liczbę typu double
strtod	
atoi	Przekształcenie znaków na liczbę typu int
atol	Przekształcenie znaków na liczbę typu long int
strtol	
strtoul	Przekształcenie znaków na liczbę typu unsigned long int
Pseudo-losowe liczby	
rand	Generowanie liczb pseudo-losowych
srand	Inicjowanie generatora liczb pseudo-losowych
Dynamicznie przydzielana pamięć	
calloc	Alokowanie pamięci dla tablicy
malloc	Alokowanie pamięci (blok)
realloc	Ponowna alokacja pamięci
free	Zwalnienie zaalokowanej pamięci
Funkcje oddziaływujące ze środowiskiem uruchomienia	
abort	Przerwanie procesu
atexit	Uruchomienia funkcji po zakończeniu programu
exit	Zakończenie wzywanego procesu
getenv	Zwrócenie wartości przez system
system	Wywołanie polecenia systemowego
Wyszukiwanie i sortowanie	
bsearch	Wyszukiwanie w tablicy
qsort	Sortowanie elementów tablicy
Arytmetyka liczb całkowitych	
abs	Wartość bezwzględna
labs	

div	Dzielenie liczb całkowitych
ldiv	

10.14.1 Konwersja ciągu znaków na liczby

`double atof (const char *str)`

Opis

Funkcja konwertująca tekst na liczbę typu double. Przede wszystkim pomijane są początkowe białe znaki, następnie po napotkaniu pierwszego nie białego znaku funkcja sprawdza czy jest to poprawny znak do skonwertowania, jeśli tak jest to przechodzi do kolejnego znaku. W momencie napotkania pierwszego nie poprawnego znaku funkcja kończy konwertowanie, a reszta znaków jest ignorowana. Poprawnymi znakami są:

- Opcjonalny znak plus (+) lub minus (-)
- Cyfry wliczając w to kropkę dziesiętną
- Opcjonalny wykładnik – litera **e**, lub **E** ze znakiem oraz cyfry (np. 1E+2)

Jeśli pierwszy znak w ciągu nie jest poprawnym znakiem to konwersja nie jest wykonywana.

Parametry

Parametrem jest ciąg znaków, może to być tablica znaków, lub wskaźnik wskazujący na dowolny ciąg znaków.

Zwracana wartość

W przypadku sukcesu funkcja zwraca liczbę rzeczywistą typu double. Jeśli konwersja nie została wykonana to wartość 0.0 jest zwracana.

Przykład

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

int main (int argc, char *argv[])
{
      double *w;
      int i;
```

```
	w = (double *) calloc(argc-1, sizeof (double));

	if (argc < 2)
	{
		fprintf(stderr, "Uzycie: %s parametr(y)\n", argv[0]);
		return -1;
	}

	for (i = 1; i <= argc - 1; i++)
		w[i-1] = atof(argv[i]);
	for (i = 0; i < argc - 1; i++)
		printf("w[%d] = %lf\n", i, w[i]);
	free(w);

	return 0;
}
```

Omówienie programu

Aby mieć możliwość wpisywania nie ograniczonej liczby argumentów (liczb do skonwertowania) musimy skorzystać z dynamicznie przydzielanej pamięci (opisane w rozdziale 9.). Alokujemy miejsca na (argc-1) · sizeof (double) ponieważ pierwszym argumentem jest nazwa programu. Ponieważ zaalokowaną pamięć indeksujemy od zera, tak więc indeks w musimy obniżyć o jeden.

```
double strtod (const char *str, char **ptr)
```

Opis

Funkcja strtod jest bardzo podobna do funkcji atof lecz różni się pewną rzeczą, a mianowicie chodzi o drugi parametr. Tak jak w atof funkcja strtod sprawdza po kolei znaki, jeśli trafi na nie poprawny znak to przerywa swoje działanie, a pobrane znaki konwertuje do liczby typu double, jeśli ptr nie jest NULL to reszta znaków wskazywana jest przez ptr. Poprawnymi znakami są te same znaki, co w funkcji powyżej.

Parametry

Pierwszym parametrem jest tablica znaków, lub wskaźnik wskazujący na ciąg znaków. Drugim parametrem jest wskaźnik do wskaźnika na ciąg znaków.

Zwracana wartość

W przypadku sukcesu funkcja zwraca skonwertowaną liczbę w postaci liczby typu double. Jeśli nie było znaków do skonwertowania to funkcja zwraca wartość 0.0.

Przykład

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

int main (void)
{
	char *str = "10.3 1000.43232";
	char *ptr;
	double dl;

	dl = strtod(str, &ptr);

	printf("%f %s\n", dl, ptr);
	dl = strtod(ptr, NULL);
	printf("%f\n", dl);

	return 0;
}
```

Omówienie programu

Wskaźnik str wskazuje na dwie liczby oddzielone spacją (tak naprawdę jest to ciąg znaków, póki co, nie są to liczby typu double). Do zmiennej dl przypisujemy wywołanie strtod z argumentami str, oraz &ptr. Ampersand jest istotny ponieważ prototyp funkcji informuje nas, że funkcja przyjmuje wskaźnik do wskaźnika, czyli adres wskaźnika. Po wykonaniu tej funkcji w zmiennej dl posiadamy już pierwszą część (część do pierwszego nie poprawnego znaku, tutaj spacja) ciągu znaków, a wskaźnik ptr wskazuje na pozostałą część. Kolejne wywołanie przypisuje do dl skonwertowany pozostały ciąg znaków. Jako drugi argument podajemy NULL.

```
int atoi (cont char *str)
```

Opis

Funkcja bardzo podobna do powyższych z tą różnicą, że konwertuje znaki do liczby typu całkowitego. Tak samo jak w przypadku funkcji atof, funkcja najpierw odrzuca wszystkie początkowe białe znaki, a następnie sprawdza czy pierwszy nie biały znak jest poprawnym znakiem. Jeśli tak jest to kolejne znaki są sprawdzane, aż do napotkania pierwszego nie poprawnego znaku. Jeśli pierwszym nie poprawnym znakiem jest biały znak lub ciąg znaków zawiera same białe znaki, to konwersja nie jest wykonywana. Poprawnymi znakami są cyfry oraz opcjonalny znak plus lub minus poprzedzający cyfry.

Parametry

Parametrem jest tablica znaków, lub wskaźnik do ciągu znaków.

Zwracana wartość

W przypadku sukcesu funkcja zwraca skonwertowaną liczbę jako liczbę typu całkowitego. Jeśli w tablicy nie było żadnego poprawnego znaku to zero jest zwracane. Jeśli liczba jest spoza zakresu to stała INT_MAX, lub INT_MIN jest zwracana.

Przykład

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

int main (void)
{
	char *wskTab[] = {"   100", "  1E5", "-4.3", "  2.3", "   ",
	           "98293489238492384923"};
	int rozm = sizeof (wskTab) / sizeof (wskTab[0]);
	int i;
	int tab[rozm];

	for (i = 0; i < rozm; i++)
		tab[i] = atoi(wskTab[i]);
	for (i = 0; i < rozm; i++)
		printf("%d\n", tab[i]);

	return 0;
}
```

Omówienie programu

Aby pokazać różne przykłady i nie tworzyć wielu tablic znaków utworzono tablicę wskaźników (opisane w punkcie 5.10). Wyniki są przewidywalne, lecz ostatnia pozycja (bardzo duża liczba) może okazać się ciekawa. Zwrócona zostaje wartość INT_MAX opisana w punkcie 10.6, analogicznie byłoby w przypadku bardzo dużej liczby ujemnej.

```
long int atol (const char *str)
```

Opis

Funkcja jest analogiczna do atoi z tą różnicą, że zwraca wartości w postaci liczby typu long int. W przypadku przekroczenia zakresu zwracane są wartości LONG_MAX, lub LONG_MIN opisane w punkcie 10.6.

Parametry

Parametrem jest tablica znaków, lub wskaźnik do ciągu znaków.

Zwracana wartość

W razie sukcesu funkcja zwraca liczbę w postaci long int. Jeśli nie było poprawnych znaków wartość zero jest zwracana.

Przykład

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
	char *tab = "9832328";
	long lg;
	lg = atol(tab);
	printf("%ld\n", lg);

	return 0;
}
```

```
long int strtol (const char *str, char **ptr, int base)
```

Opis

Działanie funkcji strtol jest bardzo podobne do omówionych już funkcji. Róznica między nimi jest następująca, a mianowicie możemy podać w jakim systemie liczbowym liczba jest zapisana (parametr base – podstawa systemu liczbowego). Zasady odnośnie poprawnych znaków i drugiego parametru są analogiczne do tych przedstawionych w poprzednich funkcjach z tą różnicą, że poprawne znaki ustalane są na podstawie parametru base (dla **hex** nie dozwolonym znakiem będzie np. Z, a dla binarnego wszystkie za wyjątkiem 0 i 1). Dozwolonymi wartościami dla parametru base jest wartość z przedziału <2; 36>.

Parametry

Pierwszy parametr to tablica znaków, lub wskaźnik na ciąg znaków. Drugim parametrem jest adres wskaźnika na ciąg znaków. Trzecim parametrem jest podstawa systemu liczbowego.

Zwracana wartość

W przypadku sukcesu funkcja zwraca skonwertowaną liczbę w postaci long int. Jeśli nie można było przeprowadzić konwersacji znaków z powodu, że znaki nie należą do danego systemu liczbowego wartość zero jest zwracana. Jeśli skonwertowana liczba jest poza dopuszczalnym zakresem danego typu to wartość LONG_MAX, lub LONG_MIN jest zwracana.

Przykład

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

int main (void)
{
      char *wskTab[] = {" 100", "0xAAA", "001011010101", "cacaca", "0779"};
      int rozmiar = sizeof (wskTab) / sizeof (wskTab[0]);
      long int tab[rozmiar];

      tab[0] = strtol(wskTab[0], NULL, 10);
      tab[1] = strtol(wskTab[1], NULL, 16);
      tab[2] = strtol(wskTab[2], NULL, 2);
      tab[3] = strtol(wskTab[3], NULL, 16);
      tab[4] = strtol(wskTab[4], NULL, 8);

      printf("%ld\n%ld\n%ld\n%ld\n%ld\n", tab[0], tab[1], tab[2], tab[3],
tab[4]);

      return 0;
}
```

Omówienie programu

Weryfikacja czy znak należy do danego systemu liczbowego odbywa się za pośrednictwem trzeciego parametru funkcji strtod. Znaki **0x** poprzedzające liczbę szesnastkową mogą lecz nie muszą występować. Jak widać ostatnia liczba poprzedzona znakiem 0 jest liczbą ósemkową, konwertowanie jej kończy się na cyfrze 9, ponieważ ta liczba nie wchodzi w skład ósemkowego systemu liczbowego. Podobnie jak dla systemu szestnastkowego w systemie ósemkowym nie musi występować na początku zero.

```
unsigned long int strtoul (const char *str, char **ptr, int base)
```

Opis

Funkcja jest analogiczna do funkcji strtol, z tą różnicą, że zwraca wartość typu unsigned long int. Wszystkie zasady są analogiczne.

Parametry

Takie same jak dla strtol.

Zwracana wartość

W przypadku sukcesu funkcja zwraca skonwertowaną liczbę w postaci unsigned long int. Jeśli

konwersji nie udało się wykonować wartość zero jest zwracana. Jeśli skonwertowana liczba jest spoza dopuszczalnego zakresu wartość ULONG_MAX jest zwracana.

10.14.2 Pseudo-losowe liczby

```
int rand (void)
```

Opis

Funkcja zwracająca liczbę pseudo-losową. Opis tej funkcji i całego mechanizmu z jakiego się korzysta został przedstawiony w rozdziale 9. Aby liczba pseudo-losowa była z zawężonego przedziału należy użyć operacji modulo (rand() % wartosc).

Parametry

Brak.

Zwracana wartość

Funkcja zwraca psuedo losową liczbę z zakresu <0; RAND_MAX>.

```
void srand (unsigned int seed)
```

Opis

Funkcja jest generatorem liczb pseudo-losowych. Jako parametr przyjmuje liczbę na podstawie której generuje wedug określonego algorytmu liczbę pseudo-losową wyświetlaną za pomocą funkcji rand. Aby funkcja rand losowała za każdym razem inne wartości, argument funkcji srand musi być różny, można to osiągnąć za pomocą funkcji time.

Parametry

Dodatnia liczba całkowita.

Zwracana wartość

Brak.

Przykład

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

int main (void)
{
```

```
	int random;
	printf("Zarodek staly: ");
	srand(1);
	random = rand();
	printf("%d\n", random % 1000);

	printf("Zarodek zmienny: ");
	srand(time(0));
	random = rand();
	printf("%d\n", random % 1000);

	return 0;
}
```

Omówienie programu

Wartość random % 1000 losuje liczbę z zakresu <0; 999>.

10.14.3 Dynamicznie przydzielana pamięć

```
void *calloc (size_t n, size_t size)
```

Opis

Funkcja szczegółowo opisana w rozdziale 9.

Parametry

Pierwszym parametrym jest ilość n (elementów tablicy), drugim jest rozmiar pojedynczego elementu. size_t jest typem danych liczb całkowitych dodatnich, tak więc można z powodzeniem używać unsigned int.

Zwracana wartość

Funkcja zwraca wskaźnik do zaalokowanego miejsca. Przed użyciem należy zrzutować wskaźnik na konkretny typ.

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

Opis

Funkcja szczegółowo opisana w rozdziale 9.

Parametry

Parametrem jest dodatnia liczba całkowita, która definiuje jak wielki blok (w bajtach) ma zostać

zarezerwowany.

Zwracana wartość

Funkcja zwraca wskaźnik do początku zarezerwowanego bloku o rozmiarze size bajtów.

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

Opis

Ponieważ funkcja realloc nie została opisana w rozdziale 9. pozwolę sobie tutaj ją omówić, a wraz z opisem pokazać przykład zastosowania. Funkcja realloc zmienia rozmiar zarezerwowanej pamięci wskazywanej przez ptr. Zwiększa lub zmniejsza, w razie konieczności przenosi cały blok w inne miejsce i zwraca wskaźnik do początku bloku pamięci. Przy pewnych warunkach funkcja realloc zachowuje się jak funkcja free, a przy innych jak funkcja malloc. Jeśli jako pierwszy parametr podamy NULL to funkcja rezerwuje pamięć dokładnie tak samo jak malloc. Natomiast jeśli jako size podamy zero, to funkcja zwalnia pamięć w taki sam sposób jak free.

Parametry

Pierwszym parametrem jest wskaźnik do zaalokowanej wcześniej pamięci za pomocą funkcji malloc, calloc lub realloc. Drugim parametrem jest rozmiar bloku w bajtach.

Zwracana wartość

Funkcja zwraca wskaźnik do początku bloku zarezerwowanej na nowo pamięci. Jeśli podczas realokacji blok pamięci został przeniesiony w inne miejsce to funkcja zwraca wskaźnik do tego miejsca. Jeśli nie powiedzie się realokacja pamięci to funkcja zwraca NULL, natomiast blok pamięci wskazywany przez ptr zostaje nie ruszony. Jeśli realloc użyjemy jako free to NULL jest zwracane.

Przykład

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

int main (void)
{
      int i, ilosc = 15;
      int *w = (int *) calloc(10, sizeof (int));
      w = realloc (w, ilosc*sizeof(int));

      srand(time(0));
      for (i = 0; i < ilosc; i++)
            w[i] = rand() % 10;
```

```
	for (i = 0; i < ilosc; i++)
		printf("w[%d] = %d\n", i, w[i]);

	if (realloc (w, 0) == NULL)
		printf("Pamiec wyczyszczona\n");

	return 0;
}
```

Omówienie programu

Na początku za pomocą funkcji calloc rezerwujemy miejsce dla 10 elementów tablicy o rozmiarze pojedynczego elementu typu int. W kolejnej linijce realokujemy pamięc, jako pierwszy argument podajemy w, a jako drugi argument podajemy wyrażenie takie jakie widać, ponieważ musimy podać ilość miejsca w bajtach, więc 15 · sizeof (int) pozwoli na zapisanie 15 elementów tablicy liczb całkowitych. Należy zwrócić uwagę na to, że jeśli nie było by wywołania tej funkcji program się wysypie, ponieważ w instrukcjach for będziemy odwoływać się do nie dozwolonego miejsca. Instrukcja if pokazuje, że wywołanie funkcji realloc z drugim parametrem jako zero zwraca NULL jeśli pamięć zostanie wyczyszczona.

`void free (void *ptr)`

Opis

Funkcja opisana w rozdziale 9.

Parametry

Wskaźnik do zaalokowanej wcześniej pamięci.

Zwracana wartość

Brak

10.14.4 Funkcje oddziaływujące ze środowiskiem uruchomienia

`void abort (void)`

Opis

Funkcja kończy działanie programu w nie prawidłowy sposób. abort generuje sygnał **SIGABRT**, który

kończy program wraz ze zwróceniem kodu błędu odpowiadającemu nie prawidłowemu zakończeniu programu. Kod ten zwracany jest do środowiska wywołania programu, a nie w miejscu wywołania funkcji.

Parametr

Brak

Zwracana wartość

Brak

Przykład

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

int main (void)
{
	double n = 0.0, k = 10.0;

	if (n != 0)
		printf("%f\n", k/n);
	else
		abort();
	return 0;
}
```

Aby sprawdzić jaki kod został zwrócony do środowiska, wystarczy, że wpiszesz następujące polecenie, po zakończeniu programu.

```
$ echo $?
```

```
int atexit (void (* function) (void))
```

Opis

Funkcja atexit używana jest wtedy, gdy chcemy wykonać jakąś funkcję na zakonczenie działania programu. Funkcja podana jako parametr atexit wykona się tylko wtedy, gdy program zakończy się w prawidłowy (poprawny) sposób. Jeśli występuje więcej wywołań funkcji atexit w jednym programie, to kolejność wykonywania wskazywanych funkcji jest odwrotna (jeśli w programie występują dwa wywołania atexit, to najpierw wykona się funkcja z drugiego wywołania, a później dopiero z pierwszego). Podając wskaźnik do funkcji mamy na myśli, że rejestrujemy ją do wykonania na zakończenie.

Parametry

Parametrem jest wskaźnik do funkcji (nazwa funkcji bez nawiasów).

Zwracana wartość

Jeśli funkcja została pomyślnie zarejestrowana to zero jest zwracane, w przeciwnym wypadku wartość nie zerowa jest zwracana.

Przykład 1

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

void goodbye (void);
void goodmorning (void);

int main (void)
{
      atexit(goodbye);
      atexit(goodmorning);
      printf("Zaraz program sie zakonczy\n");
      return 0;
}

void goodbye (void)
{
      printf("Program skonczyl swoje dzialanie, dowidzenia!\n");
}

void goodmorning (void)
{
      printf("Program jednak skonczyl swoje dzialanie\n");
}
```

Omówienie programu 1

Najpierw pojawi się napis z instrukcji printf (funkcja main), dlatego, że program jeszcze się nie skończył. Zaraz po tym występuje return, tak więc wykonywane są funkcje zarejestrowane przez atexit. Jak powiedziane było, jeśli jest więcej wywołań atexit, to wykonywane są w odwrotnej kolejności, dlatego też najpierw wyświetlony zostanie napis z funkcji goodmorning, a następnie z funkcji goodbye.

Przykład 2

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

```
void napis (void);

int main (void)
{
	atexit(napis);
	abort();
	return 0;
}

void napis (void)
{
	printf("Ten napis nie zostanie wyswietlony\n");
}
```

Omówienie programu 2

Jak wspomniano, funkcje zarejestrowane przez atexit wykonują się wtedy i tylko wtedy, gdy program kończy swoje działanie normalnie. Dlatego też w tym przypadku nie wykona się funkcja napis, ponieważ program jest przerywany przez funkcję abort.

```
void exit (int status)
```

Opis

Funkcja ta wywołana w funkcji main działa tak samo jak return, tzn przerwywa wykonywanie programu w sposób poprawny i zwraca wartość liczbową podaną jako argument do miejsca wywołania (konsola systemu operacyjnego). Jak wiemy funkcja return użyta w innej funkcji przerywa działanie tej funkcji i zwraca sterowanie do miejsca wywołania. Funkcja exit wywołana w innej funkcji przerywa działanie całego programu. Funkcja exit dodatkowo wywołuje funkcję fclose dla wszystkich otwartych plików.

Parametry

Parametrem jest liczba typu całkowitego, która zostanie zwrócona do miejsca wywołania programu.

Zwracana wartość

Brak

Przykład

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

void goodbye (void);
```

```
void terminate (void);

int main (void)
{
	atexit(goodbye);
	terminate();
}

void goodbye (void)
{
	printf("Dowidzenia\n");
}

void terminate (void)
{
	exit (0);
}
```

Opis programu

Jak widać program jest zakończony za pomocą funkcji terminate. Dzięki temu, że funkcja exit kończy program w poprawny sposób funkcja goodbye wykona się na zakończenie.

`char *getenv (const char *name)`

Opis

Funkcja pobiera nazwę zmiennej środowiskowej (*ang. environment variable*), która posiada pewną wartość. Wskaźnik odnoszący się do ciągu znaków określających wartość owej zmiennej zostaje zwrócony. Jeśli parametrem funkcji getenv nie jest zmienna środowiskowa to NULL jest zwracane.

Parametry

Parametrem jest nazwa zmiennej środowiskowej.

Zwracana wartość

W przypadku, gdy podanym parametrem jest istniejąca nazwa zmiennej środowiskowej, wskaźnik do tej wartości jest zwracany. Jeśli dana zmienna nie istnieje, to NULL jest zwracane.

Przykład

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

int main (int argc, char *argv[])
{
	char *w;
```

```
	int i;

	if (argc < 2)
	{
		fprintf(stderr, "Uzycie: %s env_variable(s)\n", argv[0]);
		return -1;
	}

	for (i = 1; i < argc; i++)
	{
		w = getenv(argv[i]);
		if (w != NULL)
			printf("%s\n", w);
		else
			fprintf(stderr, "%s - nie jest zmienna srodowiskowa\n",
argv[i]);
	}

	return 0;
}
```

Zmiennymi środowiskowymi są np. **PATH**, **HOME**, **USER**, **SHELL**. Równie dobrze można stworzyć swoją zmienną środowiskową i sprawdzić, czy faktycznie dana wartość, którą przypisaliśmy kryje się pod nią za pomocą powyższego programu. Aby stworzyć zmienną środowiskową wpisz poniższe polecenie.

```
$ export VARNAME=154
```

Uruchomienie tego programu może wyglądać następująco.

```
./main PATH HOME USER SHELL VARNAME DDD
```

Gdzie w przypadku **DDD** zostanie wyświetlony komunikat, że nie jest to zmienna środowiskowa.

`int system (const char *command)`

Opis

Funkcja wykonuje polecenie systemowe. Po wykonaniu polecenia kontrola powraca do programu z wartością typu int.

Parametry

Parametrem jest ciąg znaków będący poleceniem systemowym, np. **ls -l**.

Zwracana wartość

Jeśli udało się wykonać polecenie to zero jest zwracane, w przeciwnym przypadku zwracana jest wartość nie zerowa.

Przykład

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

int main (void)
{
	char *wsk[3] = {
			"ls -l", "ps -Al", "ls -1"
	};
	char num[2];
	int n;

	printf("1 - Wyswietl ze szczegolami zawartosc\n");
	printf("2 - Wyswietl liste procesow\n");
	printf("3 - Wyswietl zawartosc, kazdy plik w nowej linii\n");

	do
	{
		printf(": ");
		fgets(num, sizeof (num), stdin);
		n = atoi(num);
	} while (n < 1 || n > 3);

	system (wsk[n-1]);

	return 0;
}
```

Należy wziąć pod uwagę fakt, iż wyświetlanie menu w razie pomylenia się użytkownika podczas wprowadzania nie jest wolne od „błędów”. Należało by zastosować pewne sztuczki, aby wyświetlanie było bardziej efektywne.

10.14.5 Wyszukiwanie i sortowanie

```
void qsort (void *base, size_t n, size_t size, int (* comparator)
                   (const void *, const void *))
```

Opis

Funkcja sortująca n elementów tablicy wskazywanej przez base, w której rozmiar każdego elementu

wynosi size. Funkcja comparator używana jest do określenia kolejności elementów (rosnąco, malejąco).

Parametry

Pierwszym parametrem jest wskaźnik do tablicy, drugim jest ilość elementów tablicy, trzecim jest rozmiar pojedynczego elementu tablicy, czwartym jest wskaźnik na funkcję porównującą dwa elementy. Co do ostatniego parametru, to należy się więcej szczegółów, a mianowicie:

- funkcja musi pobierać dwa argumenty zdefiniowane jako void *.
- funkcja musi zwracać wartość identyfikującą, który z elementów jest większy (dla kolejności rosnącej):
    - jeśli elem1 < elem2, zwróc wartość ujemną (dodatnią dla kolejności malejącej)
    - jeśli elem1 = elem2, zwróc zero
    - jeśli elem1 > elem2, zwróć wartość dodatnią (ujemną dla kolejności malejącej)

Zwracana wartość

Brak.

Przykład

```
#include <stdio.h>
int porownaj (const void *elem1, const void *elem2);
void wyswietl (int tab[], int size);
int main (void)
{
      int tab[] = {12, 22, 10, 3, 20, 98, 32, 45};
      int size = sizeof (tab[0]);
      int num = sizeof (tab)/size;

      wyswietl(tab, num);
      qsort (tab, num, size, porownaj);
      wyswietl(tab, num);

      return 0;
}

int porownaj (const void *elem1, const void *elem2)
{
      return (*(int *)elem1 - *(int *)elem2);
}

void wyswietl (int tab[], int size)
{
```

```
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", tab[i]);
    printf("\n");
}
```

Omówienie programu

Wyjaśnienie oczywiście się należy. Tworzymy tablicę z zainicjowanymi wartościami. Zmienne pomocnicze będą przechowywały pewne wartości, a mianowicie: size posiada wartość pojedynczego elementu, w tym przypadku rozmiar typu int, num ilość elementów tablicy. Funkcja wyswietl wyświetla po prostu wszystkie elementy tablicy. Ciekawa sytuacja jest w funkcji porownaj przypomnij sobie sytuację, która zaistniała w punkcie 5.3, wtedy gdy zmienialiśmy wartość stałej. Musieliśmy przekazać jako argument adres, lecz zrzutowany na typ int *. Ponieważ nie możemy użyć operatora dereferencji do typu void *, to musimy go zrzutować na typ int *. Ujeliśmy to w nawias, dlatego, że po zrzutowaniu musimy wyciągnąć kryjącą się pod tym adresem wartość, dlatego też jest jedna gwiazdka z przodu. Na pierwszy rzut oka może wydawać się to dziwne, może trudne, lecz w gruncie rzeczy jest to odejmowanie dwóch wartości, kryjących się pod wskazanymi adresami. Jeśli elem1 jest większe to wynikiem będzie liczba dodatnia, jeśli są równe to zero, w przeciwnym wypadku wartość ujemna zostanie zwrócona.

```
void *bsearch (const void *key, const void *base, size_t n,
size_t size, int (* comparator) (const void *, const void *))
```

Opis

Funkcja szuka podanej wartości w tablicy o ilości elementów n wskazywanej przez base, gdzie każdy element zajmuje size bajtów. Funkcja zwraca wskaźnik do wystąpienia elementu.

Parametry

Pierwszym parametrem jest wskaźnik do elementu szukanego, drugim jest wskaźnik do tablicy, którą przeszukujemy, trzecim jest ilość elementów tablicy, czwartym rozmiar pojedynczego elementu, a ostatnim parametrem jest funkcja porównująca – analogiczna jak w funkcji qsort. Ponieważ przeszukiwanie jest binarne, to funkcja bsearch wymaga, aby przeszukiwana tablica była posortowana w porządku rosnącym.

Zwracana wartość

W przypadku znalezienia szukanego elementu funkcja zwraca wskaźnik do niego. Jeśli element nie

występował w tablicy to funkcja zwraca NULL.

Przykład

```
#include <stdio.h>
int porownaj (const void *elem1, const void *elem2);
void wyswietl (int tab[], int size);
int main (void)
{
      int tab[] = {12, 22, 10, 3, 20, 98, 32, 45};
      int size = sizeof (tab[0]);
      int num = sizeof (tab)/size;
      int l = 22;
      int *w;

      wyswietl(tab, num);
      qsort (tab, num, size, porownaj);
      wyswietl(tab, num);

      w = (int *)bsearch((void *)&l, tab, num, size, porownaj);

      if (w != NULL)
            printf("Wartosc: %d znajduje sie pod adresem: %p\n", *w, w);
      else
            printf("Wartosc: %d nie znajduje sie w tablicy\n", l);

      return 0;
}

int porownaj (const void *elem1, const void *elem2)
{
      return (*(int *)elem1 - *(int *)elem2);
}

void wyswietl (int tab[], int size)
{
      int i;
      for (i = 0; i < size; i++)
            printf("%d ", tab[i]);
      printf("\n");
}
```

Omówienie programu

Większość jest taka sama jak w przypadku qsort, ponieważ aby wyszukiwać elementu tablicy za pomocą bsearch tablica musi być posortowana w kolejności rosnącej. Funkcja zwraca wskaźnik void *, dlatego też musimy zrzutować go na typ int *, dzięki czemu będziemy mogli wyświetlić jego wartość, jeśli zostanie ona znaleziona, oraz adres. W pierwszym argumencie pobieramy adres zmiennej l, a następnie rzutujemy ją na void *, ponieważ wymaga tego deklaracja pierwszego parametru.

10.14.6 Arytmetyka liczb całkowitych

`int abs (int n)`

Opis

Funkcja zwracająca wartość bezwzględną liczby całkowitej.

Parametry

Parametrem jest liczba całkowita.

Zwracana wartość

Zwracaną wartością jest wartość bezwzględna liczby n.

Przykład

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

int main (void)
{
	int n = -10;

	printf("%d\n", n);
	n = abs(n);
	printf("%d\n", n);

	return 0;
}
```

`long int labs (long int n)`

Opis

Funkcja analogiczna do abs.

Parametry

Liczba typu long int.

Zwracana wartość

Wartość bezwzględna liczby n zwrócona jako typ long int.

`div_t div (int n, int d)`

Opis

Funkcja wykonująca operację dzielenia na liczbach całkowitych. Wartość umieszczana jest

w strukturze typu div_t, która posiada dwa pola typu całkowitego quot oraz rem, które odpowiednio przechowują wynik dzielenia całkowitego oraz resztę z dzielenia.

Parametry

Pierwszym parametrem jest dzielna, drugim dzielnik.

Zwracana wartość

Wartość zwracana jest do struktury. Do pola quot zostaje przypisana wartość dzielenia całkowitego, a do rem reszta z dzielenia, jeśli występuje.

Przykład

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

int main (void)
{
	div_t str;

	str = div (23, 4);

	printf("%d %d\n", str.quot, str.rem);
	return 0;
}
```

`ldiv_t ldiv (long n, long d)`

Opis

Funkcja analogiczna do funkcji div. Struktura jest analogiczna, tylko pola są typu long int.

Parametry

Dzielna i dzielnik typu long int.

Zwracana wartość

Wartość zwracana jest do struktury. Do pola quot zostaje przypisana wartość dzielenia całkowitego, a do rem reszta z dzielenia, jeśli występuje.

10.15 string.h

W nagłówku string.h znajdują się funkcje wykonujące pewne czynności na tekstach. W poniższej tabeli zestawiono wszystkie te funkcje, a pod tabelą opis wraz z przykładem.

Nazwa	Opis
Kopiowanie	
strcpy	Funkcja kopiująca znaki
strncpy	Funkcja kopiująca konkretną ilość znaków
memcpy	Kopiowanie z jednego do drugiego obiektu
memmove	j/w oraz działa gdy obiekty nachodzą na siebie
Dołączanie	
strcat	Funkcja dopisująca znaki
strncat	Funkcja dopisująca konkretną ilość znaków
Porównywanie	
strcmp	Funkcja porównująca znaki
strncmp	Funkcja porównująca konkretną ilość znaków
memcmp	Porównanie określonej liczby znaków zawartych w dwóch obiektach
Wyszukiwanie	
strpbrk	Wyszukiwanie wielu znaków w ciągu znaków
strstr	Wyszukiwanie słowa w ciągu znaków
strtok	Wyszukiwanie w tekście ciągu znaków przedzielone konkretnymi znakami
memchr	Wyszukiwanie znaku w obiekcie
strchr	Wyszukiwanie znaku w ciągu znaków
strrchr	Wyszukiwanie znaku w ciągu znaków od tyłu
strspn	Obliczanie długości przedrostka
strcspn	Obliczanie długości przedrostka
Inne	
strlen	Długość tekstu
strerror	Wskaźnik do tekstu komunikatu o błędzie
memset	Wstawianie znaku do konkretnej początkowej ilości znaków obiektu

10.15.1 Kopiowanie

```
char *strcpy (char *destination, const char *source)
```

Opis

Funkcja strcpy kopiuje ciąg znaków na który wskazuje source do tablicy znaków wskazywanej przez destination wraz ze znakiem końca. Aby nie pisać po pamięci należy dobrać odpowiedni rozmiar tablicy destination. Lepszym rozwiązaniem jest funkcja strncpy.

Parametry

Pierwszym parametrem jest tablica do której znaki zostaną skopiowane. Drugim parametrem jest ciąg znaków, który ma zostać skopiowany.

Zwracana wartość

Funkcja zwraca wskaźnik do destination.

Przykład

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

int main (void)
{
      char tab[50];
      char *napis = "Ala ma kota";
      char info[] = "Czesc";

      strcpy(tab, napis);
      printf("%s\n", tab);
      strcpy(tab, info);
      printf("%s\n", tab);
      strcpy(tab, "Hello World");
      printf("%s\n", tab);

      return 0;
}
```

```
char *strncpy (char *destination, const char *source, size_t n)
```

Opis

Funkcja strncpy tak samo jak wspomniana powyżej strcpy kopiuje znaki z source do destination, natomiast jako trzeci parametr przyjmuje maksymalną ilość znaków, które są kopiowane. Dzięki temu funkcja ta jest bezpieczniejsza, ponieważ jeśli rozmiar tablicy destination jest za mały by przechować tekst z source to tekst zostanie obcięty pod warunkiem, że ustawimy odpowiedni rozmiar jako n. W przypadku gdy znak końca kopiowanego tekstu zostanie wykryty przed maksymalną ilością kopiowanych znaków, reszta pozycji w tablicy destination zostaje wypełniona zerami. Znak końca nie zostaje dodany automatycznie (jeśli ilość znaków kopiowanych jest większa niż n) do tablicy

destination, więc o tym należy pamiętać.

Parametry

Pierwszym parametrem jest tablica do której znaki zostaną skopiowane. Drugim parametrem jest ciąg znaków, który ma zostać skopiowany, trzecim parametrem jest ilość znaków które mają zostać skopiowane.

Zwracana wartość

Funkcja zwraca wskaźnik do destination.

Przykład

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

int main (void)
{
	char tab[20];
	char *napis = "Ala ma kota, a kot ma 5 lat";
	char info[] = "Czesc";
	int rozmiar = sizeof (tab);

	strncpy(tab, napis, rozmiar-1);
	tab[rozmiar-1] = '\0';
	printf("%s\n", tab);
	strncpy(tab, info, rozmiar);
	printf("%s\n", tab);
	strncpy(tab, "Hello World", rozmiar);
	printf("%s\n", tab);

	return 0;
}
```

```
void *memcpy (void *destination, const void *source, size_t n)
```

Opis

Funkcja memcpy kopiuje pewien blok pamięci w inne miejsce, a mianowicie kopiuje blok wskazywany przez source, do miejsca wskazywanego przez destination. Kopiowana jest n ilość bajtów. Bezpieczniejszą funkcją, która ma analogiczne działanie jest memmove.

Parametry

Pierwszym parametrem jest miejsce docelowe kopiowanego bloku, drugim jest początek bloku kopiowanego, a trzecim ilość kopiowanych bajtów.

Zwracana wartość

Funkcja zwraca wskaźnik do destination.

Przykład

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

void wydrukuj (int *t, int size);
int main (void)
{
	int tab[] = {10, 21, 11, 24, 22, 32, 11, 222, 19};
	int *w;
	w = (int *)malloc (sizeof (tab));

	memcpy (w, tab, sizeof (tab));
	*w = 100;
	wydrukuj (tab, sizeof (tab)/sizeof (tab[0]));
	wydrukuj (w, sizeof (tab)/sizeof (tab[0]));

	printf("%p\n", tab);
	printf("%p\n", w);

	return 0;
}
void wydrukuj (int *t, int size)
{
	int i;
	for (i = 0; i < size; i++)
		printf("%d ", *(t+i));
	printf("\n");
}
```

Omówienie programu

Stworzyliśmy tablicę z zainicjowanymi wartościami całkowitymi. Następnie tworzymy wskaźnik na typ int, aby zarezerwować taką samą ilość miejsca jaką zajmuje tablica tab. Po zarezerwowaniu miejsca, kopiujemy zawartość bloku pamięci zarezerwowanego przez tab (9 elementów cztero bajtowych) do miejsca, które przed chwilą zarezerwowaliśmy. Aby uwidocznić, że są to dwa różne obszary w pamięci przed wydrukowaniem zawartości, zmieniona została wartość zerowego elementu wskazywanego przez w, a następnie wydrukowane zostały wartości z obu obszarów pamięci.

```
void *memmove (void *destination, const void *source, size_t n)
```

Opis

Funkcje jest podobna do memcpy, z tą różnicą, że kopiowanie bloku pamięci odbywa się za pośrednictwem tymczasowego bloku pamięci. Dokładnie odbywa się to w następującej kolejności, kopiowany jest blok wskazywany przez source o rozmiarze n do tymczasowej tablicy o rozmiarze n bajtów, a następnie n bajtów z tymczasowej tablicy kopiowanych jest do miejsca wskazywanego przez destination. Zapobiega to kopiowania nadmiernej ilości danych do miejsca, którego rozmiar jest mniejszy (*ang. overflow*).

Parametry

Pierwszym parametrem jest miejsce docelowe kopiowanego bloku, drugim jest początek bloku kopiowanego, a trzecim ilość kopiowanych bajtów.

Zwracana wartość

Funkcja zwraca wskaźnik do destination.

Przykład

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

int main (void)
{
	char tab[50];
	char *w;
	fgets(tab, sizeof (tab)-1, stdin);
	tab[sizeof (tab)-1] = '\0';

	w =(char *)malloc (sizeof (tab));
	memmove (w, tab, sizeof (tab));
	tab[0] = 'X';
	printf("%s", tab);
	printf("%s", w);
	return 0;
}
```

Omówienie programu

Program jest bardzo podobny do tego z funkcji memcpy. W tym programie wprowadzamy znaki z klawiatury, które następnie kopiujemy do miejsca, które zostało zarezerwowane. Aby pokazać, że są to dwa różne obszary pamięci, w tablicy tab na zerowej pozycji został zmieniony znak na **X**, co przy wydruku dowodzi, że blok pamięci wskazywany przez **w** ma zawartość wpisaną przez użytkownika.

10.15.2 Dołączanie

```
char *strcat (char *destination, const char *source)
```

Opis

Funkcja kopiuje znaki z source do destination, lecz nie wymazuje poprzedniej zawartości tylko dołącza do istniejącego ciągu znaków nowo skopiowane znaki. Podobnie jak w przypadku funkcji strcpy funkcja ta nie sprawdza, czy dołączane znaki zmieszczą się w tablicy destination, co jest nie bezpieczne. Dlatego lepiej stosować funkcję strncat. Znak końca zostaje ustawiony na końcu nowego ciągu znaków.

Parametry

Pierwszym parametrem jest tablica do której znaki zostaną dołączone. Drugim parametrem jest ciąg znaków.

Zwracana wartość

Funkcja zwraca wskaźnik do destination.

Przykład

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

int main (int argc, char *argv[])
{
	char tab[100];

	if (argc != 3)
	{
		fprintf(stderr, "Uzycie: %s arg1 arg2\n", argv[0]);
		return -1;
	}

	strcat (tab, argv[1]);
	strcat (tab, argv[2]);
	printf("%s\n", tab);

	return 0;
}
```

```
char *strncat (char *destination, const char *source, size_t n)
```

Opis

Funkcja robiąca dokładnie to samo co strcat z możliwością ograniczenia kopiowanych znaków. Jeśli

ilość znaków z source jest mniejsza niż rozmiar n, to kopiowane są tylko te znaki wliczając w to znak końca.

Parametry

Pierwszym parametrem jest tablica do której znaki zostaną dołączone. Drugim parametrem jest ciąg znaków. Trzecim parametrem jest ilość dołączanych znaków.

Zwracana wartość

Funkcja zwraca wskaźnik do destination.

Przykład

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

int main (void)
{
      char tab[30] = "Ala ma kota, a ";
      char tab2[] = "kot ma 5 lat xxxxxxxxxxxxxxxx";

      strncat (tab, tab2, 12);
      printf("%s\n", tab);

      return 0;
}
```

10.15.3 Porównywanie

```
int strcmp (const char *str1, const char *str2)
```

Opis

Funkcja porównuje dwa ciągi znaków (str1, str2). Zaczyna od pierwszego znaku w każdym z ciągów znaków i jeśli są one takie same to kontynuuje porównywanie. Funkcja przerywa działanie jeśli porównywane znaki będą się różnić, lub jeden z ciągów się skończy.

Parametry

Pierwszym oraz drugim parametrem jest wskaźnik do ciągu znaków.

Zwracana wartość

Funkcja zwraca wartość całkowitą. Jeśli oba ciągi znaków są takie same wartość zero jest zwracane. Jeśli pierwszy nie pasujący znak ma większą wartość w str1 niż w str2 to wartość dodatnia jest zwracana, w przeciwnym przypadku zwracana jest wartość ujemna.

Przykład 1

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

int main (void)
{
      char *str1 = "aaab";
      char *str2 = "aaaa";

      int w;

      w = strcmp (str1, str2);
      if (w > 0)
            printf("Wyraz str1 wiekszy\n");
      else if (w < 0)
            printf("Wyraz str1 mniejszy\n");
      else
            printf("Wyrazy rowne\n");

      return 0;
}
```

Przykład 2

```
#include <stdio.h>
#include <string.h>
void create (char *filename);

int main (int argc, char *argv[])
{
      if (argc != 3)
      {
            fprintf(stderr, "Uzycie: %s option filename\n", argv[0]);
            fprintf(stderr, "Options: \n-r - remove file\n");
            fprintf(stderr, "-c - create file\n");
            return -1;
      }

      if (!strcmp (argv[1], "-r"))
            if (!remove(argv[2]))
                  fprintf(stderr, "Plik: %s usunieto\n", argv[2]);
            else
                  fprintf(stderr, "Pliku: %s nie udalo sie usunac\n",
argv[2]);

      if (!strcmp (argv[1], "-c"))
            create(argv[2]);
      return 0;
}
```

```
void create (char *filename)
{
      FILE *wp;

      if ((wp = fopen(filename, "w")) != NULL)
      {
            fprintf(stderr, "Plik: %s utworzono\n", filename);
            fclose(wp);
      }
      else
            fprintf(stderr, "Pliku: %s nie udalo sie utworzyc\n",
filename);
}
```

Omówienie programu 1

Jak widać str1 będzie większe, dlatego, że pierwszym różniącym się znakiem jest b, którego wartość numeryczna jest większa od a.

Omówienie programu 2

Ten program z kolei wykorzystuje pewien mechanizm, który w programach Linuksowych jest bardzo często wykorzystywany, a mianowicie opcje podczas uruchamiania programu. Temat ten był poruszony w rozdziale 6 i tam była przedstawiona bardziej uniwersalna metoda, lecz dla prostych programów można zastosować ten sposób. Sprawdzamy czy drugi argument (pamiętamy pierwszy to nazwa programu) to znaki "-r" i jeśli tak jest to podany plik (trzeci argument) zostaje usunięty (oczywiście, jeśli istnieje). Jeśli użylibyśmy opcji "-c" to plik zostanie utworzony.

```
int strncmp (const char *str1, const char *str2, size_t n)
```

Opis

Działanie funkcji strncmp jest identyczne z funkcją strcmp z tą różnicą, że mamy możliwość sprawdzenia pewnej konkretnej ilości początkowych znaków. Funkcja porównuje znaki od pierwszego w każdym z ciągów i kończy działanie w momencie, gdy znaki się różnią, jeden z ciągów się skończy, lub porówna n znaków.

Parametry

Pierwszym oraz drugim parametrem jest wskaźnik do ciągu znaków, trzecim parametrem jest ilość porównywanych znaków.

Zwracana wartość

Funkcja zwraca wartość całkowitą. Jeśli oba ciągi znaków są takie same wartość zero jest zwracana. Jeśli pierwszy nie pasujący znak ma większą wartość w str1 niż w str2 to wartość dodatnia jest zwracana, w przeciwnym przypadku zwracana jest wartość ujemna.

Przykład

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

int main (void)
{
	char *op1 = "aaAaaab";
	char *op2 = "aaAaaab fdfsd";
	int w;

	w = strncmp (op1, op2, 7);
	if (!w)
		printf("7 poczatkowych znakow - takie same\n");

	return 0;
}
```

```
int memcmp (const void *ptr1, const void *ptr2, size_t n)
```

Opis

Funkcja memcmp porównuje początkowych n bajtów bloków pamięci wskazywanych przez ptr1 oraz ptr2. Jeśli blokiem wskazywanym przez oba wskaźniki będzie ciąg znaków, to funkcja zachowuje się analogicznie do funcji strcmp.

Parametry

Paramert pierwszy i drugi to wskaźniki do pewnego bloku pamięci, który ma zostać porównany. Ostatnim parametrem jest ilość porównywanych bajtów.

Zwracana wartość

Funkcja zwraca wartość całkowitą. Jeśli porównywane bloki są takie same, to wartość zero jest zwracana. Jeśli pierwszy różniący się bajt ma większą wartość w ptr1 niż w ptr2 to wartość dodatnia jest zwracana, wartość ujemna zwracana jest w przeciwnym wypadku. Wartości bajtów porównywane są jako unsigned char.

Przykład 1

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

```c
#include <string.h>

int main (void)
{
	char *nap1 = "Ala";
	char *nap2 = "Ola";
	int w;

	w = memcmp (nap1, nap2, 3);

	if (!w)
		printf("Takie same\n");
	else if (w < 0)
		printf("nap1 < nap2\n");
	else
		printf("nap1 > nap2\n");

	return 0;
}
```

Przykład 2

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

int main (void)
{
	int tab1[] = {9, 1, 2, 3, 7, 1};
	int tab2[] = {9, 1, 2, 5, 5, 3};
	int w;
	w = memcmp(tab1, tab2, 12);

	if (!w)
		printf("Takie same\n");
	else if (w < 0)
		printf("tab1 < tab2\n");
	else
		printf("tab1 > tab2\n");

	return 0;
}
```

Omówienie programu 1

Użyta w programie tym funkcja memcmp daje analogiczny skutek jak funkcja strcmp dlatego, że porównywaną ilością bajtów jest liczba 3, co bezpośrednio wiąże się z ilością znaków obu napisów. Każdy bajt jest porównywany i jeśli na którejś pozycji znaki się różnią to odpowiednia wartość jest zwracana.

Omówienie programu 2

Tutaj sprawa wygląda trochę ciekawiej, bowiem porównujemy początkowe trzy elementy tablicy, lecz ilością bajtów nie jest 3 – jakby można było pomyśleć – tylko 12, bo przecież każdy element tablicy typu int zajmuje cztery bajty.

10.15.4 Wyszukiwanie

```
char *strchr (const char *source, int character)
```

Opis

Funkacja przeszukuje źródło (source) w poszukiwaniu znaku (character). Znak końca **\0** również wliczany jest jako część tekstu, tak więc można go wyszukać.

Parametry

Pierwszym parametrem jest wskaźnik do tekstu, drugim parametrem jest wyszukiwany znak.

Zwracana wartość

W przypadku znalezienia znaku wskaźnik do miejsca w którym on występuje jest zwracany. Funkcja zwraca wskaźnik do pierwszego wystąpienia znaku. Jeśli znaku nie udało się znaleźć **NULL** jest zwracane.

Przykład

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

int main (void)
{
      char str[] = "100,00";
      char *w;

      w = strchr (str, ',');
      if (w != NULL)
            *w = '.';
      printf("%s\n", str);
      return 0;
}
```

`char *strrchr (const char *source, int character)`

Opis

Funkcja robiąca dokładnie to samo co strchr z tą różnicą, że wyszukuje znak od końca.

Parametry

Pierwszym parametrem jest wskaźnik do tekstu, drugim parametrem jest wyszukiwany znak.

Zwracana wartość

W przypadku znalezienia znaku wskaźnik do miejsca w którym on występuje jest zwracany. Funkcja zwraca wskaźnik do pierwszego od końca wystąpienia znaku. Jeśli znaku nie udało się znaleźć NULL jest zwracane.

Przykład

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

int main (void)
{
      char tab[] = "100,000,00";
      char *w;
      w = strrchr(tab, ',');
      if (w != NULL)
            *w = '.';
      printf("%s\n", tab);
      return 0;
}
```

`size_t strspn (const char *str1, const char *str2)`

Opis

Funkcja zwraca długość początkowego tekstu z str1, który składa się jedynie ze znaków str2.

Parametry

Pierwszym i drugim parametrem są wskaźniki do tekstu.

Zwracana wartość

Długość początkowego tekstu z str1, który zawiera jedynie znaki z str2. Jeśli w str2 są identyczne znaki jak w str1 to zwrócona wartość jest długością tekstu str1. Jeśli pierwszy znak z str1 nie występuje w str2 to zwracane jest zero.

Przykład

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

```
#include <string.h>

int main (void)
{
	size_t no;
	char *str1 = "Ala ma kota";
	char *str2 = str1;

	no = strspn (str1, "lamA ");
	printf("%d\n", no);
	no = strspn (str1, str2);
	printf("%d\n", no);
	no = strspn (str1, "la ma kota");
	printf("%d\n", no);

	return 0;
}
```

```
size_t strcspn (const char *str1, const char *str2)
```

Opis

Funkcja przesukuje str1 w poszukiwaniu pierwszego wystąpienia jakiegokolwiek znaku z str2.

Parametry

Pierwszym i drugim parametrem są wskaźniki do tekstu.

Zwracana wartość

W przypadku znalezienia w str1 jakiegokolwiek znaku z str2 funkcja zwraca pozycję na której ten znak się znajduje. Jeśli takiego znaku nie znaleziono, to długość str1 jest zwracana.

Przykład

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

int main (void)
{
	char *info = "Dzien dobry";
	size_t no;

	no = strcspn (info, "QXZ");
	printf("%d\n", no);

	no = strcspn (info, "wie");
	printf("%d\n", no);

	return 0;
}
```

Omówienie programu

Pierwsze wywołanie funkcji strcspn zwróci długość ciągu znaków wskazywanego przez info dlatego, że żadnen ze znaków **Q**, **X**, **Z** nie występuje w tym tekście. Drugie wywołanie funkcji zwróci wartość dwa, ponieważ pierwszym wystąpieniem jakiegokolwiek znaku z str2 („wie") jest litera **i**, która jest na pozycji drugiej (licząc od zera).

```
char *strpbrk (const char *str1, const char *str2)
```

Opis

Funkcja zwracająca wskaźnik do pierwszego wystąpienia w str1 jakiegokolwiek znaku z str2.

Parametry

Pierwszym i drugim parametrem jest wskaźnik do tekstu.

Zwracana wartość

Funkcja zwraca wskaźnik do pierwszego wystąpienia jakiegokolwiek znaku z str2 w ciągu znaków wskazywanym przez str1. Jeśli żaden znak nie został znaleziony funkcja zwraca **NULL**.

Przykład

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

int main (void)
{
      char *napis = "Ala ma kota, a kot ma 5 lat";
      char *w;
      char *smg = "euioaEUIOA";
      w = strpbrk(napis, smg);

      printf("W zdaniu: %s. Wystepuja nastepujace samogloski: \n", napis);

      while (w != NULL)
      {
            printf("%c ", *w);
            w = strpbrk(w+1, smg);
      }
      printf("\n");
      return 0;
}
```

Omówienie programu

Wskaźnik smg wskazuje na samogłoski. Do wskaźnika w przypisujemy wywołanie funkcji strpbrk.

Następnie sprawdzamy czy jakakolwiek samogłoska została znaleziona (różne od NULL). Jeśli tak, to drukujemy ją, a następnie przypisujemy do w wywołanie funkcji strpbrk, która w tym miejscu przyjmuje jako pierwszy parametr nie wskaźnik do napis, a do w+1. Chodzi o to, że jeśli w wskazuje na znak z tego ciągu, to można też odwołać się do dalszej części ciągu, aż do znaku **\0**. Dlatego też przesuwamy wskaźnik na następną pozycję i od niej szukamy kolejnych samogłosek.

```
char *strstr (const char *str1, const char *str2)
```

Opis

Funkcja strstr wyszukuje w ciągu wskazywanym przez str1 ciągu (całego) str2. Funkcja jest użyteczna, jeśli wyszukujemy np. wyraz, zdanie.

Parametry

Pierwszym i drugim parametrem są wskaźniki do tekstu.

Zwracana wartość

Funkcja zwraca wskaźnik do pierwszego wystąpienia całego ciągu znaków str2 w str1. Jeśli wyraz (zdanie) z str2 nie występuje w str1 zwracaną wartością jest NULL.

Przykład

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

int main (void)
{
	char *napis = "Siala baba mak, nie wiedziala jak,\na dziad wiedzial, 
nie powiedzial,\ndostal 10 lat\n";
	char *w;

	w = strstr (napis, "nie wiedziala jak");
	if (w != NULL)
		printf("%s", w);
	return 0;
}
```

Omówienie programu

Do w przypisujemy wywołanie funkcji strstr, która szuka fragmentu „nie widziala jak” w tekście wskazywanym przez napis. Jeśli taki ciąg znaków się znajduje, to przypisany zostaje wskaźnik do pierwszego znaku znalezionej frazy. Następnie drukowany jest ten fragment wraz zresztą tekstu.

```
void *memchr (const void *ptr1, int value, size_t n)
```

Opis

Funkcja przeszukuje n początkowych bajtów bloku pamięci wskazywanego przez ptr w poszukiwaniu wartości value (interpretowanej jako unsigned char).

Parametry

Pierwszym parametrem jest wskaźnik do bloku pamięci, drugim jest szukana wartość, a trzecim jest ilość przeszukiwanych bajtów.

Zawracana wartość

Funkcja zwraca wskaźnik do miejsca wystąpienia szukanej wartości w przypadku znalezienia jej, lub **NULL** jeśli takiej wartości nie znalazła.

Przykład

```
#include <stdio.h>
#include <string.h>
void info (int *w, int size, int value);
int main (void)
{
	int tab[] = {0, 1, 2, 4, 8, 16, 32, 64};
	int *w;
	int size, value;

	size = 16;
	value = 8;
	w = (int *)memchr (tab, value, size);
	info (w, size, value);

	size = 20;
	w = (int *)memchr (tab, value, size);
	info (w, size, value);

	printf("%p\n", tab+4);

	return 0;
}
void info (int *w, int size, int value)
{
	if (w == NULL)
	printf("W pierwszych %d bajtach nie ma wartoci: %d\n", size, value);
	else
	{
	printf("W pierwszych %d bajtach jest wartosc: %d\n", size, value);
	printf("Pod adresem: %p\n", w);
	}
}
```

Omówienie programu

Na początku tworzona jest tablica liczb całkowitych z zainicjowanymi wartościami. Do zmiennych pomocniczych przypisujemy wartości, a konkretnie ilość bajtów oraz szukaną wartość, które potrzebne są do funkcji memchr. Wywołanie funkcji przypisujemy do wskaźnika w, oraz rzutujemy wynik na (int *). Następnie wywołujemy funkcję info, która przyjmuje jako argumenty wskaźnik w, **rozmiar** oraz **wartość** w celu ustalenia i powiadomienia użytkownika, czy dana wartość w przeszukiwanej ilości bajtów została znaleziona. Jeśli tak to dodatkowo wyświelany jest adres komórki pamięci, pod którym szukana wartość się znajduje. Jak widać w obszarze 20 bajtów liczba 8 występuje.

```
char *strtok (char *str, const char *delimiters)
```

Opis

Funkcja rozdziela pewne części napisu między którymi występują konkretne znaki (delimiters). Znaki te podaje się jako drugi parametr. Pierwsze wywołanie funkcji wymaga, aby pierwszym parametrem był wskaźnik do tekstu. W kolejnych wywołaniach funkcja wymaga jako pierwszego parametru wartości NULL. Przeszukiwanie ciągu rozpoczyna się od pierwszego znaku wskazywanego przez str, który **nie** występuje w delimiters, a kończy się w momencie napotkanie któregokolwiek znaku zawartego w delimiters. Znak ten zostaje zamieniony na **\0** i przesukiwanie tekstu rozpoczyna się od kolejnego znaku. Przeszukiwanie kończy się w momencie gdy strtok natrafi na znak **\0** – koniec napisu.

Parametry

Pierwsze wywołanie funkcji wymaga, aby pierwszym parametrem była tablica znaków. Kolejne wywołania funkcji wymagają NULL. Drugim parametrem jest wskaźnik do ciągu znaków zawierający konkretne znaki.

Zwracana wartość

Po każdym wywołaniu funkcja zwraca wskaźnik do początku tekstu (od pierwszego znaku nie należącego do delimiters, aż do **\0**). Wartość NULL jest zwracana po wykryciu końca tekstu.

Przykład

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

int main (void)
{
```

```
	char str[] = "Czesc - To - Ja; Artur; Co, slychac?";
	char *delimiters = "-;, ";
	char *w;

	w = strtok (str, delimiters);
	while (w != NULL)
	{
		printf("%s\n", w);
		w = strtok (NULL, delimiters);
	}
	return 0;
}
```

Omówienie programu

Naszym zadaniem jest wyświetlić każdy wyraz ciągu znaków z str w nowej linii. Wyrazy oddzielone są pewnymi znakami (delimiters). Tak więc do rozwiązania tego zadania używamy funkcji strtok. W pierwszym wywołaniu podajemy tablicę znaków, a następnie w pętli drukujemy ciąg znaków od jednego delimitera do drugiego. Następne wywołanie funkcji jak widać jako pierwszy parametr pobiera NULL. Zakończenie pętli następuje w momencie, gdy ciąg znaków zakończy się.

10.15.5 Inne

```
size_t strlen (const char *str)
```

Opis

Funkcja obliczająca długość napisów. Rozpoczyna liczenie od pierwszego znaku, a kończy na znaku kończącym napis **\0**, którego nie wlicza.

Parametry

Parametrem jest wskaźnik do ciągu znaków.

Zwracana wartość

Zwracaną wartością jest ilość znaków.

Przykład

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

int main (void)
{
	char *wsk = "Napis skladajacy sie z kilkunastu znakow";
	char tab[] = "Ciekawe ile ten napis ma znakow";
```

```
	char tab2[40] = "Ile znakow, a jaki rozmiar?";

	printf("strlen (wsk):  %d\n", strlen(wsk));
	printf("strlen (tab):  %d\n", strlen(tab));
	printf("strlen (tab2): %d\n\n", strlen(tab2));

	printf("sizeof (wsk):  %d\n", sizeof (wsk));
	printf("sizeof (tab):  %d\n", sizeof (tab));
	printf("sizeof (tab2): %d\n", sizeof (tab2));
	return 0;
}
```

Pewna uwaga

Dlaczego wartości sizeof (wsk) i strlen (wsk) się różnią? Z bardzo prostej przyczyny. Wskaźnik to zmienna, która wskazuje na inny obszar pamięci, tak więc jej rozmiar to rozmiar zmiennej. Dlaczego strlen (tab) to 31, a sizeof (tab) to 32? Ponieważ funkcja strlen nie zlicza znaku **\0**. Jeśli mówimy o rozmiarze, to ten znak jest częścią tablicy, dlatego też jest uwzględniany. Dlaczego strlen (tab2) i sizeof (tab2) się różnią? Rozmiar jest z góry określony, a znaków jest po prostu mniej.

```
void *memset (void *ptr, int value, size_t n)
```

Opis

Funkcja ustawia na pierwszy n bajtach wskazywanych przez ptr wartość podaną jako drugi parametr.

Parametry

Pierwszym parametrem jest wskaźnik do bloku pamięci, drugim parametrem jest ustawiana wartość, a ostatnim ilość bajtów do zamiany.

Zwracana wartość

Funkcja zwraca ptr.

Przykład 1

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

int main (void)
{
	char tablica[] = "Pewien ciag znakow, ktory zostanie zakryty";

	printf("%s\n", tablica);
	memset(tablica, '.', 11);
	printf("%s\n", tablica);
```

```
	return 0;
}
```

Przykład 2

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

int main (void)
{
	int rozmiar = 200;
	char *x;

	x = (char *)malloc(rozmiar);
	x = memset(x, '-', rozmiar);

	printf("%s\n", x);
	printf("%c\n", *(x+10));
	return 0;
}
```

Omówienie programu 1

Jedenaście początkowych bajtów tablicy znaków zostanie zmienionych za pomocą funkcji memset. Na tych pozycjach funkcja ta ustawia znak podany jako drugi argument, czyli po wydrukowaniu zobaczymy na początku jedenaście kropek.

Omówienie programu 2

W tym programie alokujemy pamięć, a konkretnie 200 bajtów, na których potem ustawiamy znaki '-'. Aby odwołać się do pojedynczego znaku, można zrobić to tak jak w ostatniej instrukcji printf, gdzie wyciągamy wartość, która kryje się na 10 pozycji względem punku bazowego zarezerwowanej pamięci.

10.16 time.h

Nagłówek time.h zawiera wszystkie informacje potrzebne do pobierania daty oraz wyświetlania informacji o czasie. Zestawienie zawartości omawianego nagłówka znajduje się w poniższej tabeli.

Nazwa	Opis
Manipulacja czasem	
clock	Funkcja clock

difftime	Zwracanie różnicy pomiędzy czasem
mktime	Konwersja struktury tm do time_t
time	Pobieranie aktualnej godziny
Konwersje	
asctime	Konwertowanie struktury tm do ciągu znaków
ctime	Konwertowanie wartości time_t do ciągu znaków
gmtime	Konwertowanie time_t do tm jako czas UTC
localtime	Konwertowanie time_t do tm jako czas lokalny
strftime	Formatowanie czasu do ciągu znaków
MAKRA	
CLOCKS_PER_SEC	Taktowanie zegara na sekundy
NULL	Wskaźnik NULL
TYPY DANYCH	
clock_t	Typ clock
size_t	Dodatnia liczba całkowita
time_t	Typ time
struct tm	Struktura tm

Tabela 10.16.1 Zestawienie nagłówka time.h

10.16.1 Manipulacja czasem

```
clock_t clock (void)
```

Opis

Funkcja clock pobiera czas procesora, który był potrzebny do wykonania określonego zadania. Aby sprawdzić ile wykonywał się np. jakiś algorytm sortowania możemy użyć tej właśnie funkcji. Funkcję clock umieszczamy przed oraz po operacji, którą chcemy zmierzyć. Różnica tych dwóch wartości podzielona przez stałą **CLOCKS_PER_SEC** daje ilość sekund.

Parametry

Brak

Zwracana wartość

W przypadku nie powodzenia funkcja zwraca -1. Jeśli błędy nie wystąpiły to zostaje zwrócona liczba

taktów zegara.

Przykład

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

void bubblesort (int tab[], int rozmiar);
void losowo (int tab[], int rozmiar);
void wyswietl (int tab[], int rozmiar);

int main (void)
{
	clock_t poczatek, koniec;
	double sekundy;
	int i;
	int tab[10000];
	int rozmiar = sizeof (tab) / sizeof (tab[0]);

	losowo(tab, rozmiar);
	wyswietl(tab, rozmiar);

	poczatek = clock();
	bubblesort(tab, rozmiar);
	koniec = clock();
	wyswietl(tab, rozmiar);

	sekundy = ((double) (koniec - poczatek)) / CLOCKS_PER_SEC;

	printf("Sortowanie tablicy o ilosci elementow: %d zajelo: %lf\n", 
rozmiar, sekundy);

	return 0;
}

void bubblesort (int tab[], int rozmiar)
{
	int i, j, tmp;

	for (i = 0; i < rozmiar-1; i++)
		for (j = 0; j < rozmiar-1-i; j++)
			if (tab[j+1] < tab[j])
			{
				tmp = tab[j];
				tab[j] = tab[j+1];
				tab[j+1] = tmp;
			}
}

void losowo (int tab[], int rozmiar)
{
```

```
	int i;
	srand(time(NULL));

	for (i = 0; i < rozmiar; i++)
		tab[i] = rand() % 1000;

}

void wyswietl (int tab[], int rozmiar)
{
	int i;
	for (i = 0; i < rozmiar; i++)
		printf("tab[%d] = %d\n", i, tab[i]);
}
```

Omówienie programu

W tym miejscu nie będę tłumaczyć jak działa algorytm bąbelkowy (bo taki został użyty) powiem jedynie o sposobie wyliczania czasu. A mianowicie przed wywołaniem funkcji bubblesort do zmiennej typu clock_t (poczatek) przypisujemy wywołanie funkcji clock, tuż po bubblesort wywołujemy tę funkcję po raz drugi przypisując ją do innej zmiennej. Różnica pomiędzy końcową, a początkową wartością podzielona przez **CLOCKS_PER_SEC** daje ilość sekund. Aby wynik był liczbą rzeczywistą należy zrzutować ją na typ rzeczywisty, co zostało uczynione.

```
double difftime (time_t time2, time_t time1)
```

Opis

Funkcja obliczająca różnicę w sekundach pomiędzy time1, a time2.

Parametry

Pierwszym parametrem jest zmienna przechowywująca późniejszy z dwóch czasów, drugim parametrem jest czas wcześniejszy.

Zwracana wartość

Funkcja zwraca różnicę czasów (time2 – time1) jako liczbę rzeczywistą.

Przykład

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

int main (void)
{
```

```
	time_t time1, time2;
	double wynik;
	char name[50];

	time (&time1);
	printf("Wpisz swoje imie: ");
	fgets(name, sizeof (name), stdin);
	time (&time2);
	wynik = difftime (time2, time1);

	printf("\nWitaj, %sWpisanie imienia zajelo Ci zaledwie: %.2lfs\n", 
name, wynik);
	return 0;
}
```

time_t mktime (struct tm *timeptr)

Opis

Funkcja konwertująca strukturę tm do obiektu typu time_t.

Parametry

Parametrem jest wskaźnik do struktury tm.

Zwracana wartość

Funkcja zwraca ilość sekund, które upłynęły od pewnego dnia roku jako obiekt typu time_t. Ten dzień jest sprecyzowany w strukturze wskazywanej przez timeptr. Jeśli wystąpił błąd i data nie może zostać przedstawiona – wartość -1 jest zwracana.

Przykład

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

void uzupelnijReszte (struct tm *tim);
void pobierzLiczbe (char *opis, char *buf, int *l);
int main (void)
{
	struct tm tim;
	char buffor[25];

	pobierzLiczbe("Dzien", buffor, &tim.tm_mday);
	pobierzLiczbe("Miesiac", buffor, &tim.tm_mon);
	pobierzLiczbe("Rok", buffor, &tim.tm_year);
	tim.tm_year -= 1900;
	tim.tm_mon -= 1;
	uzupelnijReszte(&tim);
```

```
	if (mktime(&tim) == -1)
		fprintf(stderr, "Bledna data\n");
	else
	{
		strftime(buffor, sizeof (buffor), "%A", &tim);
		printf("%s\n", buffor);
	}
	return 0;
}
void pobierzLiczbe (char *opis, char *buf, int *l)
{
	printf("%s: ", opis);
	fgets(buf, 5, stdin);
	*l = atoi(buf);
}

void uzupelnijReszte (struct tm *tim)
{
	tim->tm_hour = 0;
	tim->tm_min = 0;
	tim->tm_sec = 1;
	tim->tm_isdst = -1;
}
```

Omówienie programu

W programie posiadamy trzy funkcję. Funkcja pobierzLiczbe odpowiada za pobranie dnia, miesiąca oraz roku. Pierwszym parametrem jest ciąg znaków, który zostaje wyświetlony podczas wprowadzania danych. Drugim parametrem jest tablica znaków, do której będziemy zapisywać te liczby, natomiast trzecim jest wskaźnik do struktury tm. Po trzykrotnym wywołaniu funkcji, od liczby lat wprowadzonej przez użytkownika odejmujemy 1900, a od liczby miesiąca odejmujemy 1 (aby dowiedzieć się czemu tak zobacz strukturę tm w punkcie 10.16.4 – liczba lat liczona jest od 1900 roku, liczba miesięcy jest liczona od 0). Funkcja uzupelnijReszte uzupełnia pozostałe pola struktury tm, jest to istotne, ponieważ mktime oblicza ilość sekund od tej daty, dlatego wszystkie pola muszą być wypełnione. mktime zwróci -1 jeśli wpisane przez użytkownika wartości nie będą mogły zostać zamienione na odpowiednią datę. Jeśli wszystko poszło zgodnie z oczekiwaniami, to zmienna typu strukturowego tim została uzupełniona o konkretną datę, a następnie za pomocą funkcji strftime przekształcona do czytelnej wersji (tutaj po prostu nazwa dnia – zobacz tabelę w funkcji strftime).

```
time_t time (time_t *timer)
```

Opis

Funkcja pobiera informacje o czasie, a konkretnie o ilości sekund, która upłynęła od 1 stycznia 1970 roku. Jeśli jako parametr podamy NULL, to po prostu zostanie zwrócona wartość sekund. Jeśli jako argument podamy adres do zmiennej, to również wartość sekund zostanie zwrócona i jednocześnie przypisana do tej zmiennej, dzięki czemu w difftime, mogliśmy obliczyć różnicę.

Parametry

NULL, lub adres zmiennej typu time_t.

Zwracana wartość

Ilość sekund, która upłynęła od 1 stycznia 1970 roku od godziny 00:00.

Przykład

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

int main (void)
{
      time_t sekundy = time(NULL);
      double minuty = sekundy / 60.0;
      double godziny = minuty / 60;
      double doby = godziny / 24;
      double lata = doby / 365;

      printf("%ld tyle sekund uplynelo od 01/01/1970\n", sekundy);
      printf("%.2lf tyle minut\n", minuty);
      printf("%.2lf tyle godzin\n", godziny);
      printf("%.2lf tyle dob\n", doby);
      printf("%.2lf tyle lat\n", lata);

      return 0;
}
```

10.16.2 Konwersje

```
char *asctime (const struct tm *tmptr)
```

Opis

Funkcja konwertująca zawartość struktury tm, wskazywanej przez tmptr do wersji „dla ludzi”, czyli takiej bardziej zrozumiałej.

Parametry

Parametrem jest wskaźnik do struktury tm.

Zwracana wartość

Funkcja zwraca w postaci ciągu znaków datę wraz z godziną w formacie zrozumiałym dla ludzi. A mianowicie:

```
DTG MSC DD GG:MM:SS RRRR
```

DTG – dzień tygodnia, MSC – miesiąc, DD – dzień, GG – godzina, MM – minuta, SS – sekunda, RRRR – rok.

Przykład

Przykład znajduje się przy opisie funkcji localtime oraz gmtime.

```
char *ctime (const time_t *timer)
```

Opis

Funkcja jest bardzo podobna do funkcji asctime. Konwertuje obiekt typu time_t wskazywany przez timer do czytelnej postacji. Zwracana wartość jest w takim samym formacie co asctime.

Parametry

Parametrem jest wskaźnik do obiektu typu time_t.

Zwracana wartość

Funkcja zwraca datę jako ciąg znaków w postaci identycznej jak asctime.

Przykład

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

int main (void)
{
      time_t tNow;
      time (&tNow);
      char *w;
      w = ctime (&tNow);

      printf("%s", w);
      return 0;
}
```

`struct tm *gmtime (const time_t *timer)`

Opis

Funkcja konwertująca obiekt typu time_t do struktury tm jako czas UTC (GMT timezone).

Parametry

Wskaźnik do obiektu typu time_t.

Zwracana wartość

Funkcja zwraca wskaźnik do struktury tm.

Przykład

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

int main (void)
{
      time_t tNow;
      struct tm *tInfo;

      time (&tNow);
      tInfo = gmtime (&tNow);
      printf("%s", asctime(tInfo));

      return 0;
}
```

`struct tm *localtime (const time_t *timer)`

Opis

Funkcja konwertująca obiekt typu time_t do struktury tm jako czas lokalny.

Parametry

Wskaźnik do obiektu typu time_t.

Zwracana wartość

Funkcja zwraca wskaźnik do struktury tm.

Przykład

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

int main (void)
{
      time_t tNow;
      struct tm *tInfo;
```

```
    time (&tNow);
    tInfo = localtime(&tNow);
    printf("%s", asctime(tInfo));
    return 0;
}
```

```
size_t strftime (char *ptr, size_t max, const char *format, const struct tm *tmptr)
```

Opis

Funkcja kopiuje do miejsca wskazywanego przez ptr zawartość wskazywaną przez format, w której mogą wystąpić przekształcenia, które uzupełniane są za pomocą tmptr. max ogranicza ilość kopiowanych znaków.

Parametry

Pierwszy parametr to wskaźnik do tablicy, drugim parametrem jest ilość kopiowanych znaków, trzeci to format w jakim zostaną zapisane dane w tablicy, a czwarty to wskaźnik do struktury tm. Poniżej znajduje się tabela, w której znajdują się przekształtniki używane w tekście wskazywanym przez format.

Przekształtnik	Oznaczenie	Przykład
%a	Skrócona nazwa dnia tygodnia *	Sat
%A	Pełna nazwa dnia tygodnia *	Saturday
%b	Skrócona nazwa miesiąca *	Oct
%B	Pełna nazwa miesiąca *	October
%c	Reprezentacja daty i czasu *	Sat Oct 23 23:02:29 2010
%d	Dzień miesiąca (01 – 31)	23
%H	Godzina – format 24h	18
%I	Godzina – format 12h	8
%j	Dzień roku (001 – 366)	231
%m	Numer miesiąca (01 – 12)	4
%M	Minuta (00 – 59)	45
%p	AM lub PM	AM
%S	Sekundy (00 – 59)	32
%U	Numer tygodnia – Niedziela pierwszym dniem tygodnia (00 – 53)	52

%w	Numer dnia tygodnia (0 – 6) 0 – Niedziela	3
%W	Numer tygodnia – Poniedziałek pierwszym dniem tygodnia (00 – 53)	32
%x	Reprezentacja daty *	10/23/10
%X	Reprezentacja czasu *	23:21:59
%y	Rok – ostatnie dwie cyfry	99
%Y	Rok – pełna reprezentacja roku	1999
%Z	Nazwa strefy czasowej, lub skrót	CEST
%%	Znak %	%

Pozycje oznaczone gwiazdką (*) zależą od lokalnych ustawień.

Zwracana wartość

Jeśli wszystkie znaki wliczając w to **\0** z format zostały skopiowane do ptr, to ilość skopiowanych znaków (bez znaku **\0**) zostaje zwrócona. W przeciwnym wypadku zero jest zwracane.

Przykład

```
#include <stdio.h>
#include <time.h>
#define MAX 1000

int main (void)
{
      char buffor[MAX];
      char *form = " %a\n %A\n %b\n %B\n %c\n %d\n %Z\n %W\n %X\n";
      time_t timeNow;
      struct tm *timeInfo;

      time (&timeNow);
      timeInfo = localtime (&timeNow);

      strftime (buffor, sizeof (buffor), form, timeInfo);
      printf("%s\n", buffor);
      return 0;
}
```

10.16.3 Makra

CLOCKS_PER_SEC

Makro **CLOCKS_PER_SEC** odpowiada za reprezentację ilości taktów zegara na sekundę. Dzieląc wartość uzyskaną za pomocą funkcji clock, przez to makro uzyskamy ilość sekund. Przykład znajduje

się w punkcie 10.16.1.

`NULL`

Makro NULL jest z reguły używane w celu oznaczenia, że wskaźnik nie wskazuje na żaden obiekt.

10.16.4 Typy danych

`clock_t`

Typ danych zdolny przechowywać ilość taktów zegara oraz wspiera operacje arytmetyczne. Taki typ danych zwracany jest przez funkcję clock.

`size_t`

Typ danych odpowiadający dodatniemu całkowitemu typowi danych. Operator sizeof zwraca dane tego typu.

`time_t`

Typ danych zdolny przechowywać czas, oraz umożliwia wykonywanie operacji arytmetycznych. Ten typ danych zwracany jest przez funkcję time oraz używany jako parametr nie których funkcji z nagłówka time.h.

`struct tm`

Struktura zawierająca kalendarz oraz datę podzieloną na części na poszczególne pola. W strukturze tej znajduje się dziewięć pól typu int.

Nazwa	Znaczenie	Zasięg
tm_sec	Sekundy (po minucie)	0 – 59
tm_min	Minuty (po godzinie)	0 – 59
tm_hour	Godziny liczone od północy	0 – 23
tm_mday	Dzień miesiąca	1 – 31
tm_mon	Miesiąc (liczony od stycznia)	0 – 11
tm_year	Rok (liczony od 1900)	-
tm_wday	Dzien tygodnia (liczony od niedzieli)	0 – 6
tm_yday	Dzień roku (liczony od 1 stycznia)	0 – 365
tm_isdst	Czy jest dzień?	1 lub 0

Tabela 10.16.4.1 Struktura tm

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

int main (void)
{
	time_t tNow;
	struct tm *tInfo;
	time (&tNow);

	tInfo = localtime (&tNow);

	printf("%d:%d:%d\n", tInfo->tm_hour, tInfo->tm_min, tInfo->tm_sec);
	printf("%d/%d/%d\n", tInfo->tm_mday, tInfo->tm_mon, tInfo->tm_year
+1900);
	printf("%d %d %d\n", tInfo->tm_wday, tInfo->tm_yday,tInfo->tm_isdst);
	return 0;
}
```

Omówienie programu

Najpierw definiujemy zmienną tNow, w której będziemy przechowywać ilość sekund, które upłyneły od 01/01/1970. Następnie tworzymy wskaźnik do struktury tm tInfo. Za pomocą funkcji time uzupełniamy zmienną tNow o te sekundy oraz do tInfo przypisujemy wywołanie localtime z konkretną ilością sekund. Od teraz możemy odwoływać się do wszystkich pól struktury. Ciekawostka może być przy wyświetlaniu roku, ponieważ liczba wyświetlona liczona jest od 1900 roku, tak więc dostalibyśmy wartość 110, co aktualnym rokiem nie jest.

11 MySQL – Integracja programu z bazą danych

Aby mieć możliwość połączenia programu z bazą danych MySQL musimy użyć specjalnej biblioteki mysql.h. Problem polega na tym, że trzeba mieć jeszcze zainstalowaną bazę danych MySQL. W dodatku C opisane jest jak się z tym uporać, przedstawiony jest zestaw podstawowych poleceń SQL oraz przykład bazy danych, który może pomóc w zrozumieniu niniejszego listingu. W tym miejscu przedstawiony jest przykład wraz z opisem za co odpowiadają poszczególne części programu.

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

void usage (char *filename, int n);
void initDatabase (void);
void showData (void);

char *servername, *username, *password, *database;
MYSQL *connection;
MYSQL_RES *result;
MYSQL_ROW row;

int main (int argc, char *argv[])
{
	char *filename = argv[0];
	usage(filename, argc);

	servername = argv[1];
	username = argv[2];
	password = argv[3];
	database = argv[4];

	initDatabase();		// Nawiazanie polaczenia

	showData();
	return 0;
}

void usage (char *filename, int n)
{
	if (n != 5)
	{
		fprintf(stderr, "Uzycie: %s servername username password
database\n", filename);
		exit (-1);
	}
}

void initDatabase (void)
```

```
{
	connection = mysql_init(NULL);

	if (!mysql_real_connect(connection, servername, username, password,
database, 0, NULL, 0))
	{
		fprintf(stderr, "%s\n", mysql_error(connection));
		exit (-1);
	}
}

void showData (void)
{
	char command[250];
	int i;

	snprintf(command, sizeof (command), "SELECT * FROM `%s`.`tallest`;",
database);

	if (mysql_query(connection, command))
	{
		fprintf(stderr, "%s\n", mysql_error(connection));
		exit (-1);
	}
	result = mysql_use_result(connection);

	while ((row = mysql_fetch_row(result)) != NULL)
	{
		for (i = 0; i < 7; i++)
			printf("%s ", row[i]);
		printf("\n");
	}
	mysql_free_result(result);
	mysql_close(connection);
}
```

Listing 11.1 Użycie C i MySQL

Na pierwszy rzut oka program może i zajmuje więcej miejsca niż można było się spodziewać, lecz w gruncie rzeczy nie jest to nic nowego. Jedyną nową rzeczą są funkcje MySQL i sposób ich użycia. Reszta została przedstawiona w poprzednich rozdziałach, nie mniej jednak opiszę działanie programu krok po kroku, lecz najpierw sposób kompilacji i uruchomienia naszego programu:

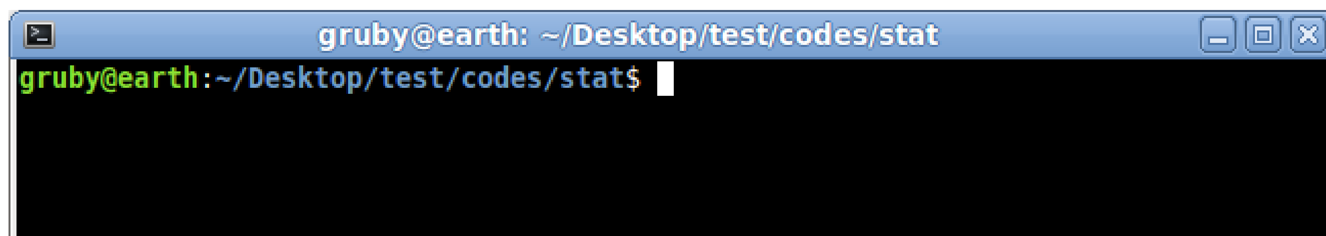
```
gcc main.c -o main $(mysql_config --cflags) $(mysql_config --libs)
```

```
./main localhost root TUTAJ_WPISZ_SWOJE_HASLO building
```

Tworzymy zmienne globalne, a właściwie globalne wskaźniki do typów znakowych po to, aby nie przekazywać do każdej funkcji wykonującej operacje na bazie danych wszystkich potrzebnych argumentów. Na początku funkcji main tworzymy wskaźnik filename który będzie wskazywał na nazwę programu. Do funkcji usage przekazujemy jako pierwszy parametr nazwę programu, a jako drugi ilość argumentów, jeśli jest ona różna od pięciu to za pomocą funkcji exit kończymy działanie programu (małe przypomnienie, jeśli użylibyśmy w funkcji usage instrukcji return, to sterowanie programu wróciłoby do funkcji main, a to byłby błąd, dlatego użyto funkcji exit która kończy działanie całego programu). W kolejnych czterech liniach przypisujemy do globalnych wskaźników argumenty, które zostały otrzymane wraz z wywołaniem programu. Funkcja initDatabase nawiązuje połaczenie z bazą danych. Funkcje **mysql** zawarte w tej funkcji są nowe, dlatego warto je omówić. Do wskaźnika na typ MYSQL o nazwie connection, który utworzony został globalnie przypisujemy wywołanie funkcji mysql_init, która nawiązuje połączenie. Jeśli funkcja mysql_real_connect zwróci wartość zero, to znaczy, że połączenie nie udało się i program przerywa działanie z zastosowaniem odpowiedniego powiadomienia (mysql_error). Wyświetlanie informacji jest najciekawszą rzeczą. Przede wszystkim, funkcja mysql_query przyjmuje jako pierwszy argument wskaźnik do połączenia (connection), a jako drugi polecenie SQL. Zastosowano tutaj pewną sztuczkę z użyciem funkcji snprintf, za pomocą której przypisujemy do tablicy pewien ciąg znaków wraz ze zmiennymi (tutaj nazwa bazy danych). Drugim argumentem funkcji mysql_query jest więc ta tablica, w której znajduje się polecenie SQL. Jeśli funkcja zwróci wartość nie zerową, to program kończy działanie ze stosownym komunikatem. Jeśli wszystko poszło zgodnie z oczekiwaniami to do zmiennej result przypisywany jest wynik ostatniego wykonanego polecenia. Zmienną row można traktować jako tablicę, do której przypisujemy cały rekord, a do poszczególnych kolumn odwołujemy się jak do elementów tablicy, czyli indeksując je od zera. Ponieważ nasz przykład bazy danych zawiera siedem kolumn, tak więc drukujemy wszystkie siedem „elementów” tablicy row. Na koniec czyścimy zmienną result oraz zamykamy połączenie z bazą.

Dodatek A

W tym miejscu chciałbym przedstawić podstawowe polecenia Linuksa potrzebne do dostania się do katalogu, w którym mamy kody źródłowe (bo o to nam chodzi), skompilowanie oraz uruchomienie programu. A więc zacznijmy od włączenia terminala, którego wygląd prezentuje się następująco (rys A.1).



Rys. A.1 Wygląd terminala

Tak po krótce można omówić następujące części terminala, a więc tak:

- gruby – nazwa użytkownika
- earth – nazwa hosta
- ~/Desktop/test/codes/stat – katalog, w którym obecnie się znajdujemy
- $ – znak zachęty (do wprowadzania poleceń)

W poniższej tabeli znajdują się podstawowe (potrzebne nam) polecenia systemu Linux.

Nazwa polecenia	Opis
cd	Zmiana katalogu
mkdir	Tworzenie katalogu
rm	Usuwanie plików / katalogów
ls	Wyświetlanie zawartości folderu
gcc	Kompilowanie programów
chmod	Ustawiania uprawnień
chown	Ustawianie właściciela

Tabela A.1 Podstawowe polecenia Linuksa potrzebne do kompilacji i uruchomienia.

Pomimo iż do samego procesu kompilacji potrzebne jest jedno polecenie, to warto wiedzieć też o tych innych podstawowych poleceniach, które mogą ułatwić nam pracę z konsolą. Programy napisane w tej książce przeznaczone są do uruchamiana z terminala.

A.1 Zmiana katalogu

Aby zmienić katalog w terminalu, należy użyć polecenia cd. Polecenie to przyjmuje jako argument ścieżkę do katalogu bezpośrednią, lub pośrednią (liczoną od miejsca, w którym się znajdujemy). Na poniższym przykładzie zademonstruje jak dostać się na pulpit. Znaku $ nie wpisujemy – oznacza on znak zachęty – wpisujemy polecenie stojące za znakiem dolara.

```
$ cd /home/gruby/Desktop/
```

Gdzie zamiast **gruby** wpiszesz nazwę swojego użytkownika. Jeśli katalogu **Desktop** nie ma, to może być pod polską nazwą – **Pulpit**. Drugą, alternatywną metodą jest wpisanie polecenia w następującej formie.

```
$ cd ~/Desktop/
```

Przyjmijmy następujące założenia. Znajdujemy się na pulpicie, a na pulpicie jest katalog o nazwie **codes**. Aby do niego wejść możemy wpisać bezpośrednią ścieżkę, tak jak pokazano poniżej.

```
$ cd /home/gruby/Desktop/codes
```

Bądź wpisać ścieżkę pośrednią, tak jak pokazano poniżej.

```
$ cd codes
```

Przechodzenie do katalogu nadrzędnego odbywa się w następujący sposób.

```
$ cd ..
```

A.2 Tworzenie katalogu

Aby utworzyć katalog – używamy polecenia mkdir. Przykład poniższy tworzy katalog o nazwie **c_codes** w katalogu, w którym aktualnie się znajdujemy.

```
$ mkdir c_codes
```

A.3 Usuwanie plików i katalogów

Usuwanie plików polega na prostym wydaniu polecenia rm. Usuwanie pustego katalogu można zrealizować za pomocą rmdir, natomiast katalogu z zawartością rm -r. Usunięcie pliku **main.c** (jeśli plik występuje w katalogu, w którym jesteśmy) wygląda następująco.

```
$ rm main.c
```

Usuwanie katalogu z zawartością

```
$ rm -r c_codes
```

A.4 Wyświetlanie zawartości katalogu

Wyświetlanie zawartości katalogu jest bardzo często używane, tak więc pokażę jak się to robi, oraz opiszę pewne informację, które wyświetli terminal. Za pomocą polecenia ls, bez dodatkowych opcji dostaniemy wydruk zawartości bez dodatkowych informacji o plikach / katalogach. Załóżmy, że wyświetlamy zawartość katalogu **c_codes**, w którym znajdują się pliki z rozszerzeniem .c.

```
$ ls c_codes
```

Terminal wydrukuje

```
plik1.c    plik2.c    plik3.c    plik4.c
plik5.c    plik5.c    plik5.c    plik6.c
```

Ważniejszą opcją tego polecenia jest opcja **-l**, za pomocą której możemy dowiedzieć się więcej o pliku. Poniżej wyświetlone zostały informacje o pliku **plik1.c**. Polecenie ma się następująco.

```
$ ls -l plik1.c
```

Terminal wydrukuje

```
-rw-r--r--  1  gruby  gruby  0  2010-07-13 22:56  plik1.c
```

Nie będę opisywał wszystkich kolumn, opiszę tylko te, które są nam potrzebne. A mianowicie pierwszą kolumnę, w której są znaki: **-rw-r--r--**. Które oznaczają, że właściciel pliku może odczytywać (read), i zapisywać (write) plik. Grupa i pozostali mogą tylko odczytywać plik. Ma to istotne znaczenie, gdy chcemy uruchomić plik, który nie ma uprawnień do uruchamiania **-x**, taka operacja jest nie możliwa. W punkcie A.6 jest pokazane jak zmienić uprawnienia. Trzecia i czwarta kolumna oznaczają odpowiednio właściciela oraz grupę pliku.

A.5 Kompilacja programów

Kompilowanie plików z kodami źródłowymi odbywa się w następujący sposób. Wpisujemy w terminalu jedno z poniższych poleceń.

```
$ cc nazwa_pliku.c
```

Jeśli kompilator nie wykryje żadnych błędów to kompilacja pójdzie bezproblemowo i naszym plikiem wynikowym będzie plik o nazwie: **a.out**. Plik taki uruchamiamy za pomocą polecenia:

```
$ ./a.out
```

Program można skompilować inaczej, tzn nadać konkretną nazwę pliku wynikowego za pomocą polecenia:

```
$ gcc nazwa_pliku.c -o nazwa_pliku_wynikowego
```

Jeśli nasz program składa się z większej ilości plików źródłowych to wpisujemy je w następujący sposób:

```
$ gcc plik1.c plik2.c plik3.c -o nazwa_pliku_wynikowego
```

Pliki skompilowane z użyciem polecenia gcc uruchamia się analogicznie, do tego pokazanego wcześniej z tą różnicą, że wpisujemy nazwę, którą sami ustaliliśmy, tzn:

```
$ ./nazwa_pliku_wynikowego
```

A.6 Ustawianie uprawnień

Aby ustawić uprawnienia do pliku należy użyć polecenia chmod. Aby dodać możliwość wykonywania pliku (dla wszystkich) trzeba użyć opcji **+x**. Jeśli chcemy odebrać te uprawnienia to wstawiamy **-x**. Analogicznie jest z czytaniem pliku (+**r** / **-r**), oraz zapisywaniem pliku (+**w** / **-w**). W szczegóły polecenia chmod w tym miejscu też nie będę się wdawał. Chodzi o to, żeby mieć z grubsza pogląd dlaczego pliku nie możemy uruchomić i jak to naprawić. Aby ustawić możliwość wykonywania pliku wpisujemy:

```
$ chmod +x nazwa_pliku_wykonywalnego
```

Polecenie to możemy wykonać jeśli jesteśmy właścicielami pliku, bądź mamy uprawnienia administratora.

A.7 Ustawianie właściciela

Zmiana właściciela pliku odbywa się przy pomocy polecenia chown. Aby zmienić właściciela pliku **statystyki** należy wpisać polecenie w ten sposób.

```
$ chown gruby statystyki
```

Gdzie zamiast **gruby** wpisujesz nazwę użytkownika, który ma stać się właścicielem pliku. Również w tym poleceniu należy posiadać odpowiednie uprawnienia do zmiany właściciela pliku (albo zmieniany plik należy do nas, albo mamy uprawnienia administratora).

Dodatek B

W dodatku tym chcę omówić Linuksowe polecenie time, które sprawdza jak długo wykonywał się program oraz podaje informacje o użytych zasobach.

B.1 Powłoka systemowa

W gruncie rzeczy wpisanie nazwy programu w konsoli powinno uruchomić ten program i zazwyczaj tak jest. Cały dowcip polega na tym, że jeśli używamy powłoki bash, która jest obecnie najpopularniejszą powłoką to wpisanie polecenie time --version wyświetli:

```
--version: command not found

real  0m0.246s
user  0m0.168s
sys   0m0.064s
```

Dzieje się tak ponieważ powłoka bash ma wbudowaną funkcję sprawdzającą czas wykonywania programów, ale jest ona uboga – ogranicza się tylko do tych trzech linijek, które widać powyżej.

Problem ten nie występuje w powłokach sh, ksh. W powłoce csh również występuje ten problem tylko trochę inny komunikat zostaje wyświetlony.

We wszystkich powłokach działa natomiast bezpośrednie odwołanie się do programu. Wpisując w konsoli poniższe polecenie otrzymamy wersję programu GNU time.

```
$ /usr/bin/time --version
```

B.2 Polecenie time, formatowanie wyników

Za pomocą polecenia time możemy dostać bardzo wiele użytecznych informacji o używanych zasobach po skończeniu działania programu. Najprostszą formą użycia wbudowanego programu time w powłokę bash jest wykonanie polecenia:

```
$ time ./nazwa_programu
```

Wynik powinien być następujący

```
real  0m2.539s
user  0m2.448s
sys   0m0.012s
```

Natomiast jest to wbudowany program w powłokę, tak więc jego możliwości ograniczają się tylko do tych, które zostały pokazane powyżej. Aby uzyskać dostęp do wszystkich opcji oraz do możliwości formatowania drukowanego tekstu, do programu time musimy odwołać się w następujący sposób.

```
$ /usr/bin/time ./nazwa_programu
```

Podręcznik systemowy[1] opisuje dość dokładnie co jaki format robi, tak więc aby za dużo się nie rozpisywać pokażę jak się używa tego polecenia z opcjami oraz jak się formatuje drukowany tekst.

Aby uzyskać większość informacji jakie program time jest w stanie wyświetlić należy użyć opcji --verbose (-v). Wpisanie w konsoli:

```
$ /usr/bin/time --verbose ./shell
```

Wyświetli takie o to informacje:

```
Command being timed: "./shell"
      User time (seconds): 2.45
      System time (seconds): 0.01
      Percent of CPU this job got: 97%
      Elapsed (wall clock) time (h:mm:ss or m:ss): 0:02.53
      Average shared text size (kbytes): 0
      Average unshared data size (kbytes): 0
      Average stack size (kbytes): 0
      Average total size (kbytes): 0
      Maximum resident set size (kbytes): 0
      Average resident set size (kbytes): 0
      Major (requiring I/O) page faults: 0
      Minor (reclaiming a frame) page faults: 2559
      Voluntary context switches: 1
      Involuntary context switches: 249
      Swaps: 0
      File system inputs: 0
      File system outputs: 0
      Socket messages sent: 0
      Socket messages received: 0
      Signals delivered: 0
      Page size (bytes): 4096
      Exit status: 0
```

1 W konsoli wpisz: man time

Formatowanie drukowanego tekstu odbywa się za pomocą opcji --format (-f). W podręczniku systemowym w sekcji **FORMATTING THE OUTPUT** znajdują się **The resource specifiers**, są to litery poprzedzone znakiem procenta (**%**). Dla przykładu wydrukujmy kompletną liczbę sekund, których program użył bezpośrednio w trybie użytkownika oraz w trybie jądra systemu. Pierwszy z nich używa przełącznika **%U**, drugi **%S**. Jedno i drugie polecenie drukuje to samo (czasy wykonania naturalnie mogą się różnić).

```
$ /usr/bin/time --format="%U\t%S" ./nazwa_programu
```

```
$ /usr/bin/time -f "%U\t%S" ./nazwa_programu
```

\t, \n oraz \\ są dozwolonymi znakami, które odpowiednio oznaczają tabulację, przejście do nowej linii, wydrukowanie ukośnika (\).

Dodatek C

W dodatku tym omówię proces instalacji serwera MySQL, oraz pewnych czynności potrzebnych, aby biblioteka mysql.h była widziana i można było ją połączyć z naszym programem. Podstawowe polecenia SQL za pomocą których utworzymy bazę danych potrzebną do rozdziału 11 znajdują się w dalejszej części niniejszego dodatku.

C.1 Instalacja MySQL

W zależności od tego jaką dystrybucję posiadamy sposób instalacji może się różnić. Dla Debiana i Ubuntu instalacja MySQL sprowadza się do wykonania poniższego polecenia z uprawnieniami administratora.

```
# apt-get install mysql-server
```

Podczas instalacji kreator poprosi o podanie hasła. Jest to ważne, ponieważ to hasło będzie Tobie potrzebne. Po instalacji wymagane jest zrobienie trzech czynności. Biblioteka libmysqlclient-dev posiada specjalny skrypt o nazwie **mysql_config**, dzięki któremu będziemy w stanie połaczyć nasz program z bazą. Pierwszym poleceniem, które musimy wykonać to instalacja owej biblioteki:

```
# apt-get install libmysqlclient-dev
```

A następnie wykonujemy poniższe polecenia.

```
$ mysql_config --libs
```

Terminal powinien wyświetlić mniej więcej taką odpowiedź

```
-Wl,-Bsymbolic-functions -rdynamic -L/usr/lib/mysql -lmysqlclient
```

Ostatnim poleceniem jest

```
$ mysql_config --cflags
```

Na co terminal reaguje mniej więcej tak

```
-I/usr/include/mysql  -DBIG_JOINS=1  -fno-strict-aliasing   -DUNIV_LINUX
```

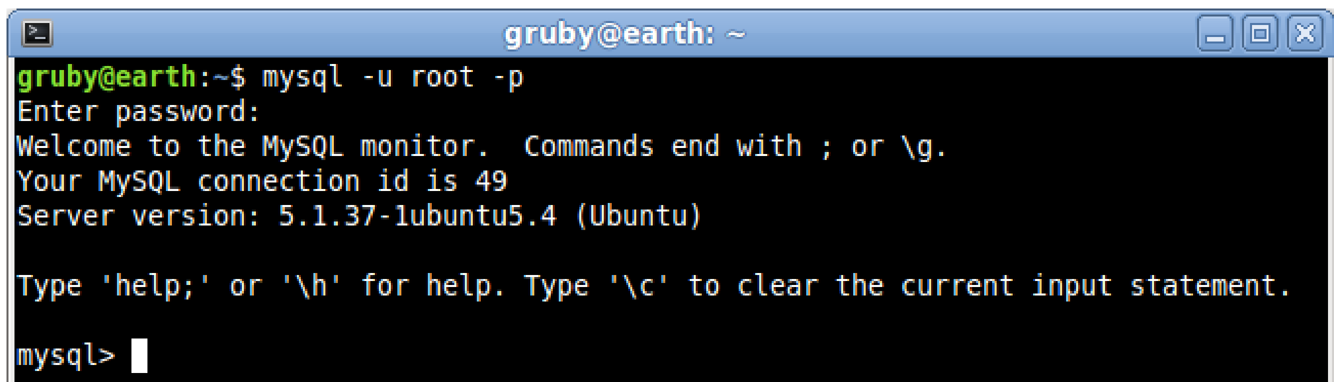
Po tych czynnościach będziemy mogli skompilować program tak, jak jest to pokazane w rozdziale 11.

C.2 Podstawowe polecenia MySQL

Aby móc korzystać z MySQL musimy w terminalu wpisać

```
$ mysql -u root -p
```

Następnie podajemy hasło i zostajemy zalogowani do bazy jako administrator. Jak pokazano na poniższym rysunku za wiele nie możemy z tego wyciągnąć jeśli nie mamy trochę pojęcia na ten temat.



Rysunek C1. Widok MySQL.

Przede wszystkim, jeśli chcemy zacząć operować na jakiejś bazie danych, to musimy ją wybrać. Ale jak to zrobić, skoro nie wiemy jakie bazy są utworzone? Z pomocą przychodzi polecenie:

```
mysql> show databases;
```

W standardzie powinny być dwie bazy, a mianowicie: **information_schema**, oraz **mysql**. Wybieramy **mysql**, właśnie jak ją wybrać? Polecenie use tutaj ma swoje zastosowanie, czyli:

```
mysql> use mysql;
```

Terminal poinformuje nas o tym, że baza została zmieniona. Teraz, aby zobaczyć jakie tabele są

dostępne w tej bazie, używamy polecenia:

```
mysql> show tables;
```

Teraz, aby zobaczyć jakie dane są np. w tabeli **help_category**. Wpisujemy polecenie:

```
mysql> select * from help_category;
```

Gdzie można by sobie to było przetłumaczyć w następujący sposób „Wybierz coś z czegoś”. W naszym przypadku „Wybierz wszystko z help_category”. Gwiazdka to znak wieloznaczności, dlatego oznacza wszystko, czyli wszystkie rekordy będące w danej tabeli.

Aby zawęzić nasze wyszukiwania do pierwszych dziesięciu rekordów musimy użyć dodatkowej części w poleceniu wyświetlającym, a mianowicie:

```
mysql> select * from help_category where help_category_id <= 10;
```

Gdzie **help_category_id** jest pierwszą kolumną naszej tabeli. Aby sprawdzić jakie kolumny ma nasza tabela wpisujemy:

```
mysql> show columns from help_category;
```

Aby wynik naszego wyszukiwania był jeszcze węższy, a konkretnie był w pewnym zadanym przedziale, musimy użyć pewnej sztuczki, którą zresztą już znasz po rozdziale operatory logiczne. Aby wyświetlić wszystkie rekordy, których nazwy rozpoczynają się na litery z przedziału <A;G> musimy wpisać następujące polecenie:

```
mysql> select * from help_category where name >= 'A' and name <= 'G';
```

Aby posortować dane w kolejności rosnącej, lub malejącej używamy polecenia następującego:

```
mysql> select * from help_category where help_category_id <= 10 order by
help_category_id desc;
```

Podajemy warunek dla którego mają zostać wyświetlone informacje, a następnie wpisujemy **order by** po którym następuje nazwa kolumny i słowo **desc**, bądź **asc**, które odpowiednio sortują malejąco bądź rosnąco. Tak więc aby wyświetlić wszystkie rekordy, których **help_category_id** jest większe od 5 a mniejsze od 15 oraz aby były posortowane malejąco według nazwy wpisujemy:

```
mysql> select * from help_category where help_category_id >= 5 and
help_category_id <= 15 order by name desc;
```

By nie ingerować w tę bazę, stworzymy nową do której będziemy mogli dodać nowe tabele, uzupełnić je o konkretną ilość rekordów, usuwać i robić różne operacje na bazie danych. Zacznijmy więc od utworzenia nowej bazy danych, którą nazwiemy **building.**

```
mysql> create database `building`;
```

Po utworzeniu bazy musimy utworzyć tabelę, w której będziemy przechowywać rekordy. Podczas tworzenia tabeli podajemy listę kolumn wraz z odpowiednimi parametrami, poniżej polecenia znajduje się opis.

```
mysql> CREATE TABLE  `building`.`tallest` (
    -> `id` INT NOT NULL AUTO_INCREMENT ,
    -> `category` VARCHAR( 40 ) NOT NULL ,
    -> `structure` VARCHAR( 40 ) NOT NULL ,
    -> `country` VARCHAR( 40 ) NOT NULL ,
    -> `city` VARCHAR( 40 ) NOT NULL ,
    -> `height` FLOAT NOT NULL ,
    -> `year` SMALLINT NOT NULL ,
    -> PRIMARY KEY (  `id` ));
```

Jeśli nie podamy na końcu średnika, to przejdziemy do nowej linii, która zaczyna się „strzałką" i wpisujemy dalszą część polecenia. Użyteczne, gdy polecenie jest dość długie, przez co robi się znacznie bardziej przejrzyste. Wszystkie polecenie SQL można pisać wielkimi bądź małymi literami, kwestia upodobań. Tabelę tworzymy w pierwszej linii podając nazwę bazy danych a po kropce nazwę tabeli. Po nawiasie otwierającym wpisujemy nazwy kolumn oraz ich atrybuty. Nie czas i miejsce na rozwodzenie się co one konkretnie oznaczają i dlaczego tak, a nie inaczej, nie mniej jednak pierwsza kolumna to **id**, która przyjmuje wartości całkowite oraz posiada **auto increment**, czyli automatyczne zwiększanie licznika. Kolejne cztery kolumny są typu **varchar**, czyli maksymalny tekst może mieć tyle ile podajemy w nawiasach, lecz w przypadku nie wykorzystania całości rozmiar jest zmniejszany. Kolumna **height** jest typu rzeczywistego, ostatnia kolumna **year** jest typu smallint.

Dodawanie rekordu odbywa się w następujący sposób:

```
mysql> INSERT INTO `building`.`tallest` (`id`, `category`, `structure`,
`country`, `city`, `height`, `year`) VALUES (NULL, 'Concrete tower',
'Guangzhou TV & Sightseeing Tower', 'China', 'Guangzhou', '610', '2009');
```

No tutaj za dużo też nie ma co tłumaczyć, pewien schemat, według którego dodajemy do bazy i już. Jedyną rzeczą o której warto powiedzieć, to to, że w miejsce wartości kolumny **id** wpisujemy NULL, bo jest **auto increment**, więc wartość zostanie automatycznie zwiększona.

Usuwanie rekordów jest rzeczą bardzo prostą, jeśli mamy id kolumny, a skoro my mamy, to możemy wykorzystać następujące polecenie:

```
mysql> DELETE FROM `building`.`tallest` WHERE id = X;
```

Gdzie X jest oczywiście pewną wartością całkowitą reprezentującą dany rekord. Edytowanie wartości danego rekordu realizuje się za pomocą następującego polecenia:

```
mysql> UPDATE `building`.`tallest` SET `year` = '2010' WHERE
`tallest`.`id` = X;
```

Można to przetłumaczyć następująco „Z bazy **building**, tabeli **tallest** uaktualniamy rok przypisując mu wartość '2010' gdzie **id** tego rekordu równa się X".

Aby usunąć tabelę należy wpisać poniższe polecenie. Natomiast zwracam uwagę na to, iż po usunięciu nie ma możliwości cofnięcia, dlatego też należy zastanowić się dwa razy zanim podejmiemy taką decyzję.

```
mysql> DROP TABLE `building`.`tallest`;
```

Analogicznie jest z usunięciem całej bazy.

```
mysql> DROP DATABASE building;
```