Podstawy programowania

wykład

Dr hab. inż. Grzegorz Dudek, prof. WSTI

Wyższa Szkoła Technologii
Informatycznych w Katowicach

Katowice 2016

Spis treści

1. INFORMACJE WSTĘPNE

1.1. Zakres tematyczny wykładu

Patrz spis treści.

Wykład oparty jest głównie na [Pro].

1.2. Zalecana literatura

[Gre] Gręboszz J.: Symfonia C++ Standard. Edition 2000, 2009.

[Pra] Prata S.: Szkoła programowania. Język C. Helion.

[Har] Harris S., Ross J.: Algorytmy od podstaw. Helion.

[Wro] Wróblewski P.: Algorytmy, struktury danych i techniki programowania. Helion.

[Pro] Programowanie w C. Wikibooks. http://upload.wikimedia.org/wikibooks/pl/6/6a/C.pdf

[Pys] Pyszczuk A.: Programowanie w języku C. http://www.arturpyszczuk.pl/files/c/pwc.pdf

1.3. Podstawowe definicje

Algorytm:

- zbiór dobrze zdefiniowanych kroków prowadzących do wykonania pewnego zadania
- w matematyce skończony ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań
- skończony ciąg/sekwencja reguł, które aplikuje się na skończonej liczbie danych, pozwalający rozwiązywać zbliżone do siebie klasy problemów
- zespół reguł charakterystycznych dla pewnych obliczeń lub czynności informatycznych

Algorytm ma przeprowadzić system z pewnego stanu początkowego do pożądanego stanu końcowego. Badaniem algorytmów zajmuje się **algorytmika**. Algorytm może zostać zaimplementowany w postaci **programu komputerowego**.

```
#include <stdio.h> //komentarz
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");
      return 0;
}
```

Przykładowy program w języku C.

```
program prog14; {komentarz}
uses crt;
var   imie:string[25];
      i:integer;
begin
      clrscr;
      write('Jakie jest twoje imie?');
      readln(imie);
      writeln('Cześć ',imie);
      for i := 25 downto 1 do
            write(imie[i]);
      writeln(' csezC'); readln;
end.
```

Przykładowy program w Pascalu.

```
<html>
<head>
  <link rel="stylesheet" type="text/css" href="./standard.css" >
  <META HTTP-EQUIV="Content-type" CONTENT="text/html; charset=UTF-8">
</head>
<body>
  <h1>Odczyt danych z pliku</h1>
  <?php
     $file_name="dane.txt";
     $answer=file($file_name);

     echo "<table border=1>\n";
     for($n=0; $n <= 3; $n++) {
           echo "<tr>";
        $elems=explode("|",$answer[$n]);
        for($i=0; $i <= 3; $i++) {
          echo "<td align=left>".$elems[$i]."</td>\n";
        }
        echo "<td><img src=spider".$elems[4].".gif ></td>\n";
        echo "</tr>\n";
     }
     echo "</table>";
  ?>
</body>
</html>
```

Przykładowy program w php.

```
<HTML>
<HEAD>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
</HEAD>
<SCRIPT LANGUAGE = "JavaScript">
<!-- Ukrycie przed przeglądarkami nieobsługującymi JavaScript
      for (var i = 1; i <= 1000; i++){ //komentarz
            if ((i % 2) != 0)
                  continue;
            document.write (i + " ");
}
// Koniec kodu JavaScript -->
</SCRIPT>
<BODY>
</BODY>
</HTML>
```

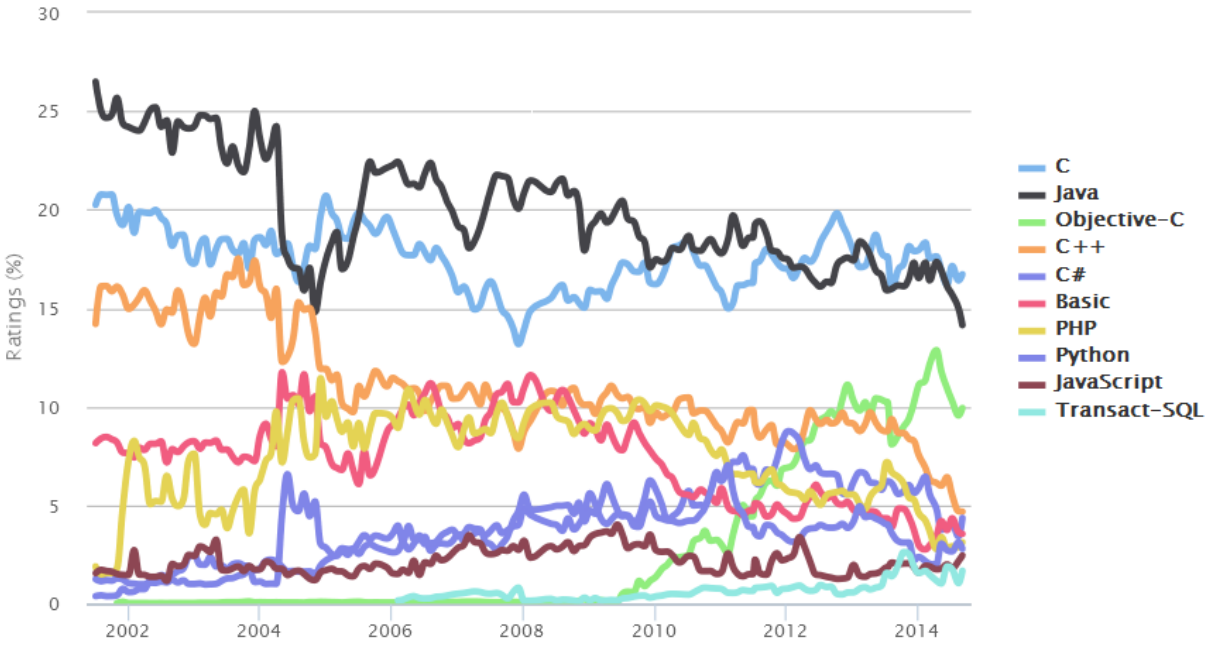
Przykładowy program w JavaScript.

```
figure('Renderer','zbuffer')
Z = peaks;
surf(Z);
axis tight
set(gca,'NextPlot','replaceChildren');
F(20) = struct('cdata',[],'colormap',[]);
for j = 1:200
    surf(.301+cos(1*pi*j/50)*Z,Z)
    F(j) = getframe;
end
```

Przykładowy program w Matlabie.

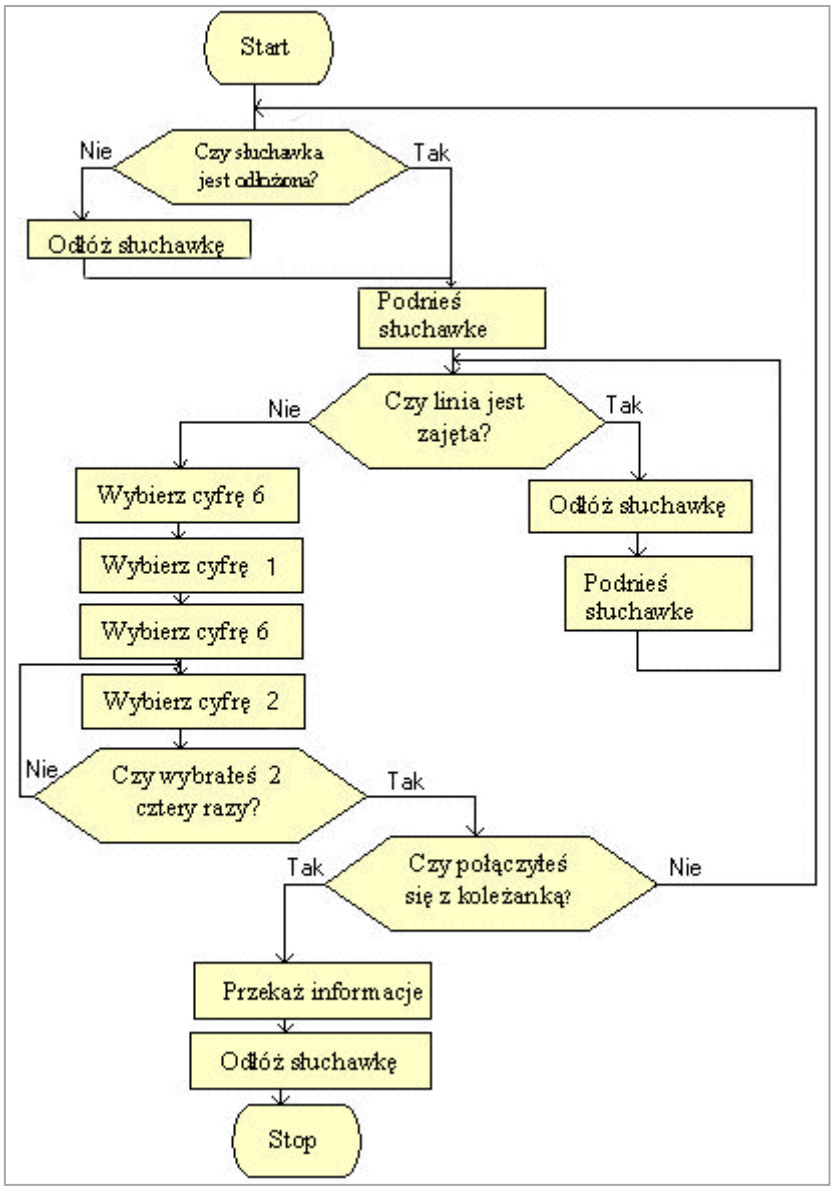
Ranking popularności języków programowania:
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

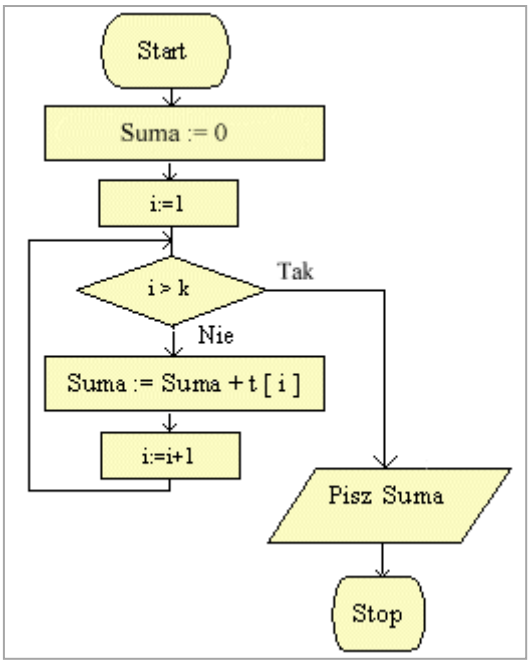
Sep 2014	Sep 2013	Change	Programming Language	Ratings	Change
1	1		C	16.721%	-0.25%
2	2		Java	14.140%	-2.01%
3	4	▲	Objective-C	9.935%	+1.37%
4	3	▼	C++	4.674%	-3.99%
5	6	▲	C#	4.352%	-1.21%
6	7	▲	Basic	3.547%	-1.29%
7	5	▼	PHP	3.121%	-3.31%
8	8		Python	2.782%	-0.39%
9	9		JavaScript	2.448%	+0.43%
10	10		Transact-SQL	1.675%	-0.32%
11	11		Visual Basic .NET	1.532%	-0.31%
12	12		Perl	1.369%	-0.32%
13	13		Ruby	1.281%	-0.10%
14	-	⇈	Visual Basic	1.272%	+1.27%
15	14	▼	Delphi/Object Pascal	1.157%	+0.26%
16	26	⇈	F#	0.990%	+0.49%
17	15	▼	Pascal	0.893%	+0.01%
18	-	⇈	Swift	0.852%	+0.85%
19	19		MATLAB	0.818%	+0.18%
20	17	▼	PL/SQL	0.809%	+0.13%



Algorytm wygodnie jest wyrażać za pomocą schematu blokowego, który pokazuje czynności/działania/instrukcje i wzajemne ich powiązania. Schemat blokowy składa się z następujących elementów:

- strzałka − wskazuje jednoznacznie powiązania i ich kierunek,
- operand − prostokąt, do którego wpisywane są wszystkie operacje z wyjątkiem instrukcji wyboru,
- predykat − romb, do którego wpisywane są wyłącznie instrukcje wyboru,
- etykieta − owal służący do oznaczania początku bądź końca sekwencji schematu.


Start
Czy słuchawka jest odłożona?
Nie
Tak
Odłóż słuchawkę
Podnieś słuchawke
Czy linia jest zajęta?
Nie
Tak
Wybierz cyfrę 6
Wybierz cyfrę 1
Wybierz cyfrę 6
Wybierz cyfrę 2
Odłóż słuchawkę
Podnieś słuchawke
Czy wybrałeś 2 cztery razy?
Nie
Tak
Czy połączyłeś się z koleżanką?
Tak
Nie
Przekaż informacje
Odłóż słuchawkę
Stop



Program komputerowy to sekwencja symboli opisująca obliczenia zgodnie z pewnymi regułami zwanymi **językiem programowania**. Program jest zazwyczaj wykonywany przez komputer, czasami bezpośrednio – jeśli wyrażony jest w języku zrozumiałym dla danej maszyny lub pośrednio – gdy jest kompilowany lub interpretowany przez inny program.

Formalne wyrażenie metody obliczeniowej w postaci języka zrozumiałego dla człowieka nazywane jest **kodem źródłowym**, podczas gdy program wyrażony w postaci zrozumiałej dla maszyny (to jest za pomocą ciągu zer i jedynek) nazywany jest **kodem maszynowym** bądź postacią binarną (wykonywalną). **Kompilator** tłumaczy kod źródłowy zapisany w danym języku programowania na kod maszynowy, dzięki czemu możliwe staje się jego późniejsze uruchomienie. **Interpreter** natomiast odczytuje kod źródłowy na bieżąco, analizuje go i wykonuje kolejne porcje przetłumaczonego kodu. Programy przeznaczone do interpretacji często nazywane są **skryptami**.

Programy komputerowe można zaklasyfikować według ich zastosowań. Wyróżnia się: aplikacje użytkowe, systemy operacyjne, gry wideo, kompilatory i inne.

W najprostszym modelu **wykonanie programu** polega na umieszczeniu go w pamięci operacyjnej komputera i wskazaniu procesorowi adresu pierwszej instrukcji. Po tych czynnościach procesor będzie wykonywał kolejne instrukcje programu, aż do jego zakończenia. Program może zakończyć się w dwojaki sposób: poprawnie lub błędnie.

Program komputerowy będący w trakcie wykonania nazywany jest **procesem** lub zadaniem.

Program można podzielić na dwie części (obszary):

- część kodu (składającą się z instrukcji sterujących działaniem procesora),
- część danych (składającą się z danych wykorzystywanych i opracowywanych przez program, np. adresów pamięci, stałych liczbowych, komunikatów tekstowych).

Programowanie jest procesem tworzenia programów. Jest to cykliczny proces polegający na:

- edycji kodu źródłowego (implementacja),
- uruchamianiu programu (kompilacja),
- analizie działania,
- powrocie do edycji kodu źródłowego w celu poprawienia błędów lub dalszego poszerzania funkcjonalności.

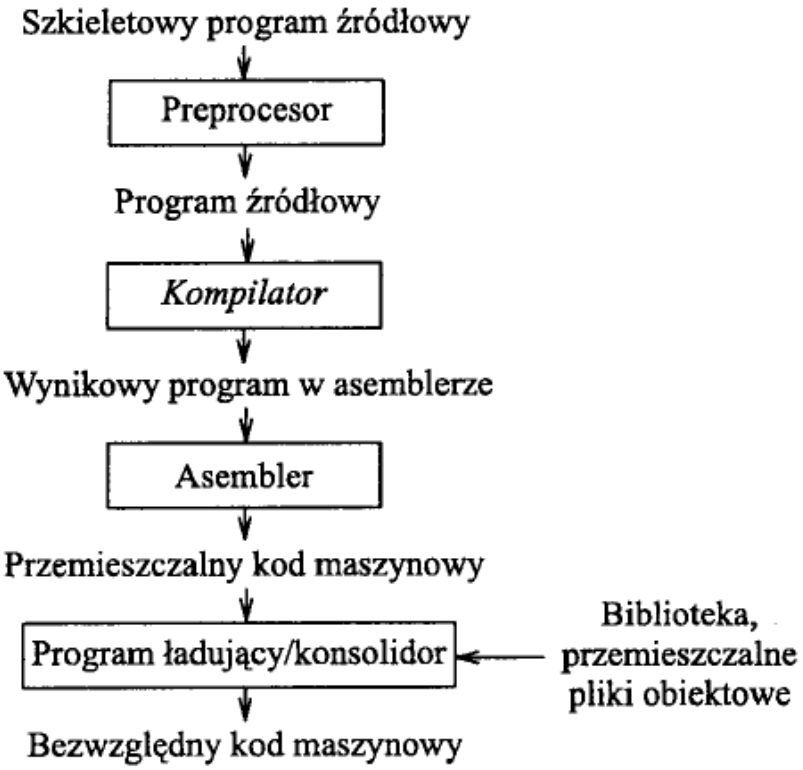
Przed przystąpieniem do programowania należy określić cel programu i zaprojektować program (tu przydają się schematy blokowe). Kod źródłowy zapisuje się w pliku tekstowym (w języku C plik taki ma rozszerzenie *.c lub *.cpp).

1.4. Proces kompilacji

Kompilator – program służący do automatycznego tłumaczenia kodu napisanego w jednym języku (języku źródłowym) na równoważny kod w innym języku (języku wynikowym). Proces ten nazywany jest kompilacją. Każda kompilacja składa się z dwóch części: **analizy** i **syntezy**. Analiza polega na rozłożeniu programu na części składowe i stworzeniu jego pośredniej reprezentacji. Synteza polega na przekształceniu reprezentacji pośredniej na program wynikowy.

Do utworzenia pliku wykonywalnego, oprócz samego kompilatora, może być również potrzebne użycie innych programów. Program źródłowy może być podzielony na moduły przechowywane w oddzielnych plikach. Zbieraniem plików programu może się zajmować oddzielny program, zwany **preprocesorem**. Preprocesor może również rozwijać skróty, zwane makrami, w instrukcje języka źródłowego.

Program wynikowy, stworzony przez kompilator, może wymagać dalszego przetwarzania zanim zostanie uruchomiony. Kompilacja przedstawiona poniżej tworzy kod w asemblerze (język), który jest tłumaczony przez asembler (program) na kod maszynowy, a potem łączony (*linkowany*) z funkcjami bibliotecznymi w kod, który dopiero może być uruchamiany na komputerze. Kompilatory, które produkują kod w asemblerze (jest to język programowania) przekazywany do dalszego przetwarzania do asemblera (jako kompilatora z języka asemblera do języka maszynowego) nazywane są kompilatorami pośrednimi.



Kod asemblera jest mnemonicznym zapisem kodu maszynowego, w którym używa się nazw zamiast binarnych kodów operacji i adresów pamięci. Typowa sekwencja rozkazów w asemblerze może mieć postać (jest to zapis instrukcji `b = a + 2;`):

w asemblerze:	w kodzie maszynowym:
`MOV a, R1`	`(np.: „0001 01 00 00000000”)`
`ADD #2, R1`	`(np.: „0011 01 10 00000010”)`
`MOV R1, b`	`(np.: „0010 01 00 00000100”)`

Analiza programu źródłowego

W kompilacji analiza składa się z trzech faz:

- **Analizy liniowej (analiza leksykalna, skanowanie)**, w której strumień znaków, składający się na program wejściowy, jest wczytywany od lewej do prawej i grupowany w symbole leksykalne (*atomy leksykalne, tokeny*), czyli ciągi znaków mających razem pewne znaczenie. Moduł wykonujący analizę leksykalną nazywa się lekserem, skanerem lub analizatorem leksykalnym.

- **Analizy hierarchicznej (analiza składniowa lub syntaktyczna),** w której znaki lub symbole leksykalne są grupowane hierarchicznie w zagnieżdżone struktury mające wspólne znaczenie.

- **Analizy semantycznej (analizy znaczeniowej),** w której przeprowadzone są pewne działania, mające zapewnić, że składniki programu pasują do siebie pod względem znaczenia.

Przykładowo w analizie liniowej, znaki instrukcji przypisania:

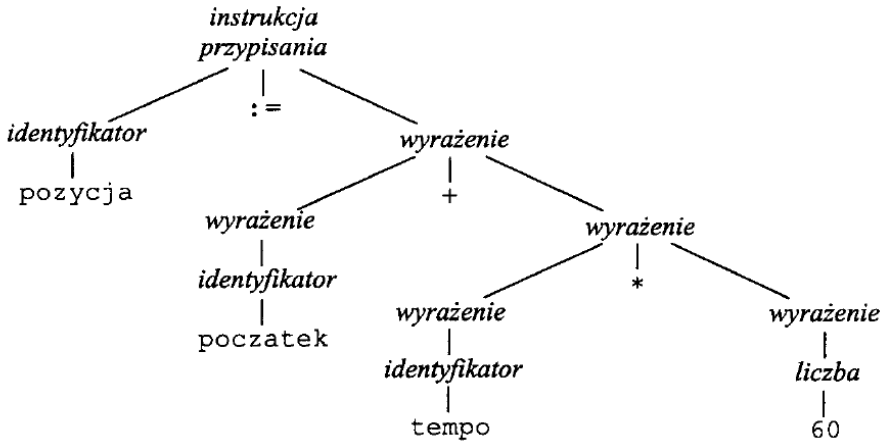
```
pozycja:=poczatek+tempo*60
```

są pogrupowane w następujące symbole leksykalne:

- identyfikator `pozycja`
- symbol przypisania `:=`
- identyfikator `poczatek`
- znak plus `+`
- identyfikator `tempo`
- znak mnożenia `*`
- liczba `60`

Ewentualne odstępy rozdzielające znaki tych symboli leksykalnych zostaną wyeliminowane podczas analizy leksykalnej.

Analiza hierarchiczna polega na grupowaniu symboli leksykalnych programu źródłowego w wyrażenia gramatyczne, które są reprezentowane przez kompilator do syntezy kodu wynikowego. Zwykle wyrażenia gramatyczne są reprezentowane przez drzewo wyprowadzenia lub drzewo składniowe.



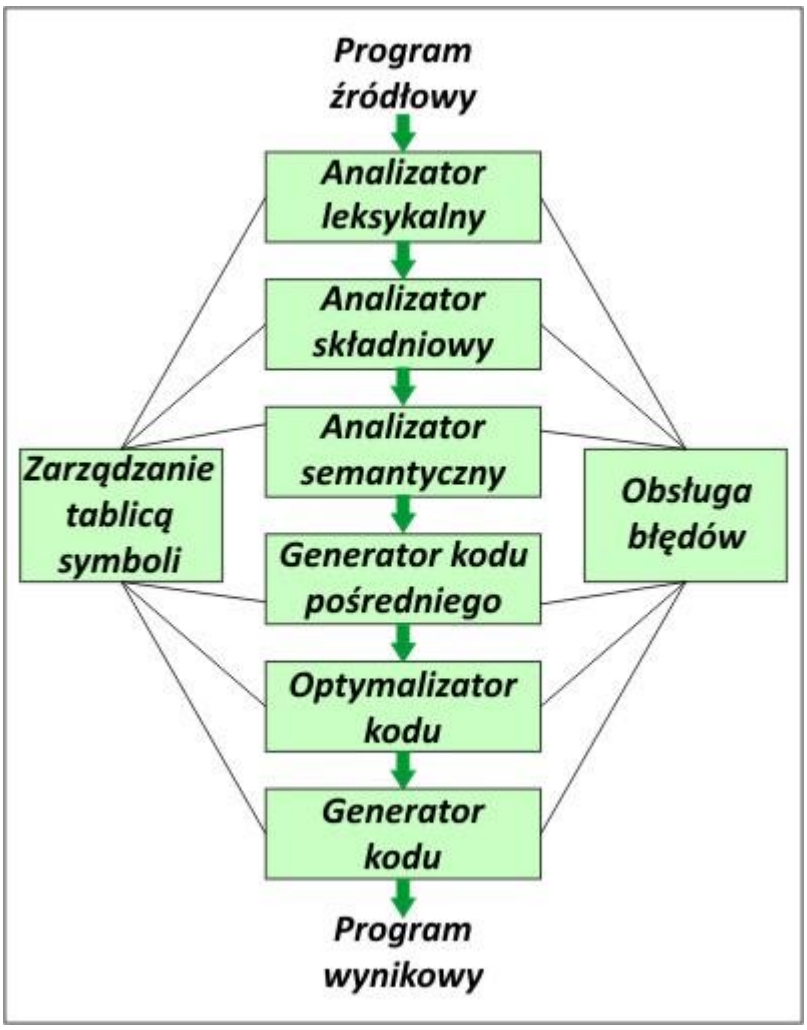
Rys. 1.4. Drzewo wyprowadzenia dla `pozycja:=poczatek+tempo*60`

Drzewo wyprowadzenia lub drzewo składniowe opisują składniową strukturę wejścia.

Analiza semantyczna polega na kontroli programu źródłowego wyszukującej błędy semantyczne oraz na zbieraniu informacji dla kolejnej fazy, jaką jest generacja kodu. Do analizy semantycznej, w celu zidentyfikowania operatorów i argumentów wyrażeń i instrukcji, używa się hierarchicznej struktury otrzymanej z analizy składniowej. Ważnym elementem analizy semantycznej jest kontrola typów.

Fazy kompilatora

Kompilator działa w fazach, które po kolei przekształcają program z jednej postaci na inną.



Kompilator może być podzielony na kilka niezależnych części wykonujących odrębne i ściśle określone zadania:

- **skaner** – dokonuje analizy leksykalnej; wyodrębnia podstawowe symbole języka (atomy lub tokeny) oraz usuwa znaki nie mające wpływu na sam program (np. odstępy, komentarze). Jego dane wyjściowe to postać pośrednia programu źródłowego zawierająca atomy wraz z krótkim opisem.

- **parser** – dokonuje analizy składniowej; ma na celu sprawdzenie poprawności syntaktycznej przez dokonanie rozbioru podprogramu na części składowe i zbudowanie odpowiedniego drzewa składniowego.

- **generator** – dokonuje przekładu kodu źródłowego w postaci wewnętrznej otrzymanej po analizie składniowej na kod wynikowy związany zazwyczaj z konkretną maszyną docelową.

- **optymalizator** – dokonujący optymalizacji kodu pod różnymi kątami, np. szybkości wykonywania, ilości pamięci wykorzystywanej przez program, możliwości zrównoleglenia itp.

- **konsolidator** (linker) – w trakcie procesu konsolidacji łączy skompilowane pliki zawierające kod obiektowy lub pliki bibliotek statycznych tworząc w ten sposób plik wykonywalny.

- **debugger** (odpluskwiacz) – program służący do dynamicznej kontroli nad wykonaniem kodu, w celu odnalezienia i identyfikacji zawartych w nim błędów. Współczesne debuggery pozwalają na efektywne śledzenie wartości poszczególnych zmiennych, wykonywanie instrukcji krok po kroku czy wstrzymywanie działania programu w określonych miejscach.

Zarządzanie tablicą symboli

Ważną funkcją kompilatora jest zapamiętywanie identyfikatorów używanych w programie źródłowym i zbieranie informacji o różnych atrybutach tych identyfikatorów (rozmiarze pamięci zajętej dla identyfikatora, typie, zasięgu), a w przypadku procedur podają liczbę, typy i metodę przekazywania argumentów oraz ewentualnie typ wyniku.

Tablica symboli jest strukturą danych zawierającą dla wszystkich identyfikatorów rekordy z ich atrybutami. Każdy identyfikator znaleziony w programie źródłowym podczas analizy leksykalnej jest dodawany do tej tablicy, a jego atrybuty mogą być dodawane w kolejnych fazach.

Wykrywanie i zgłaszanie błędów

Podczas każdej fazy kompilacji można napotkać błędy w programie źródłowym. Większość błędów wykrywana jest podczas analizy składniowej i semantycznej.

W fazie analizy leksykalnej mogą zostać wykryte błędy, jeśli znaki znajdujące się na wejściu nie stanowią żadnego symbolu leksykalnego języka. Kiedy strumień nie pasuje do zasad budowy strukturalnej (składniowej) języka, błędy są znajdowane w fazie analizy składniowej. Podczas analizy semantycznej kompilator wykrywa konstrukcje z poprawną strukturą składniową, ale na których nie daje się zastosować użytej operacji.

Błędy więc mogą być:

- leksykalne, np. błędnie wpisany identyfikator, słowo kluczowe lub operator,

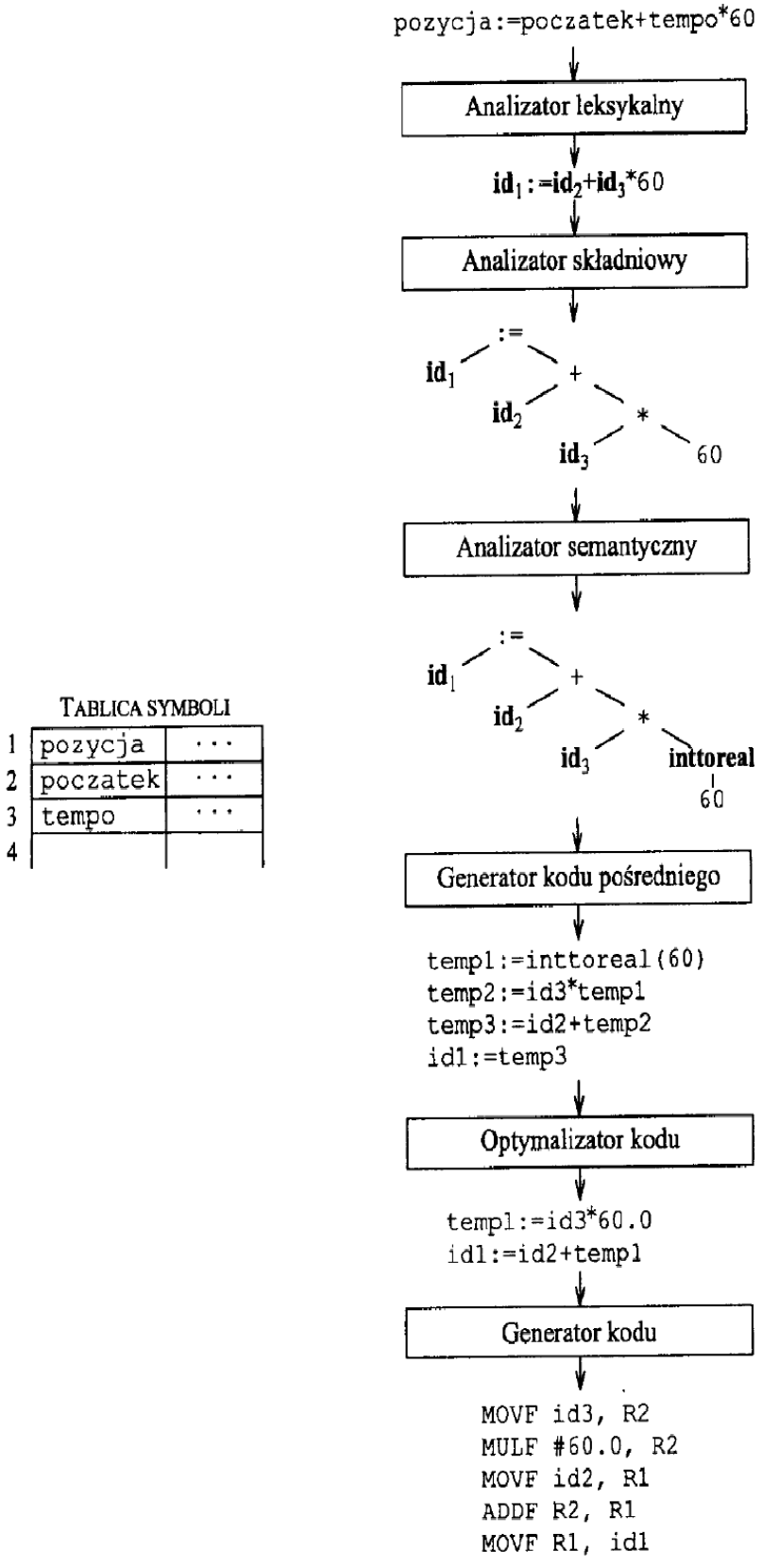
- składniowe, np. wyrażenie arytmetyczne z niewyważonymi nawiasami,

- semantyczne, np. zastosowanie operatora do niekompatybilnego argumentu,

- logiczne, np. wywołanie rekurencyjne w nieskończonej pętli.

Fazy analizy

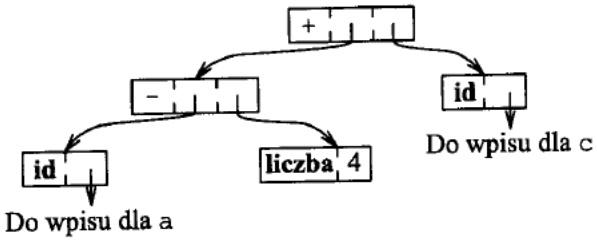
Podczas postępu translacji zmienia się wewnętrzna reprezentacja programu źródłowego w kompilatorze.



Podczas fazy analizy leksykalnej wczytuje się znaki programu źródłowego i grupuje je w strumień **symboli leksykalnych**, z których każdy reprezentuje spójną logicznie sekwencję znaków takich jak identyfikator, słowo kluczowe (if, while, ...), znaki przestankowe (przecinki, średniki, ...) lub operatory kilkuznakowe (np. :=). Ciągi znaków tworzących symbole leksykalne są zwane **leksemami**.

Niektóre symbole leksykalne są rozszerzone o tzw. **wartość leksykalną**. Przykładowo, kiedy wczytywany jest identyfikator `tempo`, analizator leksykalny generuje symbol leksykalny **id** oraz do tablicy symboli wpisuje leksem `tempo` (oczywiście tylko za pierwszym razem po jego napotkaniu). Wartość leksykalna odpowiadająca symbolowi **id** jest wskaźnikiem do tablicy symboli i wskazuje `tempo`.

Podczas analizy składniowej przekształcamy strumień symboli leksykalnych do struktury hierarchicznej. Węzły wewnętrzne są rekordami, zawierającymi jedno pole z rodzajem operatora i dwa pola ze wskaźnikami do lewego i prawego potomka. Liście są rekordami zawierającymi dwa lub więcej pól: jedno identyfikujące symbol leksykalny, a pozostałe zawierające informacje o tym symbolu.



Generacja kodu pośredniego

Po zakończeniu analizy składniowej i semantycznej niektóre kompilatory generują wprost reprezentację pośrednią programu źródłowego. Ta reprezentacja może być traktowana jako program dla pewnej abstrakcyjnej maszyny, a powinien on mieć dwie cechy: dać się łatwo utworzyć oraz przetłumaczyć na program wynikowy.

Pośrednia reprezentacja może mieć wiele postaci, np. tzw. „kod trójadresowy", przypominający asembler. Kod trójadresowy składa się z sekwencji rozkazów, z których każdy ma co najwyżej trzy argumenty, np.:

```
temp1 := inttoreal(60)

temp2 := id3*temp1

temp3 := id2+temp2

id1 := temp3
```

Reprezentacja trójadresowa ma trzy cechy:

1. Każdy rozkaz oprócz przypisania może mieć co najwyżej jeden operator. Generując reprezentację pośrednią kompilator musi ustalić kolejność operacji.

2. Kompilator musi wygenerować tymczasowe identyfikatory, aby przechowywać pośrednie wartości obliczone przez rozkaz.

3. Niektóre z instrukcji „trójadresowych” mają mniej niż trzy argumenty.

Optymalizacja kodu

W tej fazie staramy się poprawić kod pośredni, w celu otrzymania kodu maszynowego działającego szybciej. Np. przykład powyższy można zapisać efektywniej:

```
temp1 :=  id3*60.0

id1 := id2+temp1
```

Generacja kodu

Jest to ostatnia faza kompilacji, produkująca zwykle przemieszczalny kod maszynowy lub kod asemblera. Każdej zmiennej zostaje przypisany adres w pamięci. Ważnym zadaniem na tym etapie jest przypisanie właściwych zmiennych do rejestrów.

Kod po translacji na asembler, używający rejestrów 1 i 2 przyjmuje postać:

```
MOVF id3, R2

MULF #60.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id1
```

2. JĘZYK C – WIADOMOŚCI WSTĘPNE

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

Historia

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

W 1954 r. powstał Fortran (*Formula Translator*), a w 1956 r. – Fortran I. Później powstały kolejno:

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

i C w oparciu o B.

B został stworzony przez Kena Thompsona z Bell Labs; był to język interpretowany, używany we wczesnych, wewnętrznych wersjach systemu operacyjnego UNIX. Thompson i Dennis Richie rozwinęli B, nazywając go NB. Dalszy rozwój NB dał C – język kompilowany. Większa część UNIX-a została ponownie napisana w NB, a następnie w C, co dało w efekcie bardziej przenośny system operacyjny. Możliwość uruchamiania UNIX-a na różnych komputerach była główną przyczyną początkowej popularności zarówno UNIX-a, jak i C.

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

Standaryzacje

W 1978 roku Ritchie i Kerninghan opublikowali pierwszą książkę nt. języka C − "*The C Programming Language*", która przez wiele lat była swoistym wyznacznikiem, jak programować w języku C. Była więc to niejako pierwsza standaryzacja, nazywana od nazwisk twórców K&R. Producenci kompilatorów (zwłaszcza AT&T) wprowadzali swoje zmiany, nieobjęte standardem. Te nieoficjalne rozszerzenia zagroziły spójności języka, dlatego też w 1989 roku powstał standard C89 regulujący wprowadzone nowinki. Później wprowadzono standard C99. Ostatnia norma została opublikowana w 2011 roku pod nazwą ISO/IEC 9899:2011. Ta wersja języka jest potocznie nazywana C11.

Zastosowania i przyszłość języka C

Język C został opracowany jako strukturalny język programowania do celów ogólnych. Przez całą swą historię (czyli ponad 40 lat) służył do tworzenia przeróżnych programów – od systemów operacyjnych po programy nadzorujące pracę urządzeń przemysłowych. C, jako język dużo szybszy od języków interpretowanych (Perl, Python) oraz uruchamianych w maszynach wirtualnych (np. C#, Java) może bez problemu wykonywać złożone operacje nawet wtedy, gdy nałożone są dość duże limity czasu wykonywania pewnych operacji. Jest on przy tym bardzo przenośny – może działać praktycznie na każdej architekturze sprzętowej pod warunkiem opracowania odpowiedniego kompilatora. Często wykorzystywany jest także do oprogramowywania mikrokontrolerów i systemów wbudowanych. Jednak w niektórych sytuacjach język C jest mniej przydatny, zwłaszcza chodzi tu o obliczenia matematyczne, wymagające dużej precyzji (w tej dziedzinie znakomicie spisuje się Fortran) lub też dużej optymalizacji dla danego sprzętu (wtedy niezastąpiony jest język asemblera).

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

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

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

Zintegrowane Środowiska Programistyczne

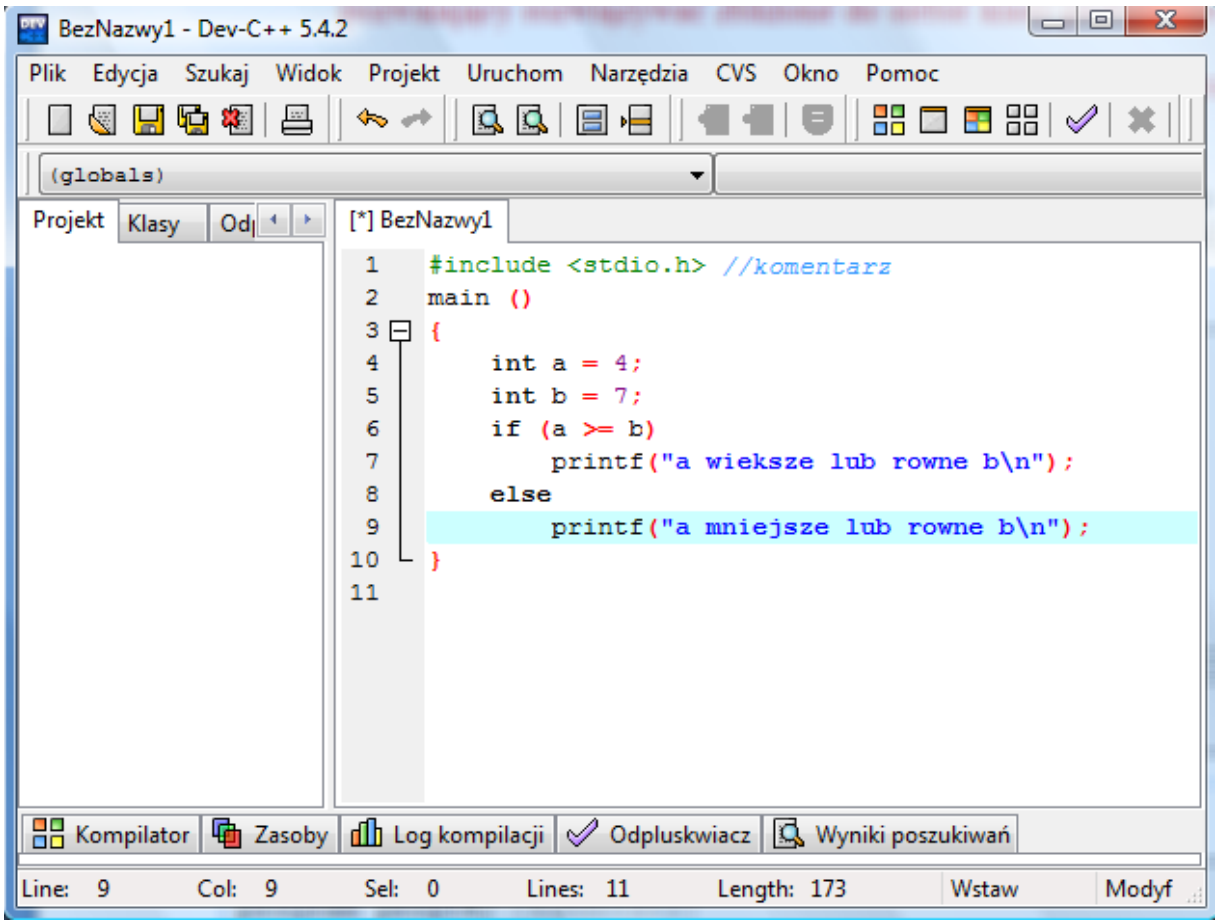
Zamiast osobnego kompilatora i edytora, możesz wybrać Zintegrowane Środowisko Programistyczne (*Integrated Development Environment*, IDE). IDE jest zestawem wszystkich programów, które potrzebuje programista, najczęściej z interfejsem graficznym. IDE zawiera kompilator, linker i edytor, z reguły również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

Bardzo popularny IDE to płatny (istnieje także jego darmowa wersja) Microsoft Visual C++ (MS VC++). Popularne darmowe IDE to np.:

- Code::Blocks dla Windows jak i Linux, dostępny na stronie www.codeblocks.org,
- KDevelop (Linux) dla KDE,
- NetBeans multiplatformowy, darmowy do ściągnięcia na stronie www.netbeans.org,
- Eclipse z wtyczką CDT (współpracuje z MinGW i GCC),
- Borland C++ Builder dostępny za darmo do użytku prywatnego,
- Xcode dla Mac OS X i nowszy kompatybilny z procesorami PowerPC i Intel,
- Geany dla systemów Windows i Linux; współpracuje z MinGW i GCC, www.geany.org,
- Pelles C, www.smorgasbordet.com,
- Dev-C++ dla Windows, dostępny na stronie www.bloodshed.net.

Dev-C++ jest w pełni funkcjonalnym darmowym środowiskiem programistycznym C/C++ zawierającym wielookienkowy edytor kodu źródłowego z podświetlaniem składni, kompilator, debbuger, linker. Środowisko posiada także narzędzie do tworzenia pakietów instalacyjnych napisanych programów. Instalacja jest prosta i ogranicza się do postępowania zgodnie z

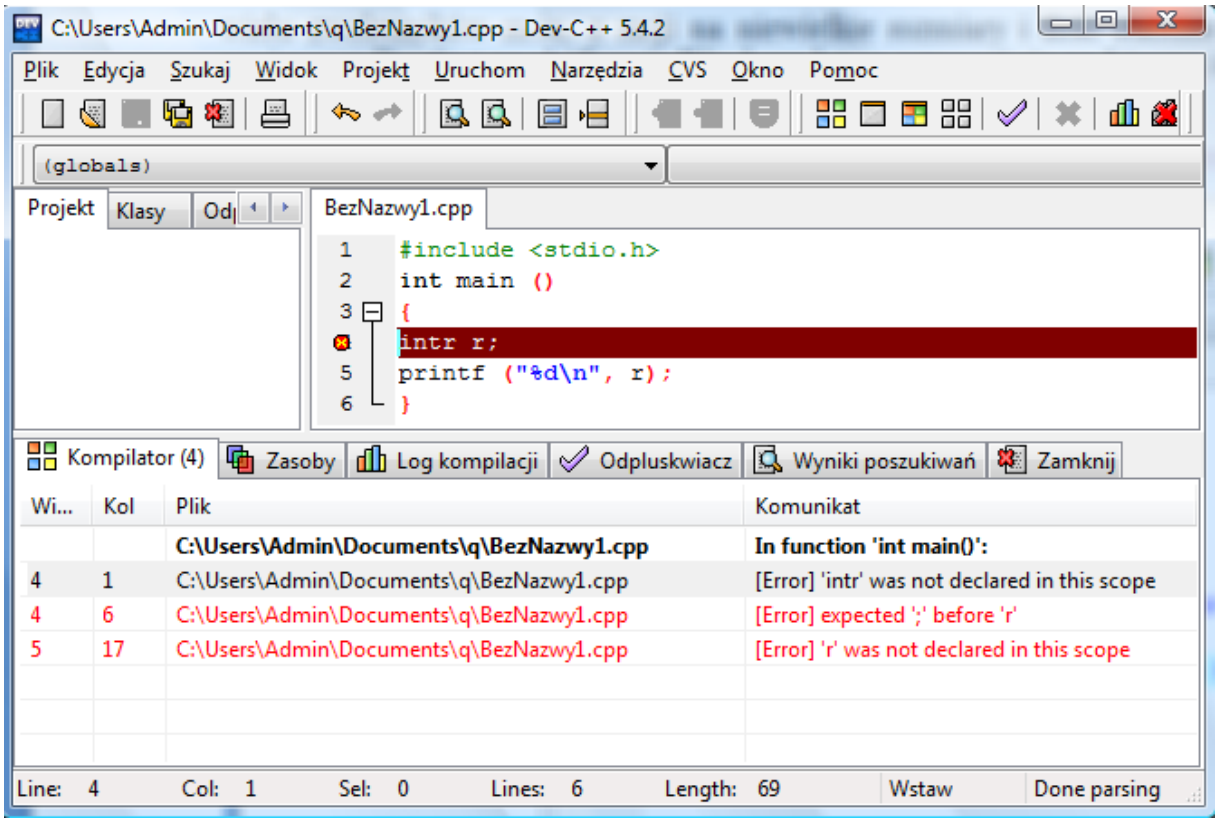
instrukcjami. Nie trzeba dodatkowo instalować żadnych innych programów. Umożliwia tworzenie aplikacji działających pod konsolą oraz okienkowych (w32api). Ma wielu zwolenników m.in. z racji na niewielkie rozmiary i dość stabilne działanie. Kod napisany w DevC++ na platformie Windows łatwo przenosi się na Linuxa i odwrotnie, gdyż Dev-C++ używa kompilatora MinGW (GCC). GCC jest to darmowy zestaw kompilatorów, m.in. języka C rozwijany w ramach projektu GNU. Dostępny jest on na dużą ilość platform sprzętowych, obsługiwanych przez takie systemy operacyjne jak: AIX, *BSD, Linux, Mac OS X, SunOS, Windows.



Komunikaty o błędach

Jedną z najbardziej podstawowych umiejętności, które musi posiąść początkujący programista jest umiejętność rozumienia komunikatów o różnego rodzaju błędach, które sygnalizuje kompilator. Wszystkie te informacje pomogą Ci szybko wychwycić ewentualne błędy (których na początku zawsze jest bardzo dużo). Kompilator ma za zadanie pomóc Ci w szybkiej poprawie ewentualnych błędów, dlatego też umiejętność analizy komunikatów o błędach jest tak ważna.

Przykładowo, jeśli próbujemy skompilować program jak na rysunku poniżej, kompilator zasygnalizuje błędy.



Co widzimy w raporcie o błędach? W linii 4 użyliśmy nieznanego (*not declared*) identyfikatora `intr` – kompilator mówi, że nie zna tego identyfikatora. Ponieważ `intr` nie został rozpoznany jako żaden znany typ, linijka `intr r;` nie została rozpoznana jako deklaracja zmiennej i kompilator zgłasza błąd składniowy (*syntax error*): *expected ';' before 'r'*. W konsekwencji `r` nie zostało rozpoznane jako zmienna i kompilator zgłosi to jeszcze w następnej linijce, gdzie używamy `r`.

3. PIERWSZY PROGRAM W C

Przyjęło się, że pierwszy program napisany w dowolnym języku programowania, powinien wyświetlić tekst "Hello World!" (Witaj Świecie!). Sam język C nie ma żadnych mechanizmów przeznaczonych do wprowadzania i wypisywania danych – musimy zatem skorzystać z odpowiadających za to **funkcji** – w tym przypadku `printf`, zawartej w standardowej bibliotece C. W języku C **deklaracje** funkcji zawarte są w **plikach nagłówkowych** posiadających najczęściej rozszerzenie `.h`, choć można także spotkać rozszerzenie `.hpp`.

W celu umieszczenia w swoim kodzie pewnego pliku nagłówkowego, używamy dyrektywy kompilacyjnej `#include`. Przed procesem kompilacji, w miejsce tej dyrektywy wstawiana jest treść podanego pliku nagłówkowego, w którym zanjduje się deklaracja funkcji.

Poniższy przykład obrazuje, jak przy użyciu dyrektywy `#include` umieścimy w kodzie plik standardowej biblioteki C `stdio.h` (*Standard Input/Output Headerfile*) zawierającą definicję funkcji `printf`:

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

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

```
#include "moj_plik_naglowkowy.h"
```

W każdym programie definiujemy główną funkcję **`main`**, uruchamianą przy starcie programu, zawierającą właściwy kod. Definicja funkcji zawiera, oprócz nazwy i kodu, także typ wartości zwracanej i argumentów pobieranych. Konstrukcja funkcji main:

```
int main (void)
{
      //ciało funkcji
}
```

Typem zwracanym przez funkcję jest `int` (*Integer*), czyli liczba całkowita (będzie to wartość zwracana poleceniem `return`). W nawiasach umieszczane są argumenty funkcji, tutaj zapis `void` oznacza ich pominięcie. Funkcja `main` jako argumenty może pobierać parametry linii poleceń z jakimi program został uruchomiony i pełną ścieżkę do katalogu z programem. Kod funkcji (ciało) umieszcza się w nawiasach klamrowych `{` i `}`.

Wewnątrz funkcji możemy wpisać poniższy kod:

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

Wszystkie polecenia kończymy średnikiem. `return 0;` określa wartość jaką zwróci funkcja (program). Liczba zero zwracana przez funkcję `main()` oznacza, że program zakończył się bez błędów. Błędne zakończenie często (choć nie zawsze) określane jest przez liczbę `1`. Funkcję `main()` kończymy nawiasem klamrowym zamykającym.

Kod całego programu powinien wyglądać jak poniżej:

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

Teraz wystarczy go tylko skompilować i uruchomić.

Może się zdarzyć, że program skompiluje się, uruchomi, ale jego efektu działania nie będzie widać. Dzieje się tak, ponieważ nasz pierwszy program po prostu wypisuje komunikat i od razu kończy działanie, nie czekając na reakcję użytkownika. Jeśli korzystasz ze Zintegrowanego Środowiska Programistycznego (IDE), możesz zaznaczyć, by nie zamykało ono programu po zakończeniu jego działania. Innym sposobem jest dodanie instrukcji, które wstrzymywałyby zakończenie programu. Można to zrobić dodając przed linią z `return` funkcję pobierającą znak z wejścia:

```
getchar();
```

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

Rodzina systemów Unix/Linux:

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

Rodzina systemów MS Windows:

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

4. PODSTAWY JĘZYKA C

4.1. Struktura blokowa

Teraz omówimy podstawową strukturę programu napisanego w C. Język C jest językiem strukturalnym, który ma budowę blokową. Blok jest grupą instrukcji, połączonych w ten sposób, że są traktowane jak jedna całość. W C blok zawiera się pomiędzy nawiasami klamrowymi `{ }`. Blok może także zawierać inne bloki.

Generalnie, blok zawiera ciąg kolejno wykonywanych poleceń. Polecenia zawsze (z nielicznymi wyjątkami) kończą się średnikiem (`;`). W jednej linii może znajdować się wiele poleceń, choć dla zwiększenia czytelności kodu najczęściej pisze się pojedynczą **instrukcję** w każdej linii. Jest kilka typów poleceń, np. instrukcje przypisania, instrukcje warunkowe czy instrukcje pętli. Pomiędzy poleceniami są również odstępy – spacje, tabulacje oraz przejścia do następnej linii, przy czym dla kompilatora te trzy rodzaje odstępów mają takie samo znaczenie. Dla przykładu, poniższe trzy fragmenty kodu źródłowego, dla kompilatora są takie same:

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

2. printf("Hello world");
   return 0;

3. printf("Hello world");



   return 0;
```

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

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

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

4.2. Zasięg zmiennych

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

4.3. Funkcje

Funkcje są ściśle związane ze strukturą blokową – funkcją jest po prostu blok instrukcji, który jest wywoływany w programie za pomocą pojedynczego polecenia. Zazwyczaj funkcja wykonuje pewne określone zadanie. Każda funkcja ma swoją nazwę, za pomocą której jest potem wywoływana w programie oraz blok wykonywanych poleceń. Funkcja może pobierać pewne dane, czyli argumenty funkcji, może także zwracać pewną wartość po zakończeniu wykonywania. Wiele funkcji zawartych jest w standardowych bibliotekach C, np. `printf`. Funkcje można definiować samodzielnie na potrzeby swojego programu. Dobrym nawykiem jest dzielenie dużego programu na zestaw mniejszych funkcji – dzięki temu łatwiej odnaleźć błąd w programie.

Jeśli chcesz użyć jakiejś funkcji, to powinieneś wiedzieć:

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

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

4.4. Biblioteki standardowe

Język C, w przeciwieństwie do innych języków programowania (np. Fortranu czy Pascala) nie posiada absolutnie żadnych słów kluczowych, które odpowiedzialne by były za obsługę wejścia i wyjścia. Składnia C opracowana jest tak, by można było bardzo łatwo przełożyć ją na kod maszynowy. To właśnie dzięki temu programy napisane w języku C są takie szybkie. Pozostaje jednak pytanie – jak umożliwić programom komunikację z użytkownikiem?

W 1983 roku, kiedy zapoczątkowano prace nad standaryzacją C, zdecydowano, że powinien być zestaw instrukcji identycznych w każdej implementacji C. Nazwano je Biblioteką Standardową (czasem nazywaną "libc"). Zawiera ona podstawowe funkcje, które umożliwiają wykonywanie takich zadań jak wczytywanie i zwracanie danych, modyfikowanie zmiennych łańcuchowych, działania matematyczne, operacje na plikach, i wiele innych, jednak nie zawiera żadnych funkcji, które mogą być zależne od systemu operacyjnego czy sprzętu, jak grafika, dźwięk czy obsługa sieci. W programie "Hello World" użyto funkcji z biblioteki standardowej – `printf`, która wyświetla na ekranie sformatowany tekst.

4.5. Komentarze i styl pisania programu

Komentarze – to tekst włączony do kodu źródłowego, który jest pomijany przez kompilator, i służy jedynie dokumentacji. W języku C komentarze zaczynają się od **`/*`** a kończą **`*/`**. Inny typ komentarza, przejęty z języka C++, zaczyna się od **`//`**, a kończy znakiem końca linii.

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

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

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

Dobry styl pisania kodu jest o tyle ważny, że powinien on być czytelny i zrozumiały; po to w końcu wymyślono języki programowania wysokiego poziomu (w tym C), aby kod było łatwo zrozumieć. I tak, należy stosować: **wcięcia** dla odróżnienia bloków kolejnego poziomu (zawartych w innym bloku, podrzędnych; nawiasy klamrowe otwierające i zamykające blok powinny mieć takie same wcięcia), nazwy funkcji i zmiennych powinny kojarzyć się z zadaniem, jakie dana funkcja czy zmienna pełni w programie (np.: `int_to_str`, `IntToStr`).

4.6. Preprocesor

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

Przykładem najczęściej używanej dyrektywy jest `#include`, która nakazuje preprocesorowi włączyć (*include*) w tym miejscu zawartość podanego pliku, tzw. pliku nagłówkowego; najczęściej to będzie plik zawierający funkcje z którejś biblioteki standardowej (`stdio.h` – *STandard Input-Output*, rozszerzenie `.h` oznacza plik nagłówkowy C). Dzięki temu, zamiast wklejać do kodu swojego programu deklaracje kilkunastu, a nawet kilkudziesięciu funkcji, wystarczy wpisać jedną magiczną linijkę!

4.7. Nazwy zmiennych, stałych i funkcji

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

Przykłady błędnych nazw:

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

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

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

5. ZMIENNE

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

5.1. Czym są zmienne?

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

Deklaracja zmiennych

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

```
typ nazwa_zmiennej;
```

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

```
int wiek;
```

Zmiennej w momencie zadeklarowania można od razu przypisać wartość (nazywa się to **inicjalizacją**):

```
int wiek = 17;
```

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

```
int wiek = 17;
printf("%d\n", wiek);
int kopia_wieku; /* tu stary kompilator C zgłosi błąd */
/* deklaracja występuje po instrukcji (printf). */
kopia_wieku = wiek;
```

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

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

Należy go zapisać tak:

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

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

Zasięg zmiennej

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

```
#include <stdio.h>
int a,b; /* nasze zmienne globalne */

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

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

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

Zmienne, które funkcja deklaruje do "własnych potrzeb" nazywamy **zmiennymi lokalnymi**. Nasuwa się pytanie: czy będzie błędem nazwanie tą samą nazwą zmiennej globalnej i lokalnej? Odpowiedź brzmi: nie. Natomiast w danej funkcji da się używać tylko jej wersji lokalnej. Tej konstrukcji należy unikać:

```
int a=1; /* zmienna globalna */
int main()
{
      int a=2; /* to już zmienna lokalna */
      printf("%d", a); /* wypisze 2 */
}
```

Czas życia

Czas życia to czas od momentu przydzielenia dla zmiennej miejsca w pamięci (stworzenie obiektu) do momentu zwolnienia miejsca w pamięci (likwidacja obiektu). Zakres ważności zmiennej to część programu, w której nazwa zmiennej znana jest kompilatorowi.

```
main()
```

```
{
      int a = 10;
      { /* otwarcie lokalnego bloku */
            int b = 10;
            printf("%d %d", a, b);
      } /* zamknięcie lokalnego bloku, zmienna b jest usuwana */
      printf("%d %d", a, b); /* BŁĄD: b juz nie istnieje */
} /* tu usuwana jest zmienna a */
```

Zdefiniowaliśmy dwie zmienne typu `int`. Nazwa zmiennej `a` jest znana kompilatorowi przez cały program. Nazwa zmiennej `b` jest znana tylko w lokalnym bloku, dlatego nastąpi błąd w ostatniej instrukcji.

Niektóre kompilatory uznają powyższy kod za poprawny!

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

```
{
...
}
```

Stałe

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

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

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

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

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

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

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

W przeciwieństwie do języka C++, w C stała to cały czas zmienna, której kompilator pilnuje, by nie zmieniła się.

5.2. Typy zmiennych

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

Typ zmiennej wskazuje właśnie sposób w jaki zawartość pamięci (bity), w której znajduje się zmienna będzie interpretowana. Określając typ przekazuje się komputerowi informację, ile pamięci trzeba zarezerwować dla zmiennej, a także w jaki sposób wykonywać na niej operacje.

Każda zmienna musi mieć określony swój typ w miejscu deklaracji i tego typu nie może już zmienić. Lecz dane, które reprezentuje zmienna możemy **konwertować** (**rzutować**) na inny typ, jeśli jest to potrzebne do wykonanaia jakiejś operacji. Rzutowanie zostanie opisane później, w rozdziale Operatory.

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

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

- `char` − jednobajtowe liczby całkowite; służy do przechowywania znaków,
- `int` − typ całkowity, o długości domyślnej dla danej architektury komputera,
- `float` − typ zmiennopozycyjny (zwany również zmiennoprzecinkowym), reprezentujący liczby rzeczywiste (4 bajty),
- `double` − typ zmiennopozycyjny podwójnej precyzji (8 bajtów).

W języku C nie istnieje specjalny typ zmiennych przeznaczony na zmienne typu logicznego ("prawda"/"fałsz"). Jest to inne podejście niż na przykład w językach Pascal albo Java definiujących osobny typ "boolean", którego nie można mieszać z innymi typami zmiennych. W C do przechowywania wartości logicznych zazwyczaj używa się typu `int`.

Więcej na temat tego, jak język C rozumie prawdę i fałsz, znajduje się w rozdziale Operatory.

int

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

- System dziesiętny

  12, 13, 45, 35

- System ósemkowy (oktalny)

  `010` czyli 8

  `016` czyli 8 + 6 = 14

  `018` BŁĄD

  System ten operuje na cyfrach od 0 do 7. Tak więc 8 jest niedozwolona. Jeżeli chcemy użyć zapisu ósemkowego musimy zacząć liczbę od `0`.

- System szesnastkowy (heksadecymalny)

  `0x10` czyli 1·16 + 0 = 16

  `0x12` czyli 1·16 + 2 = 18

  `0xff` czyli 15·16 + 15 = 255

  W tym systemie możliwe cyfry to 0, 1, ..., 9 i dodatkowo znaki a, b, c, d, e, f, które oznaczają 10, 11, ..., 15. Aby użyć takiego systemu musimy poprzedzić liczbę ciągiem `0x`. Wielkość znaków (np. a lub A) w takich literałach nie ma znaczenia.

Ponadto w niektórych kompilatorach przeznaczonych głównie do mikrokontrolerów spotyka się jeszcze użycie systemu binarnego. Zazwyczaj dodaje się wtedy przedrostek `0b` przed liczbą. W tym systemie możemy oczywiście używać tylko i wyłącznie cyfr 0 i 1. Tego typu rozszerzenie bardzo ułatwia programowanie niskopoziomowe układów. Należy jednak pamiętać, że jest to tylko i wyłącznie rozszerzenie.

Przykłady deklaracji zmiennych typu `int` podano poniżej.

```
int liczbaStudentow, i, j = 5, k = 0x10;
```

float

Ten typ stosowany jest do zapisu liczb rzeczywistych. Istnieją dwa sposoby zapisu:

- System dziesiętny

  3.14, 45.644, 23.54, 3.21

- System "naukowy" − wykładniczy

  `6e2` czyli $6 \cdot 10^2$ czyli 600

  `1.5e3` czyli $1.5 \cdot 10^3$ czyli 1500

  `3.4e-3` czyli $3.4 \cdot 10^{-3}$ czyli 0.0034

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

Reprezentacja zmiennopozycyjna

System zmiennopozycyjny opiera się na notacji naukowej:

$3200000000 = 3{,}2 \cdot 10^9$ (`3.2e9`)

$0{,}00000000000000124 = 1{,}24 \cdot 10^{-15}$ (`1.24e-15`)

W notacji naukowej podajemy kilka cyfr znaczących oraz określamy rząd wielkości poprzez podanie o ile należy przesunąć przecinek w lewo lub w prawo. Przy takim przedstawieniu cyfry znaczące nazywamy **mantysą**, a potęgę podstawy obrazującą, o ile należy przesunąć przecinek dziesiętny – **cechą**.

Ponieważ komputer operuje na bitach, mantysę i cechę wyrażamy binarnie, a jako podstawę potęgi stosujemy 2. Wtedy liczbę *x* wyrażamy w postaci:

$$x = (-1)^z \cdot m \cdot 2^c$$

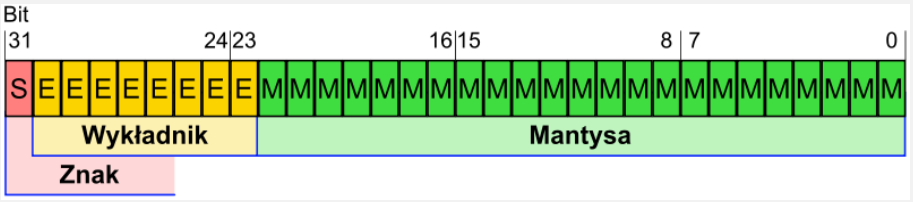
gdzie *z* jest bitem znaku.

Daną liczbę dziesiętną możemy wyrazić na wiele sposobów, np.:

$$3/8 = 3/8 \cdot 2^0 = 3/4 \cdot 2^{-1} = 3/2 \cdot 2^{-2} = 3 \cdot 2^{-3} = 3/16 \cdot 2^1 = \ldots$$

Zgodnie z normą IEEE 754 przyjmuje się, że w kodowaniu binarnym mantysę dobieramy tak, aby jej wartość mieściła się w przedziale [1, 2). Oznacza to, że z powyższych reprezentacji wybieramy tę: $3/2 \cdot 2^{-2}$.

Przyjmuje się, że typ `float` reprezentowany jest na 32 bitach z czego pierwszy to bit znaku, kolejnych 8 to bity wykładnika (cechy), a 23 końcowe to bity mantysy.



Wykładnik zapisywany jest w kodzie binarnym z nadmiarem. W tym kodowaniu oblicza się

wartość dziesiętną liczby zakodowanej w naturalnym kodzie binarnym. Od wartości tej odejmuje się tzw. nadmiar (*bias*), który w naszym przypadku wynosi 127. Najmniejsza wartość wykładnika zakodowana na 8 bitach (00000000) to dziesiętne zero. Po odjęciu nadmiaru otrzymujemy –127. Największa wartość wykładnika (11111111) to 255. Po odjęciu nadmiaru to 128. Największa i najmniejsza wartość wykładnika ma znaczenie specjalne i nie używa się jej do kodowania wykładnika. Wykładnik może więc przyjmować wartości całkowite z zakresu [–126, 127].

Mantysę zapisuje się bez części całkowitej, która zawsze wynosi 1. Zapisujemy jedynie część ułamkową mantysy. Dziesiętną wartość części ułamkowej obliczamy stosując dekodowanie z naturalnego kodu binarnego:

$b_{22}\cdot 2^{-1} + b_{21}\cdot 2^{-2} + b_{20}\cdot 2^{-3} + ... + b_0\cdot 2^{-23}$, gdzie b_i to bit i-ty od końca

Przykład

Postać zakodowana: 0 0100 0100 111 1100 1010 0010 0111 1100

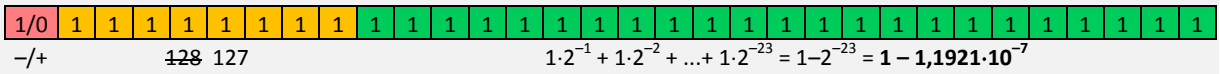
0 – bit znaku: +

0100 0100 – cecha: 68–127= –59

111 1100 1010 0010 0111 1100 – mantysa: $1\cdot 2^{-1} + 1\cdot 2^{-2} + ...+ 0\cdot 2^{-23} = 1/2+1/4+...+0 = 0{,}816806010...$

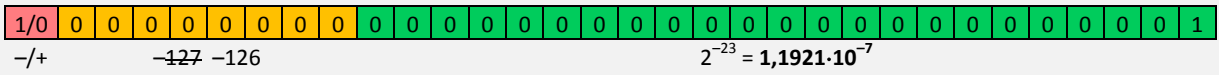
Liczba zdekodowana: $1{,}816806010\cdot 2^{-59} \approx 3.1517e{-}18$

Zakres liczb wyrażony przez typ `float` wynosi:



$+/- (1 + 1 - 1{,}1921\cdot 10^{-7})\cdot 2^{127} = +/- 3{,}4028\cdot 10^{38}$

Dokładność liczb wyrażonych przez typ `float` wynosi (o ile różnią się dwie sąsiednie liczby):



$+/- (1 + 1{,}1921\cdot 10^{-7})\cdot 2^{-126} = +/- 1{,}1755\cdot 10^{-38}$

double

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

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

```
1.5f (float)
```

```
1.5 (double)
```

Przykłady deklaracji zmiennych typu `double` i `float` podano poniżej.

```
double temperatura, a, b = 3.56e-5;
float masaAtomowa, x, y = 6.022e+23;
```

char

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

```
'a', '7', '!', '$'
```

Pojedynczy cudzysłów ' jako stałą typu `char` zapisujemy tak: `\'`, a `null` (tzw. literał pusty, który między innymi kończy napisy) tak: `\0`. Więcej znaków specjalnych:

- `\a` - alarm (sygnał akustyczny terminala)
- `\b` - backspace (usuwa poprzedzający znak)
- `\f` - wysunięcie strony (np. w drukarce)
- `\r` - powrót kursora (karetki) do początku wiersza
- `\n` - znak nowego wiersza
- `\"` - cudzysłów
- `\'` - apostrof
- `\\` - ukośnik wsteczny (backslash)
- `\t` - tabulacja pozioma
- `\v` - tabulacja pionowa
- `\?` - znak zapytania (pytajnik)
- `\ooo` – znak o kodzie `ooo` w systemie oktalnym (ósemkowym), gdzie "`ooo`" należy zastąpić trzycyfrową liczbą w tym systemie
- `\xhh` – znak o kodzie `hh` w systemie heksadecymalnym (szesnastkowym), gdzie "`hh`" należy zastąpić dwucyfrową liczbą w tym systemie
- `\unnnn` - znak o kodzie `nnnn`, gdzie "`nnnn`" należy zastąpić czterocyfrowym identyfikatorem znaku w systemie szesnastkowym, np. `\u0041` to znak `'A'`

- `\unnnnnnnn` - znak o kodzie `nnnnnnnn`, gdzie "`nnnnnnnn`" należy zastąpić ośmiocyfrowym identyfikatorem znaku w systemie szesnatkowym.

Tablica ASCII (*American Standard Code for Information Interchange*):

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

128	Ç	144	É	160	á	176	▒	192	└	208	╨	224	α	240	≡			
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±			
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥			
131	â	147	ô	163	ú	179	|	195	├	211	╙	227	π	243	≤			
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠			
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡			
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷			
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈			
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°			
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙			
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·			
139	ï	155	¢	171	½	187	╗	203	╦	219	▉	235	δ	251	√			
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	n			
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²			
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■			
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255				

Source: www.LookupTables.com

Przykłady deklaracji zmiennych typu `char`:

```
char letter1 = 'a', znak = 97, c = \x61;
```

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

Przykładem traktowania typu `char` jak `int` są poniższe programy.

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

main()
{
  for (int i=0;i<256;i++)
  {
    printf("%3d-%1c ",i,i);
    if (i%12==0)
      printf("\n");
  }
  getch();
  return 0;
}
```

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

main()
{
  char znak,poprz;

  while ((znak=getch())!=poprz)
  {
    printf("%3d,%3o,%3x-\'%c\'\n",znak,znak,znak,znak);
    poprz=znak;
  }
  return 0;
}
```

void

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

5.3. Specyfikatory

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

signed i unsigned

Na początku zastanówmy się, jak komputer może przechować liczbę ujemną. Otóż w przypadku przechowywania liczb ujemnych musimy w kodzie zmiennej przechować jeszcze jej znak. Jak wiadomo, kod zmiennej składa się z szeregu bitów. Pierwszy bit z lewej strony (nazywany także bitem najbardziej znaczącym) przechowuje znak liczby (wartość 1 tego bitu odpowiada liczbie ujemnej, natomiast 0 – liczbie dodatniej). Efektem tego jest spadek "pojemności" zmiennej, czyli zmniejszenie największej wartości, którą możemy przechować w zmiennej.

Przykład – interpretacja liczby zapisanej w tzw. kodzie uzupełnień do dwóch (U2)

Wartość dziesiętną liczby zapisanej w kodzie U2 wyraża wzór:

$$-a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i$$

gdzie: n to liczba bitów, a a_i to wartość i-tego bitu (uwaga: bity numerujemy od prawej do lewej, od 0 do n–1; skrajny lewy bit nazywamy **najstarszym**, a skrajny prawy – **najmłodszym**).

$11101101_{U2} = -1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = -128 + 109 = -19$

$01101101_{U2} = -0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 109$

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

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

Przykładowo dla zmiennej `char` (zajmującej 8 bitów zapisanej w formacie U2) wygląda to tak:

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

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

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

Liczby bez znaku pozwalają nam zapisać większe liczby przy tej samej wielkości zmiennej, ale trzeba uważać, by nie zejść z nimi poniżej zera. Wtedy "przewijają" się na sam koniec

zakresu, co może powodować trudne do wykrycia błędy w programach. Pokazuje to poniższy program.

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

main()
{
      unsigned char znak=10;

      for(int i=10; i>-5; --i)
      {
            printf("%d ",znak);
            --znak;
      }

      return 0;
}
```

short i long

`Short` i `long` są wskazówkami dla kompilatora, by zarezerwował dla danego typu mniej (`short`) lub więcej (`long`) pamięci. `short` i `long` mogą być zastosowane do typu `int`, dodatkowo `long` możemy zastosować do typu `double`. Jeśli przy `short` lub `long` nie napiszemy, o jaki typ nam chodzi, kompilator przyjmie wartość domyślną, która jest typ `int`.

Należy pamiętać, że te specyfikatory wyrażają jedynie życzenie wobec kompilatora − w wielu kompilatorach typy `int` i `long int` mają ten sam rozmiar. Standard języka C nakłada jedynie na kompilatory następujące ograniczenia:

- `int` − nie może być krótszy niż 16 bitów,
- `int` − musi być dłuższy lub równy `short`, a nie może być dłuższy niż `long`,
- `short int` − nie może być krótszy niż 16 bitów,
- `long int` − nie może być krótszy niż 32 bity.

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

5.4. Modyfikatory***

volatile

`volatile` informuje kompilator, że wartość zmiennej może być zmodyfikowana poza programem, przez zewnętrzny proces. Użycie tego modyfikatora jest konieczne w przypadku pewnych programów kompilowanych z opcją optymalizacji. Modyfikator `volatile` jest rzadko stosowany i przydaje się w wąskich zastosowaniach, jak współbieżność i współdzielenie zasobów oraz przerwania systemowe.

register

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

```
register int liczba;
```

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

static

Pozwala na zdefiniowanie zmiennej statycznej. „Statyczność” polega na zachowaniu wartości pomiędzy kolejnymi definicjami tej samej zmiennej. Jest to przede wszystkim przydatne w funkcjach. Gdy zdefiniujemy zmienną w ciele funkcji (lokalną), to zmienna ta będzie od nowa definiowana wraz z domyślną wartością (jeżeli taką podano) przy kolejnych wywołaniach funkcji. W wypadku zmiennej określonej jako statyczna, jej wartość jest „chroniona” i przy ponownym wywołaniu funkcji będzie taka sama, jak przy zakończeniu poprzedniego wywołaniu tej funkcji. Na przykład:

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

Gdy wywołamy tę funkcję trzykrotnie w ten sposób:

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

to ujrzymy na ekranie:

```
Wartosc zmiennej 3
Wartosc zmiennej 5
Wartosc zmiennej 4
```

Jeżeli jednak deklarację zmiennej zmienimy na `static int zmienna = 0`, to wartość zmiennej zostanie zachowana (nie będzie zerowana) i po podobnym wykonaniu funkcji powinnyśmy ujrzeć:

```
Wartosc zmiennej 3
Wartosc zmiennej 8
Wartosc zmiennej 12
```

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

extern

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

auto

Zupełnym archaizmem jest modyfikator `auto`, który oznacza tyle, że zmienna jest lokalna. Ponieważ zmienna zadeklarowana w dowolnym bloku zawsze jest lokalna, modyfikator ten nie ma obecnie żadnego zastosowania praktycznego.

6. OPERATORY

6.1. Przypisanie

Operator przypisania (=) przypisuje wartość prawego argumentu lewemu, np.:

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

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

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

6.2. Skrócony zapis

C umożliwia też skrócony zapis postaci `a ♦= b`, gdzie ♦ jest jednym z operatorów:

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

Ogólnie rzecz ujmując zapis `a ♦= b;` jest równoważny zapisowi `a = a ♦ (b);`, np.:

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

6.3. Rzutowanie

Zadaniem rzutowania jest konwersja danych z jednego typu na inny typ. Konwersja może być **niejawna** (domyślna konwersja wykonywana automatycznie przez kompilator) lub **jawna** (podana explicite przez programistę). Oto kilka przykładów konwersji niejawnej:

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

Podczas konwersji danych musimy liczyć się z utratą informacji, jak to miało miejsce w pierwszej linijce – zmienna `int` nie może przechowywać części ułamkowej, toteż została ona odcięta i w rezultacie zmiennej została przypisana wartość 42.

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

Do jawnego wymuszenia konwersji służy jednoargumentowy operator rzutowania postaci **`(nazwa_typu)`**, np.:

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

W pierwszym przypadku operator został użyty, by zwrócić uwagę na utratę precyzji. W drugim, dlatego że bez niego operator przesunięcia bitowego zachowuje się trochę inaczej. Obie konwersje przedstawione powyżej są dopuszczane przez standard jako jawne konwersje (tj. konwersja z `double` do `int` oraz z `int` do `unsigned int`).

Niektóre konwersje są niedopuszczalne w trybie niejawnym, np.:

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

W takich sytuacjach można użyć operatora rzutowania by wymusić konwersję:

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

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

W języku C++ wprowadzony został dodatkowo inny sposób zapisu rzutowania, zwany rzutowaniem w stylu funkcyjnym:

```
int pi = int(d);
```

6.4. Operatory arytmetyczne

W arytmetyce komputerowej nie działa prawo łączności oraz rozdzielności.

Działanie $\blacklozenge$ w zbiorze S jest łączne, jeżeli $\forall a,b,c \in S$ zachodzi:

$$a \blacklozenge b \blacklozenge c = a \blacklozenge (b \blacklozenge c) = (a \blacklozenge b) \blacklozenge c$$

Działanie $\otimes$ jest rozdzielne względem działania $\oplus$, jeżeli $\forall a,b,c \in S$ zachodzą równości:

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$

$$(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$$

Wynika to z ograniczonego rozmiaru zmiennych, które przechowują wartości. Przykład dla zmiennych o długości 16 bitów (bez znaku): Maksymalna wartość, którą może przechowywać ten typ to: $2^{16}-1 = 65535$. Zatem operacja `65530+10-20` zapisana jako `(65530+10)-20` może zaowocować czymś zupełnie innym niż `65530+(10-20)`. W pierwszym przypadku zapewne dojdzie do tzw. przepełnienia – procesor nie będzie miał miejsca, aby zapisać dodatkowy bit. Zachowanie programu będzie w takim przypadku zależało od architektury procesora. Analogiczny przykład możemy podać dla rozdzielności mnożenia względem dodawania.

Język C definiuje następujące dwuargumentowe operatory arytmetyczne:

- dodawanie (+),
- odejmowanie (−),
- mnożenie (*),
- dzielenie (/),

- reszta z dzielenia (%) określona tylko dla liczb całkowitych (tzw. dzielenie **modulo**).

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

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

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

wypisze (wbrew oczekiwaniu początkujących programistów) 3.0, a nie 3.5. Odnosi się to nie tylko do dzielenia, ale także mnożenia, np.:

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

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

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

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

Inkrementacja i dekrementacja

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

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

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

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

```
c = --b; /* po operacji b=2 c=2 */
```

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

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

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

6.5. Operacje bitowe

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

- negacja bitowa (~),
- koniunkcja bitowa (&),
- alternatywa bitowa (|) i
- alternatywa rozłączna (XOR) (^).

Działają one na poszczególnych bitach, przez co mogą być szybsze od innych operacji. Działanie tych operatorów pokazano w poniższych tabelach.

a	b	&	|	^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Działanie	Wartość binarna	Wartość dziesiętna
a	0101	5

b	0011	3
~a	1010	10
~b	1100	12
a & b	0001	1
a | b	0111	7
a ^ b	0110	6

Przy okazji warto zauważyć, że `a^b^b` to po prostu `a`. Właściwość ta została wykorzystana w różnych algorytmach szyfrowania oraz funkcjach haszujących.

Przesunięcie bitowe

Dodatkowo, język C wyposażony jest w operatory przesunięcia bitowego w lewo (<<) i w prawo (>>). Przesuwają one w danym kierunku bity lewego argumentu o liczbę pozycji podaną jako prawy argument. Rozważmy 4-bitowe liczby bez znaku (taki hipotetyczny `unsigned int`), wówczas:

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

Widać, że bity skrajne są tracone, a w puste miejsca wpisywane są zera.

Jeśli argument po lewej stronie instrukcji (`a`) jest liczbą ze znakiem, to operacja przesunięcia w lewo zachowuje się tak samo jak dla liczb bez znaku, natomiast przy przesuwaniu w prawo bit znaku nie zmienia się:

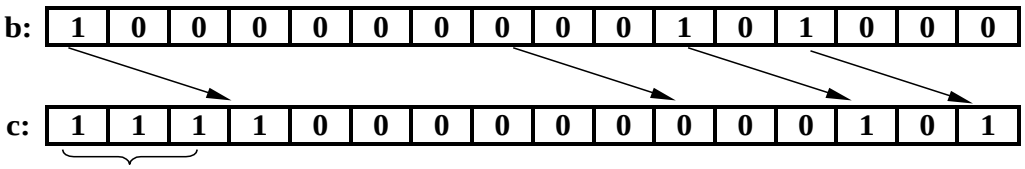
a	a>>1	a>>2
0001	0000	0000

0011	0001	0000
0101	0010	0001
1000	1100	1110
1111	1111	1111
1001	1100	1110

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

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

signed a = - 32728, b = 3, c; //a = $-2^{15} + 2^5 + 2^3$
c = a >> b;



:upełnienie bitami znaku (1 – minus, 0 – plus); c = $-2^{15}+2^{14}+2^{13}+2^{12}+2^2+2^0$ = - 4091

6.6. Porównanie

W języku C występują następujące operatory porównania:

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

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

Częste błędy

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

Porównajmy ze sobą dwa warunki:

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

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

W celu uniknięcia takich błędów niektórzy programiści zamiast pisać `a == 1` piszą `1 == a`.

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

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

6.7. Operatory logiczne

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

- negację (zaprzeczenie): !
- koniunkcję (i): &&
- alternatywę (lub): ||

Działają one bardzo podobnie do operatorów bitowych, jednak zamiast operować na poszczególnych bitach, biorą pod uwagę **wartość logiczną** argumentów.

Prawda i fałsz w języku C

Język C nie przewiduje specjalnego typu danych do operacji logicznych − operatory logiczne można stosować do liczb (np. typu `int`), tak samo jak operatory bitowe albo arytmetyczne.

Wyrażenie ma wartość logiczną 0 wtedy i tylko wtedy, gdy jest równe 0 (jest fałszywe). W przeciwnym wypadku ma wartość 1 (jest prawdziwe). Operatory logiczne w wyniku dają zawsze albo 0 albo 1.

Żeby w pełni uzmysłowić sobie, co to oznacza, spójrzmy na wynik wykonania poniższych trzech imstrukcji:

```
printf("koniunkcja: %d\n", 18 && 19);
printf("alternatywa: %d\n", 'a' || 'b');
printf("negacja: %d\n", !20);
```

```
koniunkcja: 1
alternatywa: 1
negacja: 0
```

Liczba 18 nie jest równa 0, więc ma wartość logiczną 1. Podobnie 19 ma wartość logiczną 1. Dlatego ich koniunkcja jest równa 1. Znaki `'a'` i `'b'` zostaną w wyrażeniu logicznym potraktowane jako liczby o wartościach odpowiadających kodom ASCII tych znaków − czyli oba będą miały wartość logiczną 1.

Skrócone obliczanie wyrażeń logicznych

Język C wykonuje skrócone obliczanie wyrażeń logicznych − to znaczy, oblicza wyrażenie tylko tak długo, gdy nie wie, jaka będzie jego ostateczna wartość. To znaczy, idzie od lewej do prawej obliczając kolejne wyrażenia (dodatkowo na kolejność wpływ mają nawiasy) i gdy będzie miał na tyle informacji, by obliczyć wartość całości, nie liczy reszty. Może to wydawać się niejasne, ale przyjrzyjmy się wyrażeniom logicznym (`A` i `B` symbolizują pewne warunki np. `x>=5`, `y!=b`, ...):

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

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

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

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

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

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

6.8. Operator wyrażenia warunkowego

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

```
w1 ? w2 : w3
```

Jego działanie wygląda następująco: najpierw oceniana jest wartość logiczna wyrażenia `w1`. Jeśli jest ono prawdziwe, to zwracana jest wartość wyrażenia `w2`, jeśli natomiast wyrażenie `w1` jest nieprawdziwe, zwracana jest wartość wyrażenia `w3`.

Praktyczne zastosowanie – znajdowanie większej z dwóch liczb:

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

lub zwracanie modułu liczby:

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

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

6.9. Przecinek

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

6.10. Operator sizeof

Operator `sizeof` zwraca rozmiar w bajtach (gdzie bajtem jest zmienna typu `char`) podanego typu lub typu podanego wyrażenia. Przyjmuje jedną z dwóch postaci:

- `sizeof(typ)`

- `sizeof wyrażenie`

Przykładowo:

```
#include <stdio.h>
int main()
{
      printf("sizeof(short) = %d\n", sizeof(short));
      printf("sizeof(int) = %d\n", sizeof(int));
      printf("sizeof(long) = %d\n", sizeof(long));
      printf("sizeof(float) = %d\n", sizeof(float));
      printf("sizeof(double) = %d\n", sizeof(double));
      printf("sizeof(long double) = %d\n", sizeof(long double));
      return 0;
}
```

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

6.11. Inne operatory

Poza wyżej opisanymi operatorami istnieją jeszcze:

- operator „[]” opisany przy okazji opisywania tablic,
- jednoargumentowe operatory „*” i „&” opisane przy okazji opisywania wskaźników,
- operatory „.” i „–>” opisywane przy okazji opisywania struktur i unii,
- operator „( )” będący operatorem wywołania funkcji,
- operator „( )” grupujący wyrażenia (np. w celu zmiany kolejności obliczania)

6.12. Priorytety i kolejność obliczeń

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

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

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

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

Gdy chcemy zmienić kolejność wykonywania działań, który wynika z priorytetu operatorów, stosujemy nawiasy, np:

```
2*(5/(3+2)+(6-2)/(9%5))
```

A jaki wynik da to wyrażenie pisane bez nawiasów?

Operatory można "mieszać" w jednej instrukcji, np.:

```
(11%(1<<3)) & ((0||1)?2*3:(2+3))
```

6.13. Kolejność wyliczania argumentów operatora

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

```
#include <stdio.h>

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

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

Otóż, nie wiemy czy najpierw zostanie wywołana funkcja `foo` z parametrem `1`, czy `2`. Jeżeli ma to znaczenie należy użyć zmiennych pomocniczych zmieniając definicję funkcji `main()` na:

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

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

```
#include <stdio.h>

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

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

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

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

7. INSTRUKCJE STERUJĄCE

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

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

7.1. Instrukcje warunkowe

Instrukcja if

Definicja instrukcji `if` jest następujaca:

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

Istnieje także możliwość reakcji na nieprawdziwość wyrażenia − wtedy należy zastosować słowo kluczowe `else`:

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

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

```
#include <stdio.h>

 int main ()
 {
      int a, b;
      a = 4;
      b = 6;
      if (a==b) {
            printf ("a jest równe b\n");
      } else {
            printf ("a nie jest równe b\n");
      }
      return 0;
 }
```

Stosowany jest też krótszy zapis warunków logicznych, wykorzystujący to, jak C rozumie prawdę i fałsz. Jeśli zmienna `a` jest typu `int`, zamiast:

```
if (a != 0)
```

można napisać:

```
if (a)
```

a zamiast

```
if (a == 0)
```

można napisać:

```
if (!a)
```

Czasami zamiast pisać instrukcję `if` możemy użyć operatora wyrażenia warunkowego (patrz Operatory).

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

ma dokładnie taki sam efekt jak:

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

Instrukcja switch

Aby ograniczyć wielokrotne stosowanie instrukcji `if` możemy użyć `switch`. Jej definicja jest następująca:

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

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

```
#include <stdio.h>
int main ()
{
      int a, b;
      printf ("Podaj a: ");
      scanf ("%d", &a);
      printf ("Podaj b: ");
      scanf ("%d", &b);
      switch (b) {
            case 0: printf ("Nie można dzielić przez 0!\n"); /* tutaj
                                                  zabrakło break! */
            default: printf ("a/b=%d\n", a/b);
      }
      return 0;
}
```

Pominiecie `break` czasami może być celowym zabiegiem − wówczas warto zaznaczyć to w komentarzu. Oto przykład:

```
#include <stdio.h>
int main ()
{
      int a = 4;
      switch ((a%3)) {
            case 0:
                  printf ("Liczba %d dzieli się przez 3\n", a);
                  break;
            case -2:
            case -1:
            case 1:
            case 2:
                  printf ("Liczba %d nie dzieli się przez 3\n", a);
```

```
                  break;
      }
      return 0;
}
```

Przeanalizujmy teraz przykład:

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

7.2. Pętle

Instrukcja while

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

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

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

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

Po analizie kodu mogą nasunąć się dwa pytania:

- Po co zwiększać wartość `a` o jeden? Otóż gdybyśmy nie dodali instrukcji zwiększającej `a`, to warunek zawsze byłby spełniony, a pętla "kręciłaby" się w nieskończoność.

- Dlaczego warunek to `a <= 10` a nie `a != 10`? Odpowiedź jest dość prosta. Pętla sprawdza warunek przed wykonaniem kolejnego "obrotu". Dlatego też gdyby warunek brzmiał `a != 10` to dla `a = 10` jest on nieprawdziwy i pętla nie wykonałaby ostatniej iteracji.

Instrukcja for

Od instrukcji `while` czasami wygodniejsza jest instrukcja `for`. Umożliwia ona zapis w jednej linijce: ustawienia początkowej wartości zmiennej sterującej (tzw. **licznika**), warunku zatrzymania pętli i sposobu zmiany licznika w kolejnych iteracjach. Instrukcję `for` stosuje się w następujący sposób:

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

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

- `wyrażenie1` – jest to instrukcja, która będzie wykonana przed pierwszym przebiegiem pętli. Zwykle jest to inicjalizacja zmiennej, która będzie służyła jako licznik przebiegów pętli.

- `wyrażenie2` – jest warunkiem zakończenia pętli. Pętla wykonuje się tak długo, jak prawdziwy jest ten warunek.

- `wyrażenie3` – jest to instrukcja, która wykonywana będzie po każdym przejściu pętli. Zamieszczone są tu instrukcje, które zmieniają licznik o odpowiednią wartość.

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

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

W pierwszej kolejności w pętli `for` wykonuje się `wyrażenie1`. Wykonuje się ono zawsze, nawet jeżeli warunek przebiegu pętli jest od samego początku fałszywy. Po wykonaniu `wyrażenie1` pętla `for` sprawdza warunek zawarty w `wyrażenie2`. Jeżeli jest on prawdziwy, to wykonywana jest treść pętli `for`, czyli najczęściej to co znajduje się między klamrami, lub gdy ich nie ma, następna pojedyncza instrukcja. W szczególności musimy pamiętać, że sam średnik też jest instrukcją – **instrukcją pustą**. Gdy już zostanie wykonana treść pętli

`for`, następuje wykonanie `wyrażenie3`. Należy zapamiętać, że `wyrażenie3` zostanie wykonane, nawet jeżeli był to już ostatni obieg pętli. Następnie sprawdzane jest `wyrażenie2`, jeśli nadal jest prawdziwe, wykonywana jest następna iteracja pętli.

Poniższe trzy przykłady pętli `for` w rezultacie dadzą ten sam wynik. Wypiszą na ekran liczby od 1 do 10.

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

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

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

Ostatni przykład jest bardziej wyrafinowany i korzysta z tego, że jako `wyrażenie3` może zostać podane dowolne bardziej skomplikowane wyrażenie, zawierające w sobie inne podwyrażenia.

A oto kolejny przykład użycia instrukcji `for`, która najpierw wyświetla liczby w kolejności rosnącej, a następnie malejącej.

```
#include <stdio.h>
int main()
{
      int i;
      for(i=1; i<=5; ++i){
            printf("%d", i);
      }
      for( ; i>=1; --i){
            printf("%d", i);
      }
      return 0;
}
```

Wynikiem działania powyższego programu będzie ciąg cyfr `12345654321`. Pierwsza pętla wypisze cyfry od 1 do 5 i po ostatnim swoim obiegu zinkrementuje zmienną `i`. Gdy druga pętla przystąpi do pracy, zacznie ona odliczać począwszy od liczby i = 6, a nie 5.

Następny przykład oblicza wartości kwadratów liczb od 1 do 10.

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

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

W języku C++ można deklarować zmienne w nagłówku pętli `for` w następujący sposób:

```
for(int i=0; i<10; ++i)
```

Instrukcja do..while

Pętle `while` i `for` mają wspólna cechę – może się zdarzyć, że nie wykonają się ani razu. Aby mieć pewność, że nasza pętla będzie miała co najmniej jeden przebieg musimy zastosować pętlę `do..while`. Wygląda ona następująco:

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

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

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

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

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

Instrukcja break

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

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

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

Break i pętle nieskończone

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

```
for (;;) { /* instrukcje */ }
for (;1;) { /* instrukcje */ }
for (a;a;a) { /* instrukcje */ } /*gdzie a jest dowolną liczbą
                                   różną od 0*/
while (1) { /* instrukcje */ }
do { /* instrukcje */ } while (1);
```

Takie pętle, nazywane pętlami nieskończonymi (z racji tego, że warunek pętli zawsze jest prawdziwy) przerwać może instrukcja `break`.

Wszystkie fragmenty kodu działają identycznie:

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

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

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

Instrukcja continue

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

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

Dla wartości `i` większej od 40 nie będzie wyświetlany komunikat "Koniec", chociaż pętla wykona 100 iteracji.

Inny praktyczny przykład użycia tej instrukcji:

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

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

7.3. Instrukcja goto

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

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

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

Przykład uzasadnionego użycia:

```
int i,j;
for (i = 0; i < 10; ++i) {
```

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

7.4. Natychmiastowe zakończenie programu – funkcja exit

Program może zostać w każdej chwili zakończony − do tego właśnie celu służy funkcja `exit()`. Używamy jej następująco:

```
exit (kod_wyjścia);
```

Liczba całkowita `kod_wyjścia` jest przekazywana do procesu macierzystego, dzięki czemu dostaje on informację, czy program w którym wywołaliśmy tę funkcję zakończył się poprawnie. Kody wyjścia są nieustandaryzowane i żeby program był w pełni przenośny należy stosować makra `EXIT_SUCCESS` i `EXIT_FAILURE`, choć w wielu systemach kod 0 oznacza poprawne zakończenie, a kod różny od 0 błędne.

8. PODSTAWOWE PROCEDURY WEJŚCIA I WYJŚCIA

8.1. Wejście/wyjście

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

Takie wczytywanie i "wyrzucanie" danych w terminologii komputerowej nazywamy wejściem (input) i wyjściem (output). W C do komunikacji z użytkownikiem służą odpowiednie funkcje wyjścia i wejścia. Gdy piszemy o jakiejś funkcji, podając jej nazwę dopisujemy na końcu nawias, np.: `printf()`, `scanf()`, żeby było jasne, że chodzi właśnie o funkcję.

Wyżej wymienione funkcje to jedne z najczęściej używanych funkcji w C − pierwsza służy do wypisywania danych na standardowym urządzeniu wyjściowym (zazwyczaj jest to ekran, ale może to być plik lub drukarka), natomiast druga – do wczytywania danych ze standardowego urządzenia wejściowego (zazwyczaj klawiatury, ale może to być plik).

8.2. Funkcje wyjścia

Funkcja `printf()`

W przykładzie "Hello World!" użyliśmy już jednej z dostępnych funkcji wyjścia, a mianowicie funkcji `printf()`. Z punktu widzenia swoich możliwości jest to jedna z bardziej skom-

plikowanych funkcji, a jednocześnie jest jedną z najczęściej używanych. Przyjrzyjmy się ponownie kodowi programu "Hello, World!".

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

Po skompilowaniu i uruchomieniu, program wypisze na ekranie: "Hello world!". Tekst wypisywany na ekranie umieszczamy w cudzysłowach wewnątrz nawiasów. Ogólnie, wywołanie funkcji `printf()` wygląda następująco:

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

Przykładowo:

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

wypisze

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

Format to napis ujęty w cudzysłowy, który określa ogólny kształt, schemat tego, co ma być wyświetlone. Format jest drukowany tak, jak go napiszemy, jednak niektóre znaki specjalne zostaną w nim podmienione na coś innego. Przykładowo, znak specjalny `\n` jest zamieniany na znak nowej linii (zamiana ta następuje w momencie kompilacji programu). Natomiast procent jest podmieniany na jeden z argumentów (według ich kolejności). Po procencie następuje specyfikacja, jak wyświetlić dany argument. W tym przykładzie `%d` (od *decimal*) oznacza, że argument ma być wyświetlony jako liczba całkowita.

Lista argumentów zawiera zmienne, stałe lub wyrażenia, których wartości będą podstawione za kolejne znaki `%` w formacie. Na liście argumentów możemy mieszać ze sobą argumenty różnych typów, lecz należy zwrócić uwagę, aby po znaku procentu w formacie pojawiła się specyfikacja zgodna z typem odpowiedniego argumentu.

Najczęstsze użycie `printf()`:

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

Łańcuch formatujący (`format`) może być podany jako zmienna. W związku z tym możliwa jest np. taka konstrukcja:

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

Pełna specyfikacja funkcji `printf()`: http://pl.wikibooks.org/wiki/C/printf

Funkcja `puts()`

Funkcja `puts()` przyjmuje jako swój argument ciąg znaków, który następnie wypisuje na ekran kończąc go znakiem przejścia do nowej linii. Nasz pierwszy program moglibyśmy napisać w ten sposób:

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

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

Pełna specyfikacja funkcji `puts()`: http://pl.wikibooks.org/wiki/C/puts

Funkcja `fputs()`

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

```
#include <stdio.h>
int main(void)
{
      fputs("Hello world!", stdout);
      return 0;
}
```

`stdout` jest to określenie strumienia wyjściowego (w naszym wypadku standardowe wyjście – standard output).

Pełna specyfikacja funkcji `fputs()`: http://pl.wikibooks.org/wiki/C/fputs

Funkcja `putchar()`

Funkcja `putchar()` służy do wypisywania pojedynczych znaków. Przykład użycia w programie wypisującym w prostej tabelce wszystkie liczby od 0 do 99:

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

Inny przykład użycia `putchar()`:

```
#include <stdio.h>

int main()
{
      char c;
      for (c = 'a'; c <= 'f'; ++c)
            putchar (c);
      return 0;
}
```

Pełna specyfikacja funkcji `putchar()`: http://pl.wikibooks.org/wiki/C/putchar

8.3. Funkcje wejścia

Funkcja `scanf()`

Funkcja `scanf()` jako metoda wczytywania danych jest odpowiednikiem funkcji `printf()`. W poniższym przykładzie program podaje kwadrat liczby, podanej przez użytkownika:

```
#include <stdio.h>
int main ()
{
      int liczba = 0;
      printf ("Podaj liczbę: ");
      scanf ("%d", &liczba);
      printf ("%d*%d=%d\n", liczba, liczba, liczba*liczba);
      return 0;
}
```

W funkcji `scanf()` przy zmiennej pojawił się nowy operator – `&` (etka). Bez niego funkcja `scanf()` nie skopiuje odczytanej wartości liczby do odpowiedniej zmiennej! Właściwie oznacza przekazanie do funkcji adresu zmiennej, by funkcja mogła zmienić jej wartość. Zostanie to wyjaśnione w rozdziale Wskaźniki.

Oznaczenia są podobne jak przy `printf()`, czyli `scanf("%d", &liczba);` wczytuje liczbę typu `int`, `scanf("%f", &liczba);` – liczbę typu `float`, a `scanf("%s", tablica_znaków);` – ciąg znaków.

W tym ostatnim przypadku nie ma etki. Otóż, gdy podajemy jako argument do funkcji wyrażenie typu tablicowego zamieniane jest ono automatycznie na adres pierwszego elementu tablicy. Będzie to dokładniej opisane w rozdziale poświęconym wskaźnikom.

Brak etki jest częstym błędem szczególnie wśród początkujących programistów. Ponieważ funkcja `scanf()` akceptuje zmienną liczbę argumentów to nawet kompilator może mieć kłopoty z wychwyceniem takich błędów.

Rozważmy poniższy kod:

```
#include <stdio.h>
int main(void)
{
      char tablica[100]; /* 1 */
      scanf("%s", tablica); /* 2 */
      return 0;
}
```

W linijce 1 deklarujemy tablicę 100 znaków czyli mogącą przechować napis długości 99 znaków (jeden element tej tablicy przechowuje znak końca łańcucha `'\0'`). W linijce 2 wywołujemy funkcję `scanf()`, która odczytuje tekst ze standardowego wejścia. Nie zna ona jednak rozmiaru tablicy i nie wie ile znaków może ona przechować, przez co będzie czytać tyle zna-

ków, aż napotka biały znak (format %s nakazuje czytanie pojedynczego słowa), co może doprowadzić do przepełnienia bufora (czyli pamięci zarezerwowanej dla zmiennej tablica). Niebezpieczne skutki czegoś takiego opisane są w rozdziale poświęconym napisom. Aby ich uniknąć zaraz po znaku procentu należy podawać maksymalną liczbę znaków, które może przechować bufor, czyli liczbę o jeden mniejszą, niż rozmiar tablicy. Linia 2 powinna mieć postać: `scanf("%99s", tablica);`.

Funkcja `scanf()` zwraca liczbę poprawnie wczytanych zmiennych lub `EOF` (*end of file* – znak końca pliku), jeżeli nie ma już danych w strumieniu wejściowym lub nastąpił błąd. Załóżmy dla przykładu, że chcemy stworzyć program, który odczytuje po kolei liczby i wypisuje ich trzecie potęgi. W pewnym momencie dane się kończą lub jest wprowadzana niepoprawna dana i wówczas nasz program powinien zakończyć działanie. Aby tak było, należy sprawdzać wartość zwracaną przez funkcję `scanf()` w warunku pętli:

```
#include <stdio.h>
int main(void)
{
	int n;
	while (scanf("%d", &n)==1)
	{
		printf("%d\n", n*n*n);
	}
	return 0;
}
```

Rozpatrzmy teraz trochę bardziej skomplikowany przykład. Otóż, ponownie jak poprzednio nasz program będzie wypisywał trzecie potęgi podanych liczb, ale tym razem musi ignorować błędne dane (tzn. pomijać ciągi znaków, które nie są liczbami) i kończyć działanie tylko w momencie, gdy nastąpi błąd odczytu lub koniec pliku.

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

Najpierw wywoływana jest funkcja `scanf()` i następuje próba odczytu liczby typu `int`. Jeżeli funkcja zwróciła `1`, to liczba została poprawnie odczytana i następuje wypisanie jej

trzeciej potęgi. Jeżeli funkcja zwróciła `0`, to na wejściu były jakieś dane, które nie wyglądały jak liczba. W tej sytuacji wywołujemy funkcję `scanf()` z formatem odczytującym dowolny ciąg znaków nie będący białymi znakami z jednoczesnym określeniem, żeby nie zapisywała nigdzie wyniku (`result = scanf("%s")`). W ten sposób niepoprawnie wpisana dana jest omijana. Pętla główna wykonuje się tak długo, jak długo funkcja `scanf()` nie zwróci wartości `EOF`.

Pełna specyfikacja funkcji `scanf()`: http://pl.wikibooks.org/wiki/C/scanf.

Funkcja `fgets()` *i* `gets()`

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

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

Ostatni argument to określenie tzw. strumienia, w naszym przypadku standardowe wejście – *standard input*). Funkcja czyta tekst, aż do napotkania znaku przejścia do nowej linii, który zapisuje w wynikowej tablicy (funkcja `gets()` tego znaku nie zapisuje). Jeżeli brakuje miejsca w tablicy to funkcja przerywa czytanie. W ten sposób, aby sprawdzić czy została wczytana cała linia czy tylko jej część należy sprawdzić, czy ostatnim znakiem jest znak przejścia do nowej linii. Jeżeli nastąpił jakiś błąd lub na wejściu nie ma już danych funkcja zwraca wartość `NULL`.

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

Powyższy kod wczytuje dane ze standardowego wejścia – linia po linii – i dodaje na początku każdej linii znak większości, po którym dodaje spację, ale tylko wtedy, gdy pierwszym znakiem w sczytanej linii nie jest znak większości.

W linijce `1` następuje odczytywanie linii. Jeżeli nie ma już więcej danych na wejściu lub nastąpił błąd wejścia, funkcja `fgets()` zwraca `NULL`, który ma logiczną wartość `0` i wówczas

pętla kończy działanie. W przeciwnym wypadku funkcja zwraca po prostu pierwszy argument, który ma wartość logiczną `1`. W linijce `2` sprawdzamy, czy poprzednie wywołanie funkcji wczytało całą linię, czy tylko jej część – jeżeli całą to teraz jesteśmy na początku linii i należy dodać znak większości. W linii `3` wypisujemy linię na standardowe wyjście. W linii `4` przeszukujemy tablicę znak po znaku, aż do momentu, gdy znajdziemy znak o kodzie `0` (znak końca łańcucha '\0') albo znak przejścia do nowej linii. Ten drugi przypadek oznacza, że funkcja `fgets()` wczytała całą linię.

Funkcja `gets()` czyta linię ze standardowego wejścia (usuwa ją stamtąd) i umieszcza ją w tablicy znakowej wskazywanej przez `tablica_znaków`:

```
gets(tablica_znaków)
```

Ostatni znak linii (znak nowego wiersza - '\n') zastępuje zerem (znakiem '\0').

Pełna specyfikacja funkcji `fgets()`: http://pl.wikibooks.org/wiki/C/fgets

i `gets()`: https://pl.wikibooks.org/wiki/C/gets

Funkcja `getchar()` *i* `getc()`

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

```
#include <stdio.h>
int main(void)
{
      int c;
      while ((c = getchar())!=EOF) {
            if (c==' ') {
                  c = '_';
            }
            putchar(c);
            printf("\nPrzeczytano znak o numerze %d.\n", c);
      }
      return 0;
}
```

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

Funkcja `getc(str)` czyta znak ze strumienia `str`. Zwraca kod pierwszego znaku ze strumienia. W przypadku końca pliku lub błędu funkcja zwraca wartość EOF. Instrukcja `c = getchar()` jest równoważna `c = getc(stdin)`.

Pełna specyfikacja funkcji `getchar()`: http://pl.wikibooks.org/wiki/C/getchar i

`getc()`: http://pl.wikibooks.org/wiki/C/getc

9. FUNKCJE

W C funkcja (czasami nazywana podprogramem, rzadziej procedurą) to wydzielona część programu, która przetwarza **argumenty** i ewentualnie **zwraca wartość**. Wartość ta następnie może być wykorzystana jako argument w innych działaniach lub funkcjach. Funkcja może posiadać własne zmienne lokalne.

Poznaliśmy już kilka funkcji, np.: funkcję `printf()`, drukująca tekst na ekranie, czy funkcję `main()`, czyli główną funkcję programu.

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

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

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

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

9.1. Tworzenie funkcji

Rozważmy następujący kod:

```
int iloczyn (int x, int y)
```

```
{
      int iloczyn_xy;
      iloczyn_xy = x*y;
      return iloczyn_xy;
}
```

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

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

Funkcję w języku C tworzy się następująco:

```
typ identyfikator (typ1 argument1, typ2 argument2,..., typ n argument n)
{
      /* instrukcje */
}
```

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

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

Należy odróżnić deklaracje funkcji od jej definicji. **Deklaracja** (**prototyp**) ma postać nagłówka funkcji i informuje kompilator jakich parametrów wymaga funkcja i jakiego typu zwraca wartość. Natomiast **definicja** funkcji zawiera także jej ciało.

Procedury

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

```
void identyfikator (typ1 argument1, typ2 argument2,..., typ_n argument_n)
{
      /* instrukcje */
}
```

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

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

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

9.2. Wywoływanie funkcji

Funkcje wywołuje się następująco:

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

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

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

Programiści mający doświadczenia np. z językiem Pascal mogą popełniać błąd polegający na wywoływaniu funkcji bez nawiasów okrągłych, gdy nie przyjmuje ona żadnych argumentów. Funkcję bez argumentów w C należy wywoływać podając za jej nazwą nawiasy: `identyfikator()`.

Przykładowo, mamy funkcję:

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

Jej wywołanie wygląda tak: `pisz_komunikat();`.

9.3. Zwracanie wartości

Funkcja zwraca wartość za pomocą słowa kluczowego `return`. Słowo to zawsze kończy działanie funkcji. Gdy funkcja nie zwraca żadnej wartości również powinna posiadać w swoim ciele `return`.

Użycie tej instrukcji jest bardzo proste:

```
return zwracana_wartość;
```

lub:

```
return;
```

Jeśli nasza funkcja nie zwraca konkretnej użytecznej wartości, a chcemy zaznaczyć, że jej działanie zakończyło się sukcesem piszemy: `return 0;`. Gdy chcemy zaznaczyć, że dzia-

łanie funkcji zakończyło się niepowodzeniem piszemy: `return 1;`. Ta wartość może być wykorzystana przez inne instrukcje, np. `if` , jak w przykładzie poniżej.

```
#include <stdio.h>

int funkcja1(float a, float b)
{
      if a>=b return 0;
      else return 1;
}

int main(void)
{
      float x,y;

      if !funkcja1(x,y) puts('x jest nie mniejsze od y');
      else puts('x jest mniejsze od y');

      return 0;
}
```

9.4. Funkcja main()

Każdy program w języku C musi zawierać funkcję `main()`. Jest to funkcja, która zostaje wywołana przez fragment kodu inicjującego pracę programu. Kod ten tworzony jest przez kompilator i nie mamy na niego wpływu. Istnieją dwa możliwe **prototypy** (**nagłówki**) omawianej funkcji:

```
int main(void);
int main(int argc, char **argv);
```

Argument `argc` jest liczbą określającą, ile ciągów znaków przechowywanych jest w tablicy `argv`. Pierwszym elementem tablicy `argv` (o ile istnieje) jest nazwa programu. Pozostałe przechowują argumenty podane przy uruchamianiu programu. Ostatni element `argv[argc]` ma zawsze wartość `NULL`.

Jeśli program uruchomimy poleceniem:

```
program argument1 argument2
```

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

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

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

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

```
	puts("sciezka dostepu do programu:");
	puts(argv[0]);
	puts("arumenty:");
	for (int i=1;i<argc;i++)
		puts(argv[i]); //lub printf("%s\n", argv[i]);
	getch();
	return 0;
}
```

Program uruchomiony poleceniem: `nazwa_programu aaa bbb ccc` wypisze:

```
sciezka dostepu do programu:
sciezka/nazwa_programu.exe
argumenty:
aaa
bbb
ccc
```

Funkcja `main()` nie różni się od innych funkcji i tak jak inne może wołać sama siebie (patrz rekurencja niżej), przykładowo powyższy program można zapisać tak:

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

int main(int argc, char **argv) {

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

Typowo funkcja `main()` w przypadku poprawnego zakończenia programu zwraca `0` lub `EXIT_SUCCESS`, natomiast w przypadku zakończenia błędnego zwraca `EXIT_FAILURE`. Makrodefinicje `EXIT_SUCCESS` i `EXIT_FAILURE` zdefinioane są w pliku nagłówkowym `stdlib.h`.

9.5. Dalsze informacje

Jak zwrócić kilka wartości?

Generalnie możliwe są dwa podejścia: jedno to "upakowanie" zwracanych wartości – można stworzyć tak zwaną strukturę, która będzie przechowywała kilka zmiennych (jest to opisane w rozdziale Typy złożone). Prostszym sposobem jest zwracanie jednej z wartości w normalny sposób, a pozostałych jako parametry. Do tego jednak potrzebne są wskaźniki. W dalszej części będzie to dokładniej opisane.

Przekazywanie parametrów

Istnieje kilka sposobów przekazywania argumentów do funkcji. Jednym z nich jest **przekazywanie przez wartość**. W tym przypadku, gdy wywołujemy funkcję, wartość argumentów, z którymi ją wywołujemy jest kopiowana do funkcji. Kopiowana – to znaczy, że w ciele funkcji działamy na kopiach zmiennych przekazanych do funkcji. Wartości zmiennych przekazywanych nie ulegają zmianie (możliwe jest modyfikowanie zmiennych przekazywanych do funkcji jako parametry, ale do tego w C potrzebne są wskaźniki, które poznamy później).

Przekazywanie parametrów przez wartość demonstruje przykład:

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

int fun(int x)
{
      return ++x;
}

main() {

      int a = 10, b;

      b = fun(a); /*przekazanie kopii wartości zmiennej a do funkcji, wartość ta zostanie skopiowana do zmiennej lokalnej x. Po zakończeniu działania funkcji wartość zmiennej a nie zmieni się.*/
      return 0;
}
```

Funkcje rekurencyjne

Język C ma możliwość tworzenia tzw. funkcji rekurencyjnych. Jest to funkcja, która w swojej własnej definicji (ciele) wywołuje samą siebie. Klasycznym przykładem może tu być silnia. Funkcja rekurencyjna, która oblicza silnię może mieć postać:

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

Musimy być ostrożni przy funkcjach rekurencyjnych, gdyż łatwo za ich pomocą utworzyć funkcję, która będzie sama siebie wywoływała w nieskończoność, a co za tym idzie będzie zawieszała program. Tutaj pierwszymi instrukcjami `if` ustalamy "warunki stopu", gdzie kończy się wywoływanie funkcji przez samą siebie, a następnie określamy, jak funkcja będzie

wywoływać samą siebie (odjęcie jedynki od argumentu, co do którego wiemy, że jest dodatni, gwarantuje, że dojdziemy do warunku stopu w skończonej liczbie kroków).

Warto też zauważyć, że funkcje rekurencyjne czasami mogą być znacznie wolniejsze niż podejście nierekurencyjne (iteracyjne, przy użyciu pętli). Flagowym przykładem może tu być funkcja obliczająca wyrazy ciągu Fibonacciego (pierwszy wyraz jest równy 0, drugi jest równy 1, każdy następny jest sumą dwóch poprzednich):

```
#include <stdio.h>

unsigned count;

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

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

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

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

W tym przypadku funkcja rekurencyjna jest bardzo nieefektywna.

Deklarowanie funkcji

Przed wywołaniem funkcji musimy podać jej deklarację – wymaga tego kompilator w celu sprawdzenia czy funkcja wywoływana jest poprawnie. Przeanalizujmy kod:

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

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

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

W tym przypadku obie funkcje `a()` i `b()` korzystają z siebie nawzajem. W funkcji `a()` wywołujemy funkcję `b()`, ale jej definicja `b()` jest podana później. Aby kompilator mógł sprawdzić, czy funkcja `b()` wywołana jest poprawnie, przed definicją funkcji `a()` należy zadeklarować funkcję `b()`. Deklaracja funkcji (zwana prototypem) to po prostu przekopiowana pierwsza linijka funkcji (przed otwierającym nawiasem klamrowym) z dodatkowo dodanym średnikiem na końcu. W naszym przykładzie deklaracja ma postać:

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

W deklaracji można pominąć nazwy parametrów funkcji:

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

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

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

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

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

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

Dołączane na początku programu pliki nagłówkowe zawierają właśnie prototypy funkcji i ułatwiają pisanie dużych programów.

Zmienna liczba parametrów

Używając funkcji `printf()` lub `scanf()` po argumencie zawierającym tekst z odpowiednimi modyfikatorami można podać praktycznie nieograniczoną liczbę argumentów. Deklaracje obu funkcji są następujące:

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

W deklaracji zostały użyte trzy kropki dla oznaczenia zmiennej liczby argumentów. Język C ma możliwość przekazywania teoretycznie nieograniczonej liczby argumentów do funkcji (jedynym ograniczeniem jest rozmiar stosu programu).

Przeanalizujmy przykład. Załóżmy, że chcemy napisać prostą funkcję, która mnoży wszystkie swoje argumenty (zakładamy, że argumenty są typu `int`). Przyjmujemy przy tym, że ostatnim argumentem, wskazującym na koniec listy argumentów będzie 0. Będzie ona wyglądała tak:

```
#include <stdarg.h>

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

`va_list` oznacza specjalny typ danych, w którym przechowywane będą argumenty przekazane do funkcji. Gdy chcemy w ciele funkcji skorzystać z przekazanych do niej argumentów wywołujemy makropolecenie `va_start` z parametrem `arg` (wcześniej zadeklarowanym jako `va_list`). Jako drugi parametr musimy podać nazwę ostatniego ustalonego argumentu funkcji (w naszym przypadku `pierwszy`). W celu dostępu do parametrów nieustalonych (oznaczonych przez "...") wywołujemy makro `va_arg` z parametrem `arg` i nazwą typu pobieranego parametru. Kolejne wywołania `va_arg` umożliwiają dostęp do kolejnych argumentów. Po zakończeniu działań na parametrach nieustalonych wywołujemy makro `va_end`. Aby używać ww. makropoleceń należy do programu dołączyć plik nagłówkowy `stdarg.h`.

Uwaga! Należy ściśle przestrzegać zgodności typów parametrów nieustalonych z typami przekazywanymi do makra `va_arg`. Typy `char`, `unsigned char` i `float` są niedozwolone!

Rozważmy dla przykładu funkcję:

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

double wielomian(double x, int st, ...)
{
      double wart=0, wsp;
      va_list ap;
      va_start(ap,st);
      for (;st;--st)
            wart+=va_arg(ap,double)*pow(x,st); //pow to potęgowanie
      wart+=va_arg(ap,double); //wyraz wolny
      va_end(ap);
      return wart;
}

main()
{
      printf("%lf",wielomian(2.0, 3, 1.0, 2.0, -3.0, 5.0));
      /* wart. wielomianu x^3+2x^2-3x+5 w punkcie x=2.0 (15). Jeśli parame-
      try nieustalone zostaną podane w postaci bez kropek wynik będzie
      błędny!  */
      return 0;
}
```

Funkcje przeciążone (C++)

Funkcje przeciążone to funkcje o tej samej nazwie, lecz różnych argumentach (różnej ich liczbie lub typie). Kompilator rozpoznaje, która funkcja ma być wywołana po liczbie i typach przekazanych argumentów. Przykład:

```
#include <stdio.h>

double abs(double x){
      return x<0 ? -x : x;
}

int abs(int x){
      return x<0 ? -x : x;
}

main()
{
      printf("%d, %lf", abs(10), abs(10.1);
      return 0;
}
```

Domyślne wartości argumentów funkcji (C++)

Argumentom przekazywanym do funkcji można przypisywać domyślne wartości, np:

```
double funX(char a, char b = 'c', double x = 24.11);
```

Argumenty `b` i `x` zostały zainicjowane wartościami `'c'` i `24.11`. Wywołanie funkcji z jednym argumentem oznacza, że drugi i trzeci argument przyjmą wartości domyślne. Dopuszczalne wywołania funkcji `funX`:

```
y = funX('a'); // równoważne y = funX('a', 'c', 24.11);
y = funX('a','d'); // równoważne y = funX('a', 'd', 24.11);
y = funX('a', 'd', 649.9);
```

Wszystkie argumenty występujące po argumencie, któremu nadano wartość domyślną muszą posiadać wartości domyślne.

Uwaga na dwuznaczność wywołania funkcji:

```
int fun(int i);
int fun(int j, int k =0);
y = fun(11); //która funkcja się wykona?
```

Funkcje inline (C++)

`inline` napisane na początku deklaracji funkcji jest informacją dla kompilatora, aby w miejscu wywołania tej funkcji wstawił jej kod, zamiast wskaźnika, który wskazuje lokalizację tej funkcji w pamięci. Takie funkcje dalej jednak występują w pamięci komputera, dzięki czemu możemy tworzyć do nich wskaźniki i używać ich jak w przypadku zwykłych funkcji.

Użycie funkcji `inline`:

```
inline int dodaj (int, int);  //deklaracja

int main ()
{
      int a, b, c;
      c = dodaj (a,b);
      return 0;
}

inline int dodaj (int a,int b)  //definicja
{
      return a+b;
}
```

Kompilator zamieni w trakcie kompilacji kod programu:

```
int main ()
{
      int a, b, c;
      {
            c = a+b;  //podstawianie kodu funkcji
      }
      return 0;
}
```

Funkcje `inline` mają zastosowanie, gdy zależy nam na szybkości działania programu. Status `inline` zazwyczaj dodaje się krótkim funkcjom, nie mającym więcej niż kilkanaście linijek kodu. Czasami gdy kompilator uzna, że nasza funkcja jest zbyt długa lub wywołuje się rekurencyjnie ignoruje nasze `inline`. Teoretycznie makroinstrukcje języka C mają dość podobne działanie do `inline`.

Ezoteryka C

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

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

10. PREPROCESOR

W języku C wszystkie linijki zaczynające się od symbolu "#" nie podlegają bezpośrednio procesowi kompilacji. Są to instrukcje preprocesora – elementu kompilatora, który analizuje plik źródłowy w poszukiwaniu wszystkich wyrażeń zaczynających się od "#". Na podstawie tych instrukcji generuje on kod w "czystym" języku C, który następnie jest kompilowany przez kompilator. Ponieważ za pomocą preprocesora można niemal "sterować" kompilatorem, daje on niezwykłe możliwości, które nie były dotąd znane w innych językach programowania.

Preprocesor pozwala na:

- włączanie tekstów innych plików (np. plików nagłówkowych),
- definiowanie makrodefinicji pozwalających zwiększyć czytelność programu,
- sterowanie przebiegiem kompilacji.

Dyrektywy preprocesora wykonują się w momencie kompilacji, przed generacją kodu wynikowego.

10.1. Dyrektywy preprocesora

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

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

Omówimy teraz kilka ważniejszych dyrektyw.

#include

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

```
#include <plik_nagłówkowy_do_dołączenia>
#include "plik_nagłówkowy_do_dołączenia"
```

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

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

lub:

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

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

#define

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

```
#define NAZWA_STALEJ WARTOSC
```

Przykład:

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

`#define ILOCZYN(a,b) ((a)*(b))` – spowoduje, że każde wystąpienie wywołania makrodefinicji `ILOCZYN` zostanie zastąpione przez iloczyn jej argumentów.

Jeśli w miejscu wartości znajduje się wyrażenie, to należy je umieścić w nawiasach, tak jak w drugim przykładzie. Unikniemy w ten sposób niespodzianek związanych z priorytetem operatorów. Przeanalizuj wywołanie makra:

```
y = ILOCZYN(2+5, x-1);
```

Przykłady makrodefinicji prostych:

```
#define PI 3.1416
#define TRUE 1
#define IMIE "Jan"
#define IMIE_I_NAZWISKO IMIE+"Sobieski" //makrodefinicja zagłębiona
#define DRUK_DO_PLIKU fprintf(OUT,"%d",a)
#define ROZMIAR 20
```

Przykłady makrodefinicji parametrycznych:

```
#define ILORAZ(a,b) ((a)/(b))
#define PETLA(n) for(int i=0;i<n;++i)
#define DRUK(a,b,c) fprintf(Out,"%d%d%d",a,b,c)
```

i ich przykładowe wywołania:

```
ILORAZ(6,3); ILORAZ(5+1,4-1); //równoważne ((5+1)/(4-1))
PETLA(10) //równoważne for(int i=0;i<10;++i)
DRUK(1,2,3); //równoważne fprintf(Out,"%d%d%d",a,b,c)
```

Program obrazujący zastosowanie makrodefinicji:

```
#include <stdio.h>
#include <conio.h>
#define PROGRAM main()
#define DRUKUJ puts
#define DRUKUJ2(x) printf(#x" = %d",x) /*# - konwersja do łańcucha,#x zo-
stanie zinterpretowane jako "x" */
#define SUMA(a,b) ((a)+(b))
#define a(x) a##x //## - łączenie jednostek leksykalnych

PROGRAM
{
      int s;
      int a1,a2,a3,a4,a5,a6;

      //Makrodefinicje predefiniowane
      DRUKUJ(__DATE__); //np. Nov 27 2000
      DRUKUJ(__FILE__); //np. MAKRO.CPP
      printf("%d\n",__LINE__); //np. 17
      DRUKUJ(__TIME__); //np. 23:46:24

      s=SUMA(2,3);
      DRUKUJ2(s); //printf("s"" = %d",s)

      a(1)=1; //a1=1

      getch();

      return 0;
}
```

#undef

Ta instrukcja odwołuje definicję wykonaną instrukcją `#define`.

```
#undef STALA
```

Po linii, w której umieścimy tę dyrektywę `STALA` nie będzie zastępowana wartością zgodnie z wcześniejszą makrodefinicją `#define STALA WARTOSC`.

#if #elif #else #endif

Preprocesor zawiera również instrukcje warunkowe, pozwalające na wybór tego co ma zostać skompilowane w zależności od pewnych warunków. Działanie powyższych dyrektyw jest podobne do instrukcji warunkowych w języku C.

`#if`

wprowadza warunek, który jeśli nie jest prawdziwy powoduje pominięcie kompilowania kodu następującego po tej dyrektywie, aż do napotkania jednej z dyrektyw: `#elif`, `#else`, `#endif`.

`#else`

spowoduje skompilowanie kodu jeżeli warunek za `#if` jest nieprawdziwy, aż do napotkania dyrektywy `#elif` lub `#endif`.

`#elif`

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

`#endif`

zamyka blok ostatniej instrukcji warunkowej.

Przykład:

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

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

wiersz nr 4 będzie kompilowany zawsze

#ifdef #ifndef #else #endif

Te instrukcje warunkują kompilację od tego, czy odpowiednia stała (jako makrodefinicja) została zdefiniowana.

#ifdef

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

#ifndef

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

#else,#endif

mają analogiczne zastosowanie jak te z powyższej grupy

Przykład:

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

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

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

#error

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

Przykład:

```
#ifdef BLAD
      #error Poważny błąd kompilacji
#endif
```

#warning

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

Przykład:

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

Spowoduje to takie lub podobne zachowanie kompilatora:

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

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

#line

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

Przykład:

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

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

oraz

Znak # użyty w makrach zamienia stojący za nim identyfikator na napis (łańcuch znaków). Demonstruje to poniższy przykład:

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

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

Program wypisze:

```
i=1
```

```
a=5
```

Czyli `wypisz(a)` jest rozwijane w `printf("%s=%i\n", "a", a)`.

Natomiast znaki "##" łączą (konkatenują) dwie nazwy w jedną. Przykład:

```
#include <stdio.h>
#define abc(x) int x##_zmienna
#define wypisz(x) printf("%s=%i", #x, x)

int main()
{
	...
	abc(nasza) = 2; /* dzięki temu 2 przypiszemy do zmiennej o nazwie nasza_zmienna */
	wypisz(nasza_zmienna);
	return 0;
}
```

Predefiniowane makra

W języku C wprowadzono również serię predefiniowanych makr, które mają ułatwić życie programiście. Oto one:

`__DATE__` – data w momencie kompilacji

`__TIME__` – czas kompilacji

`__FILE__` – łańcuch, który zawiera nazwę pliku, który aktualnie jest kompilowany

`__LINE__` – aktualny numer linii

`__STDC__` – w kompilatorach zgodnych ze standardem ANSI lub nowszym makro to przyjmuje wartość 1

`__STDC_VERSION__` – zależnie od poziomu zgodności kompilatora makro przyjmuje różne wartości:

- jeżeli kompilator jest zgodny z ANSI (rok 1989) makro nie jest zdefiniowane,
- jeżeli kompilator jest zgodny ze standardem z 1994 makro ma wartość 199409L,
- jeżeli kompilator jest zgodny ze standardem z 1999 makro ma wartość 199901L.

Warto również wspomnieć o identyfikatorze `__func__` zdefiniowanym w standardzie C99, którego wartość to nazwa funkcji aktualnie wywołanej.

Spróbujmy użyć tych makr w praktyce:

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

```
#if __STDC_VERSION__ >= 199901L
/*Jezeli mamy do dyspozycji identyfikator __func__ wykorzystajmy go*/
      #define BUG(message) fprintf(stderr, "%s:%d: %s (w funkcji %s)\n", \
                                   __FILE__, __LINE__, message, __func__)
#else
/*Jezeli __func__ nie ma, to go nie używamy*/
      #define BUG(message) fprintf(stderr, "%s:%d: %s\n", \
                                   __FILE__, __LINE__, message)
#endif

int main(void) {
      printf("Program ABC,  data kompilacji: %s %s\n", __DATE__, __TIME__);

      printf("%d",__STDC_VERSION__);
      BUG("Przykladowy komunikat bledu");
      return 0;
}
```

Efekt działania programu, gdy kompilowany jest kompilatorem C99:

```
Program ABC,  data kompilacji: Sep  1 2008 19:12:13
test.c:17: Przykladowy komunikat bledu (w funkcji main)
```

11. BIBLIOTEKA STANDARDOWA

Czym jest biblioteka?

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

Po co nam biblioteka standardowa?

Czysty język C nie może zbyt wiele. Tak naprawdę, to język C sam w sobie praktycznie nie ma mechanizmów do obsługi np. wejścia-wyjścia. Dlatego też większość systemów operacyjnych posiada tzw. bibliotekę standardową zwaną też biblioteką języka C. To właśnie w niej zawarte są podstawowe funkcjonalności, dzięki którym twój program może np. napisać coś na ekranie.

Jak skonstruowana jest biblioteka standardowa?

Biblioteka standardowa nie jest napisana w samym języku C. Ponieważ C jest językiem tłumaczonym do kodu maszynowego, to w praktyce nie ma żadnych przeszkód, żeby np. połączyć go z językiem niskiego poziomu, jakim jest np. asembler. Dlatego biblioteka C z jednej strony udostępnia gotowe funkcje w języku C, a z drugiej za pomocą niskopoziomowych me-

chanizmów komunikuje się z systemem operacyjnym, który wykonuje odpowiednie czynności.

Gdzie są funkcje z biblioteki standardowej?

Pisząc program w języku C używamy różnego rodzaju funkcji, takich jak np. `printf()`. Nie jesteśmy jednak ich autorami, mało tego nie widzimy nawet deklaracji tych funkcji w naszym programie. Nasz pierwszy program "Hello world" zaczynał się od takiej oto linijki:

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

To w pliku nagłówkowym `stdio.h` znajdują się deklaracje funkcji bibliotecznych.

Każdy system operacyjny ma za zadanie wykonywać pewne funkcje na rzecz programów. Wszystkie te funkcje zawarte są właśnie w bibliotece standardowej. W systemach z rodziny UNIX nazywa się ją LibC (biblioteka języka C). To tam właśnie znajduje się funkcja `printf()`, `scanf()`, `puts()` i inne: matematyczne, komunikacji przez sieć itp.

Opis funkcji biblioteki standardowej

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

Indeks alfabetyczny:

http://pl.wikibooks.org/wiki/C/Biblioteka_standardowa/Indeks_alfabetyczny

Indeks tematyczny:

http://pl.wikibooks.org/wiki/C/Biblioteka_standardowa/Indeks_tematyczny

12. OBSŁUGA PLIKÓW – ZAPIS I ODCZYT DANYCH***

Najprościej mówiąc, plik to pewne dane zapisane na dysku. Każdy plik ma określoną nazwę. W naszych programach do każdego pliku, który chcemy przeczytać lub w którym chcemy zapisać dane tworzymy identyfikator. Dzięki temu kod programu jest czytelniejszy i nie trzeba korzystać ciągle z pełnej nazwy pliku. Aby skojarzyć identyfikator z plikiem korzystamy z funkcji `open` lub `fopen`. Różnica wyjaśniona została poniżej.

Podstawowa obsługa plików

Istnieją dwie metody obsługi plików: wysokopoziomowa i niskopoziomowa. Nazwy funkcji reprezentujących pierwszą metodę zaczynają się od litery "f" (np. `fopen()`, `fread()`, `fclose()`). Identyfikatorem pliku w tym przypadku jest wskaźnik na strukturę typu `FILE`. Owa struktura to pewna grupa zmiennych, która przechowuje dane o pliku. Identyfikator pliku stanowi jego "uchwyt".

Funkcje niskopoziomowe to: `read()`, `open()`, `write()` i `close()`. Podstawowym identyfikatorem pliku jest tu liczba całkowita, która jednoznacznie identyfikuje dany plik w systemie operacyjnym. Liczba ta w systemach typu UNIX jest nazywana deskryptorem pliku.

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

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

Dane znakowe

Skupimy się teraz na najprostszym z możliwych zagadnień – zapisie i odczycie znaków oraz napisów (łańcuchów znaków).

Napiszmy program, który stworzy plik "test.txt" i umieści w nim napis "Hello world":

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

int main ()
{
      FILE *fp; /* używamy metody wysokopoziomowej - musimy mieć zatem
                   identyfikator pliku, uwaga na gwiazdkę! */
      char tekst[] = "Hello world";

      if ((fp=fopen("test.txt", "w"))==NULL)
      {
            printf ("Nie mogę otworzyć pliku test.txt do zapisu!\n");
            exit(1);
      }
      fprintf (fp, "%s", tekst); /* zapisz nasz napis w pliku */
      fclose (fp); /* zamknij plik */
      return 0;
}
```

Teraz omówimy najważniejsze elementy programu. Jak już było wspomniane wyżej, do identyfikacji pliku używa się wskaźnika na strukturę `FILE` (czyli `FILE *`). Funkcja `fopen()` zwraca ów wskaźnik w przypadku poprawnego otwarcia pliku, bądź też `NULL`, gdy plik nie może zostać otwarty. Pierwszy argument funkcji to nazwa pliku, natomiast drugi to tryb dostępu, "w" oznacza "write" (zapis). Zwrócony "uchwyt" do pliku (`*fp`) będzie mógł być wykorzystany jedynie w funkcjach zapisujących dane. Gdy otworzymy plik podając tryb "r" ("read", czytanie), będzie można z niego jedynie czytać dane. Funkcja `fopen()` została dokładniej opisana na stronie http://pl.wikibooks.org/wiki/C/fopen.

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

Pliki a strumienie

Można zauważyć, że do zapisu do pliku używamy funkcji `fprintf()`, która wygląda bardzo podobnie do `printf()` (obie funkcje tak naprawdę robią tak samo). Jedyną różnicą jest to, że w `fprintf()` musimy jako pierwszy argument podać identyfikator pliku.

Używana do wczytywania danych z klawiatury funkcja `scanf()` też ma swój odpowiednik wśród funkcji operujących na plikach – funkcję `fscanf()`.

W rzeczywistości język C traktuje tak samo klawiaturę i plik – są to źródła danych, podobnie jak ekran i plik, do których można dane kierować. Między klawiaturą i plikiem na dysku są podstawowe różnice i dostęp do nich odbywa się inaczej, jednak funkcje języka C pozwalają nam o tym zapomnieć i same zajmują się szczegółami technicznymi. Z punktu widzenia programisty, urządzenia te sprowadzają się do nadanego im w programie identyfikatora. Uogólnione pliki nazywa się w C **strumieniami.**

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

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

Aby z nich korzystać należy dołączyć plik nagłówkowy `stdio.h`.

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

Warto tutaj zauważyć, że konstrukcja:

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

jest równoważna konstrukcji:

```
printf ("Hej, ja działam!");
```

Podobnie jest z funkcją scanf().

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

działa tak samo jak:

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

Obsługa błędów

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

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

dostaniemy komunikat:

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

Zaawansowane operacje

Oto krótki program, który swoje wejście zapisuje do pliku o nazwie podanej w linii poleceń:

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

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

Tym razem skorzystaliśmy już z dużo większego repertuaru funkcji. Między innymi można zauważyć tutaj funkcję `fputc()`, która umieszcza pojedynczy znak w pliku. Ponadto w wyżej zaprezentowanym programie została użyta stała `EOF`, która reprezentuje koniec pliku (ang. *End Of File*). Powyższy program otwiera plik, którego nazwa przekazywana jest jako

pierwszy argument programu, a następnie kopiuje dane z wejścia programu (`stdin`) na wyjście (`stdout`) oraz do utworzonego pliku (identyfikowanego za pomocą `fp`). Program robi to tak długo, aż naciśniemy kombinację klawiszy Ctrl+D (w systemach unixowych) lub Ctrl+Z (w Windows), która wyśle do programu informację, że skończyliśmy wpisywać dane. Program wyjdzie wtedy z pętli i zamknie utworzony plik.

*Rozmiar pliku****

Dzięki standardowym funkcjom języka C możemy m.in. określić rozmiar pliku. Przy każdym odczycie/zapisie danych z/do pliku wskaźnik przesuwa się o liczbę przeczytanych/zapisanych bajtów. Możemy jednak ustawić wskaźnik w dowolnie wybranym miejscu. Aby odczytać rozmiar pliku powinniśmy ustawić nasz wskaźnik na koniec pliku, po czym odczytać ile bajtów wskazuje. Użyjemy do tego dwóch funkcji: `fseek()` oraz `fgetpos()`. Pierwsza służy do ustawiania wskaźnika na odpowiedniej pozycji w pliku, a druga do odczytywania, na którym bajcie pliku znajduje się wskaźnik. Kod, który określa rozmiar pliku znajduje się tutaj:

```
#include <stdio.h>

int main (int argc, char **argv)
{
      FILE *fp = NULL;
      fpos_t dlugosc; /* fpos_t to specjalny typ danych, używany do okre-
      ślania położenia danych w strumieniu */
      if (argc < 2) {
            printf ("Nie podano nazwy pliku\n");
            return 1;
      }
      if ((fp=fopen(argv[1], "rb"))==NULL) {
            printf ("Błąd otwarcia pliku: %s!\n", argv[1]);
            return 1;
      }
      fseek (fp, 0, SEEK_END); /* ustawiamy wskaźnik na koniec pliku */
      fgetpos (fp, &dlugosc);
      printf ("Rozmiar pliku: %d\n", dlugosc);
      fclose (fp);
      return 0;
}
```

Przykład – pliki graficzny

Najprostszym przykładem rastrowego pliku graficznego jest plik PPM patrz http://pl.wikipedia.org/wiki/Portable_anymap). Poniższy program pokazuje jak utworzyć plik w katalogu roboczym programu. Do zapisu nagłówka pliku używana jest funkcja `fprintf`, która zapisuje do plików binarnych lub tekstowych. Do zapisu tablicy z kolorami kolejnych pikseli używana jest funkcja `fwrite`, która zapisuje do plików binarnych.

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

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

13. TABLICE

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

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

Sposoby deklaracji tablic

Tablicę deklaruje się w następujący sposób:

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

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

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

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

```
int tablica[3] = {0,1,2};
```

Niekoniecznie trzeba podawać rozmiar tablicy przy deklaracji, np.:

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

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

Rozpatrzmy następujący kod:

```
#include <stdio.h>
#define ROZMIAR 3
int main()
{
      int tab[ROZMIAR] = {3,6,8};
      int i;
      printf ("Druk tablicy tab:\n");
      for (i=0; i<ROZMIAR; ++i) {
            printf ("Element numer %d = %d\n", i, tab[i]);
      }
      return 0;
}
```

Wynik:

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

W powyższym przykładzie użyliśmy stałej do podania rozmiaru tablicy. Jest to o tyle pożądany zwyczaj, że w razie potrzeby zmiany rozmiaru tablicy, zmieniana jest tylko wartość w jednej linijce kodu przy `#define`. w innym przypadku musielibyśmy szukać wszystkich wystąpień rozmiaru rozsianych po kodzie całego programu.

Odczyt/zapis wartości do tablicy

Tablicami posługujemy się tak samo jak zwykłymi zmiennymi. Różnica polega jedynie na podawaniu indeksu tablicy, aby dostać się do jej konkretnego elementu. Numeracja indeksów rozpoczyna się od 0!

Przeanalizuj następujący kod:

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

Jak widać, na początku deklarujemy 5-elementową tablicę, którą od razu zerujemy. Następnie pod trzeci i czwarty element (liczone począwszy od 0) podstawiamy liczby 3 i 7. Pętla ma za zadanie wyprowadzić wynik naszych działań.

Tablice znaków

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

- bufory – gdy wykorzystujemy je do przechowywania ogólnie pojętych danych, gdy traktujemy je jako po prostu "ciągi bajtów" (typ `char` ma rozmiar 1 bajta).
- napisy – gdy zawarte w nich dane traktujemy jako ciągi liter; jest im poświęcony osobny rozdział "Napisy".

Oto przykłady deklaracji tablic znaków połączone z ich inicjalizacją:

```
char tz1[20]="To jest tablica 1 \n";
char tz2[20]={'T','o',' ','t','a','b','l','i','c','a',' '
',' 2','\n','\0'}; //wymagany znak '\0'
```

Tablice wielowymiarowe

Rozważmy teraz konieczność przechowania w pamięci komputera macierzy o wymiarach 10x10. Można by tego dokonać tworząc 10 osobnych tablic jednowymiarowych, reprezentujących poszczególne wiersze macierzy. Jednak język C dostarcza nam dużo wygodniejszej metody, która w dodatku jest bardzo łatwa w użyciu. Są to tablice wielowymiarowe lub inaczej "tablice tablic". Tablice wielowymiarowe definiujemy podając przy zmiennej kilka wymiarów, np.:

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

Tak samo wygląda dostęp do poszczególnych elementów tablicy:

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

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

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

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

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

Oczywiście można tworzyć tablice o większej liczbie wymiarów, np.:

```
int i[20][10][2];
```

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

Ograniczenia tablic

Pomimo swej wygody tablice statyczne mają ograniczony, z góry zdefiniowany rozmiar, którego nie można zmienić w trakcie działania programu. Dlatego też w niektórych zastosowaniach tablice statyczne zostały wyparte tablicami dynamicznymi, których rozmiar może być określony w trakcie działania programu. Zagadnienie to zostało opisane w następnym rozdziale.

> Uwaga! Przy używaniu tablic trzeba być szczególnie ostrożnym, gdyż kompilator nie kontroluje przekroczenia przez indeks rozmiaru tablicy. Efektem będzie odczyt lub zapis pamięci, znajdującej się poza tablicą.

Wystarczy pomylić się o jedno miejsce by spowodować, że działanie programu zostanie nagle przerwane przez system operacyjny:

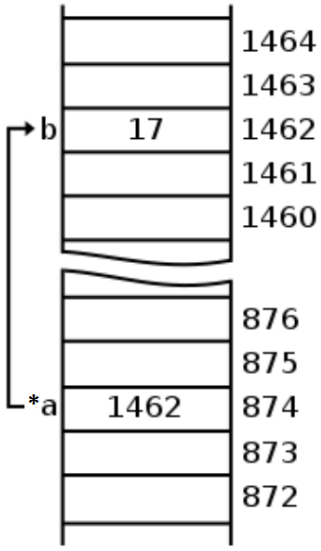
```
int tab[100];
int i;

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

14. WSKAŹNIKI

Co to jest wskaźnik?

Wskaźnik (ang. pointer) to specjalny rodzaj zmiennej, w której zapisany jest **adres** w pamięci komputera. Oznacza to, że wskaźnik wskazuje miejsce, gdzie zapisana jest jakaś informacja (np. zmienna). Możemy powiedzieć, że adres to pewna liczba całkowita, jednoznacznie definiująca położenie pewnego obiektu w pamięci komputera. Tymi obiektami mogą być np. zmienne, elementy tablic czy nawet funkcje.

Obrazowo możemy wyobrazić sobie pamięć komputera jako bibliotekę, a zmienne jako książki. Zamiast brać książkę z półki samemu (analogicznie do korzystania wprost ze zwykłych zmiennych), możemy podać bibliotekarzowi wypisany rewers z numerem katalogowym książki, a on znajdzie ją za nas. Analogia ta nie jest doskonała, ale pozwala wyobrazić sobie niektóre cechy wskaźników: numer na rewersie identyfikuje pewną książkę, kilka rewersów może dotyczyć tej samej książki, numer w rewersie możemy skreślić i użyć go do zamówienia innej książki, a jeśli wpiszemy nieprawidłowy numer, to możemy dostać nie tę książkę, którą chcemy, albo też nie dostać nic.



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

Funkcje wskaźników:

- tworzenie dynamicznych struktur danych,
- zarządzanie blokami pamięci,
- przekazywanie argumentów do funkcji.

Operowanie na wskaźnikach

By stworzyć wskaźnik do zmiennej i móc się nim posługiwać, należy przypisać mu odpowiednią wartość – adres obiektu, na jaki ma wskazywać. Skąd mamy znać ten adres? W języku C możemy "zapytać się" o adres za pomocą operatora `&` (**operatora pobrania adresu**). Przeanalizuj następujący kod:

```
#include <stdio.h>
int main (void)
{
      int x = 80;
      printf("Wartość zmiennej x: %d\n", x );
      printf("Adres zmiennej x: %p\n", &x ); //%p - od pointer
      return 0;
}
```

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

```
Wartość zmiennej liczba: 80
Adres zmiennej liczba: 0022FF74
```

Aby móc przechowywać taki adres, zadeklarujemy zmienną wskaźnikową. Ważną informacją, oprócz samego adresu wskazywanej zmiennej, jest typ wskazywanej zmiennej. Mimo że wskaźnik jest zawsze typu adresowego, kompilator wymaga od nas, abyśmy przy deklaracji podali typ zmiennej, na którą wskaźnik będzie wskazywał. Wskaźniki deklaruje się poprzedzając ich indntyfikator znakiem * (gwiazdki), np.:

```
int *wskaznik1; // zmienna wskaźnikowa na obiekt typu liczba całkowita
char *wskaznik2; // zmienna wskaźnikowa na obiekt typu znak
float *wskaznik3; // zmienna wskaźnikowa na obiekt typu liczba zmiennoprzecinkowa
```

Uwaga! Gwiazdkę łączymy ze zmienną, nie z typem.

Aby uniknąć pomyłek, lepiej jest pisać gwiazdkę tuż przy zmiennej, albo jeszcze lepiej – nie mieszać deklaracji wskaźników i zmiennych:

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

Dostęp do wskazywanego obiektu

Aby dobrać się do wartości wskazywanej przez wskaźnik, należy użyć unarnego operatora *, zwanego **operatorem wyłuskania**. Mimo, że kolejny raz używamy gwiazdki, oznacza ona teraz coś zupełnie innego. Jest tak, ponieważ używamy jej w zupełnie innym miejscu: nie przy deklaracji zmiennej (gdzie gwiazdka oznacza deklarowanie wskaźnika), a przy wykorzystaniu zmiennej, gdzie odgrywa rolę operatora, podobnie jak operator & (pobrania adresu obiektu). Program ilustrujący:

```
#include <stdio.h>
int main (void)
{
      int liczba = 80;
      int *wskaznik = &liczba; // wskaznik przechowuje adres, ktory pobieramy od zmiennej liczba
      printf("Wartosc zmiennej: %d; jej adres: %p.\n", liczba,
      &liczba);

      printf("Adres zapisany we wskazniku: %p, wskazywana wartosc: %d.\n",
      wskaznik, *wskaznik);

      *wskaznik = 42; // zapisanie liczby 42 do obiektu, na który wskazuje wskaznik

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

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

Przykładowy wynik programu:

```
Wartosc zmiennej: 80, jej adres: 0022FF74.
Adres zapisany we wskazniku: 0022FF74, wskazywana wartosc: 80.
Wartosc zmiennej: 42, wartosc wskazywana przez wskaznik: 42
Wartosc zmiennej: 66, wartosc wskazywana przez wskaznik: 66
```

Przykłady użycia wskaźników:

```
double a, b, *wsk = &a, *c; //deklaracje
*wsk = 12.7; // a = 12.7
wsk = &b; // skojarzenie wskaźnika z b
c = wsk; // kopiowanie adresu spod wsk do c; c wskazuje na b
```

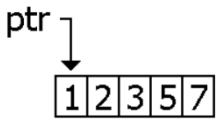
> Uwaga! Zapis `wsk` oznacza adres, a `*wsk` zawartość pamięci.

```
main()
{
      int z1, *wsk1;
      char znak1='a', znak2, *wsk_znak=&znak1;

      wsk1=&z1;   //skojarzenie wskaźnika *wsk1 ze zmienną z1
      *wsk1=50;   //przypisanie równoważne z1=50
      znak2=*wsk_znak;  //przypisanie równoważne znak2=znak1

      z1=*wsk znak;  //przypisanie dopuszczalne, z1 przyjmie wart.
      kodu ASCII znaku wskazywanego przez *wsk_znak; operacja
      odwrotna jest niedopuszczalna

      return 0;
}
```

Arytmetyka wskaźników

W języku C do wskaźników można dodawać lub odejmować liczby całkowite. Istotne jest jednak, że dodanie do wskaźnika np. liczby 2 nie spowoduje przesunięcia się w pamięci komputera o dwa bajty. Tak naprawdę przesuniemy się o 2*(rozmiar typu wskaźnika). Przeanalizujmy przykład:

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

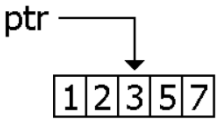


Rysunek: Wskaźnik wskazuje na pierwszą komórkę pamięci.

Gdy wykonamy

```
ptr += 2;
```

otrzymujemy



Rysunek: Przesunięcie wskaźnika na kolejne komórki.

Wskaźnik ustawi się na trzecim elemencie tablicy.

Wskaźniki można od siebie odejmować, czego wynikiem jest odległość pomiędzy dwoma wskazywanymi elementami tablicy. Odległość ta wyrażona jest liczbą elementów, które znajdują się pomiędzy wskazywanymi elementamim, a nie liczbą bajtów. Np.:

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

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

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

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

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

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

Tablice a wskaźniki

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

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

występuje w pamięci w sześciu komórkach (przy założeniu, że typ `int` zajmuje 2 bajty, a jedna komórka ma rozmiar jednego bajta):

element 1	element 2	element 3

00000000	01100100	00000000	11001000	00000001	00101100

Do trzeciego elementu można się dostać tak (komórki w tablicy numeruje się od zera):

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

albo wykorzystując metodę wskaźnikową:

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

Z definicji obie te metody są równoważne.

Z definicji (z wyjątkiem użycia operatora `sizeof`) wartością zmiennej lub wyrażenia typu tablicowego jest wskaźnik na jej pierwszy element (`tab == &tab[0]`). Co więcej, można pójść w drugą stronę i potraktować wskaźnik jak tablicę:

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

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

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

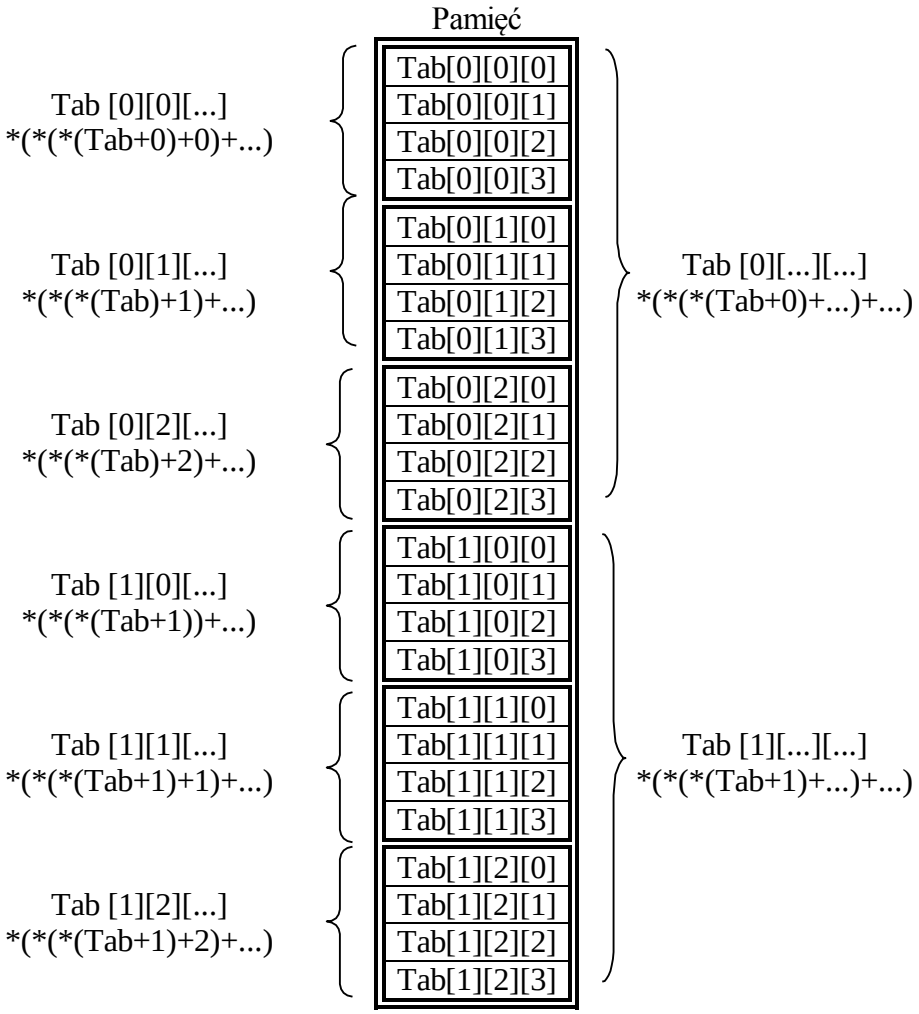
Skąd ta dziwna notacja? Uzasadnienie jest proste:

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

I jeszcze kilka przykładów użycia tablic i wskaźników:

```
int Tab[2][3][4]; //deklaracja tablicy trójwymiarowej
double *tab_wsk[5]; //tablica wskaźników przechowująca adresy
char c[4]={'a','b','c','d'}; //inicjacja tablicy znaków

Tab[0][0][0]=1; //wstawienie 1 w pierwszy element Tab

*(c+3)='e'; //to samo co c[3]='e';

*(*(*(Tab+0)+1)+2)=-1; //to samo co Tab[0][1][2]=-1

double k=3.14;
tab_wsk[2]=&k; //zapamiętanie adresu zmiennej k w 3-cim elemencie ta-
blicy wskaźników
```

Rozmieszczenie elementów `Tab[2][3][4]` w pamięci pokazano poniżej.



Wskaźnik jako argument funkcji

Gdy argumenty do funkcji przekazujemy jako zmienne, funkcja otrzymuje jedynie lokalne kopie argumentów. Wszelkie zmiany na tych kopiach wewnątrz funkcji dokonują się lokalnie i nie są widziane poza funkcją. Przekazując do funkcji wskaźnik, również zostaje stworzona kopia, ale wskaźnika, czyli adresu pamięci, gdzie przechowywane są dane. Każda operacja na wartości wskazywanej powoduje zmianę wartości zmiennej zewnętrznej. Rozpatrzmy poniższy przykład:

```
#include <stdio.h>

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

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

```
}
```

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

Uwaga! Zwróćmy uwagę na wywołanie `func(&z)`. Należy pamiętać, by do funkcji oczekującej wskaźnika przekazać **adres zmiennej**, a nie samą zmienną. Jeśli byśmy napisali `func(z)` wówczas funkcja użyłaby liczby 3 jako adres i starałaby się zmienić komórkę pamięci o adresie 3. Kompilator powinien ostrzec w takim przypadku o konwersji z typu `int` do wskaźnika, ale jeśli zignorujemy ostrzeżenie, nasz program prawdopodobnie zamknie się z komunikatem o błędzie.

Pułapki wskaźników

Ważne jest, aby przy posługiwaniu się wskaźnikami nigdy nie próbować odwoływać się do komórki wskazywanej przez wskaźnik o wartości NULL (tzw. zero adresowe), ani nie odwoływać się do niezainicjowanego wskaźnika! Oto przykłady:

```
int *wsk;
printf ("zawartosc komorki: %d\n", *wsk);    /* Błąd */
wsk = NULL;
printf ("zawartosc komorki: %d\n", *wsk);    /* Błąd */
```

Gdy używamy operatora `sizeof`, podając jako argument zmienną wskaźnikową, to uzyskana liczba będzie oznaczała rozmiar adresu, a nie rozmiar typu użytego podczas deklarowania naszego wskaźnika. Wielkość ta będzie zawsze miała taki sam rozmiar dla każdego wskaźnika, w zależności od kompilatora, a także docelowej platformy. Gdy chcemy dowiedzieć się jaki jest rozmiar danych wskazywanych przez wskaźnik używajmy `sizeof(*wskaźnik)`. Przykład:

```
char *zmienna;
int z = sizeof zmienna; /* z może być równe 4 (rozmiar adresu na ma-
szynie 32 bit) */
z = sizeof(char*);      /* to samo, co wyżej */
z = sizeof *zmienna;    /* tym razem z = rozmiar znaku, tj. 1 */
z = sizeof(char);       /* to samo, co wyżej */
```

Na co wskazuje NULL?

Analizując kody źródłowe programów często można spotkać taki oto zapis:

```
void *wskaznik = NULL; /* lub = 0 */
```

Wiemy już, że nie możemy odwołać się pod komórkę pamięci wskazywaną przez wskaźnik NULL. Po co zatem przypisywać wskaźnikowi 0? Odpowiedź może być zaskakująca: właśnie po to, aby uniknąć błędów! Większość (jeśli nie wszystkie) funkcji, które zwracają wskaźnik, w przypadku błędu zwróci właśnie NULL, czyli zero. Tutaj rodzi się kolejna wskazówka:

jeśli w danej zmiennej przechowujemy wskaźnik, zwrócony wcześniej przez jakąś funkcję zawsze sprawdzajmy, czy nie jest on równy 0/NULL. Wtedy mamy pewność, że funkcja zadziałała poprawnie.

Dokładniej, NULL nie jest słowem kluczowym, lecz stałą (makrem) zadeklarowaną przez dyrektywy preprocesora. Deklaracja taka może być albo wartością 0, wartością 0 zrzutowaną na `void*`: `(((void *)0))`, albo też jakimś słowem kluczowym deklarowanym przez kompilator.

Warto zauważyć, że przypisywanie wskaźnikowi NULL, nie oznacza, że jest on reprezentowany przez same zerowe bity. Co więcej, wskaźniki NULL różnych typów mogą mieć różną wartość!

Stałe wskaźniki

Podobnie jak możemy deklarować zwykłe stałe, tak samo możemy mieć stałe wskaźniki. Są ich dwa rodzaje. Wskaźniki na stałą wartość:

```
const int *a;
int const * a;  /* równoważnie */
```

oraz stałe wskaźniki:

```
int * const b;
```

Pierwszy to wskaźnik, którym **nie można zmienić wskazywanej wartości**, np.:

```
int x = 9;
int const *a = &x;
*a = 5; // niedopuszczalne
```

Drugi to wskaźnik, którego **nie można przestawić na inny adres**, np.:

```
int x = 9, y = 4;
int *const a = &x;
*a = 5; // teraz dopuszczalne
a = &y; // ale to niedopuszczalne
```

Obie opcje można połączyć, deklarując stały wskaźnik, którym nie można zmienić wartości wskazywanej zmiennej. Można to zrobić na dwa sposoby:

```
const int * const c;
int const * const c;  /* równoważnie  */

int i=0;
const int *a=&i;
int * const b=&i;
int const * const c=&i;
*a = 1;  /* kompilator zaprotestuje */
*b = 2;  /* ok */
*c = 3;   /* kompilator zaprotestuje */
a = b;   /* ok */
b = a;   /* kompilator zaprotestuje */
c = a;   /* kompilator zaprotestuje */
```

Wskaźniki na stałą wartość są przydatne między innymi w sytuacji przekazywania danych do funkcji, gdy danych tych nie powinniśmy wewnątrz funkcji zmieniać. Przekazanie tych danych przez zmienną też chroni je przed zmianami, ale wymaga ich skopiowania na potrzeby funkcji. Może to być problemem, gdy dane te zajmują dużo miejsca. Przykład:

```
void funkcja(const duza_struktura *ds)
{
    /* czytamy z ds, ale nie możemy jej modyfikować */
}
 ....
funkcja(&dane); /* mamy pewność, że zmienna dane nie zostanie zmie-
niona */
```

Dynamiczna alokacja pamięci

Normalnie zmienne programu przechowywane są na tzw. stosie (ang. *stack*) – powstają, gdy program wchodzi do bloku, w którym zmienne są zadeklarowane, a znikają w momencie, kiedy program opuszcza ten blok. Jeśli deklarujemy tak tablice, to ich rozmiar musi być znany w momencie kompilacji, żeby kompilator zarezerwował na stosie odpowiednią ilość pamięci.

Dostępny jest inny rodzaj rezerwacji (czyli alokacji) pamięci. Jest to alokacja na stercie (ang. *heap*). Sterta to obszar pamięci wspólny dla całego programu. Przechowywane są w nim zmienne, których czas życia nie jest związany z poszczególnymi blokami. Musimy sami rezerwować dla nich miejsce i to miejsce zwalniać, ale dzięki temu możemy to zrobić w dowolnym momencie działania programu.

Należy pamiętać, że rezerwowanie i zwalnianie pamięci na stercie zajmuje więcej czasu niż analogiczne działania na stosie. Dodatkowo, zmienna wymaga więcej miejsca na stercie niż na stosie. Tak więc używajmy dynamicznej alokacji tylko tam, gdzie jest niezbędba – dla danych, których rozmiaru nie jesteśmy w stanie przewidzieć na etapie kompilacji lub ich żywotność ma być niezwiązana z blokiem, w którym zostały zaalokowane.

Obsługa pamięci

Podstawową funkcją do rezerwacji pamięci jest funkcja `malloc()`. Podając jej rozmiar (w bajtach) potrzebnej pamięci, dostajemy wskaźnik do zaalokowanego obszaru.

Załóżmy, że chcemy stworzyć tablicę liczb typu `float`:

```
int rozmiar;
float *tablica;

rozmiar = 3;
tablica = (float*) malloc(rozmiar * sizeof *tablica);
tablica[0] = 0.1;
```

Najpierw deklarujemy zmienne – rozmiar tablicy i wskaźnik, który będzie wskazywał obszar w pamięci, gdzie będzie trzymana tablica. `rozmiar * sizeof *tablica` oblicza potrzebny rozmiar pamięci dla tablicy. Rzutowanie `(float*)` pozwala interpretować zarezerwowana pamięć jako blok zmiennych typu `float` (standardowo funkcja `malloc()` zwraca `void*`).

Jeśli dany obszar pamięci nie będzie już nam więcej potrzebny powinniśmy go zwolnić, aby system operacyjny mógł go przydzielić innym potrzebującym procesom. Do zwolnienia obszaru pamięci używamy funkcji `free()`, która przyjmuje tylko jeden argument – wskaźnik, który otrzymaliśmy w wyniku działania funkcji `malloc()`:

```
free (tablica);
```

> Uwaga! Należy pamiętać o zwalnianiu pamięci, inaczej dojdzie do tzw. wycieku pamięci – program będzie rezerwował nową pamięć, ale nie zwracał jej z powrotem i w końcu pamięci może mu zabraknąć.

Należy też uważać, by nie zwalniać dwa razy tego samego miejsca. Po wywołaniu `free()` wskaźnik nie zmienia wartości, pamięć wskazywana przez niego może też nie od razu ulec zmianie. Zaraz po wywołaniu funkcji `free()` wskazane jest przypisanie wskaźnikowi wartości 0.

Czasami możemy potrzebować zmienić rozmiar już przydzielonego bloku pamięci. Tu z pomocą przychodzi funkcja `realloc()`:

```
tablica = realloc(tablica, 2*rozmiar*sizeof *tablica);
```

Funkcja ta zwraca wskaźnik do bloku pamięci o pożądanej wielkości (lub NULL gdy zabrakło pamięci). Jeśli zażądamy zwiększenia rozmiaru, a za zaalokowanym aktualnie obszarem nie będzie wystarczająco dużo wolnego miejsca, funkcja znajdzie nowe miejsce i przekopiuje tam starą zawartość.

Ostatnią funkcją jest funkcja `calloc()`. Przyjmuje ona dwa argumenty: liczbę elementów tablicy oraz wielkość pojedynczego elementu. Podstawową różnicą pomiędzy funkcjami `malloc()` i `calloc()` jest to, że ta druga zeruje zawartość przydzielonej pamięci (do wszystkich bajtów wpisuje wartość 0).

Przykład tworzenia tablicy typu `float` przy użyciu `calloc()`:

```
int rozmiar;
float *tablica;

rozmiar = 3;
tablica = (float*) calloc(rozmiar, sizeof *tablica);
tablica[0] = 0.1;
```

*Wskaźniki na funkcje****

Dotychczas zajmowaliśmy się sytuacją, gdy wskaźnik wskazywał na jakąś zmienną. Jednak nie tylko zmienna ma swój adres w pamięci. Dotyczy to także funkcji. Ponieważ funkcja ma swój adres, to nie ma przeszkód, aby i na nią wskazywał jakiś wskaźnik.

Deklaracja wskaźnika na funkcję

Wskaźnik na funkcję ma specyficzną deklarację:

```
typ_zwracanej_wartości (*nazwa_wskaźnika)(typ1 parametr1, typ2 parametr2,
...);
```

Oczywiście parametrów może być więcej (albo też w ogóle może ich nie być). Oto przykład wykorzystania wskaźnika na funkcję:

```
#include <stdio.h>

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

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

Zwróćmy uwagę na dwie rzeczy:

- przypisując nazwę funkcji bez nawiasów do wskaźnika automatycznie informujemy kompilator, że chodzi nam o adres funkcji,
- wskaźnika używamy tak, jak normalnej funkcji, na którą on wskazuje.

Możliwe deklaracje wskaźników

Poniżej pokazano przykłady deklaracji różnych wskaźników.

`int i;`	zmienna całkowita (typu int) 'i'
`int *p;`	wskaźnik 'p' wskazujący na zmienną całkowitą
`int a[];`	tablica 'a' liczb całkowitych typu int
`int f();`	funkcja 'f' zwracająca liczbę całkowitą typu int
`int **pp;`	wskaźnik 'pp' wskazujący na wskaźnik wskazujący na liczbę całkowitą typu int
`int (*pa)[];`	wskaźnik 'pa' wskazujący na tablicę liczb całkowitych typu int
`int (*pf)();`	wskaźnik 'pf' wskazujący na funkcję zwracającą liczbę całkowitą typu int
`int *ap[];`	tablica 'ap' wskaźników na liczby całkowite typu int
`int *fp();`	funkcja 'fp', która zwraca wskaźnik na zmienną typu int
`int ***ppp;`	wskaźnik 'ppp' wskazujący na wskaźnik wskazujący na wskaźnik wskazujący na liczbę typu int
`int (**ppa)[];`	wskaźnik 'ppa' na wskaźnik wskazujący na tablicę liczb całkowitych typu int

`int (**ppf)();`	wskaźnik 'ppf' wskazujący na wskaźnik funkcji zwracającej dane typu int
`int *(*pap)[];`	wskaźnik 'pap' wskazujący na tablicę wskaźników na typ int
`int *(*pfp)();`	wskaźnik 'pfp' na funkcję zwracającą wskaźnik na typ int
`int **app[];`	tablica wskaźników 'app' wskazujących na wskaźniki wskazujące na typ int
`int (*apa[])[];`	tablica wskaźników 'apa' wskazujących na tablice liczb całkowitych typu int
`int (*apf[])();`	tablica wskaźników 'apf' na funkcje, które zwracają typ int
`int **fpp();`	funkcja 'fpp', która zwraca wskaźnik na wskaźnik, który wskazuje typ int
`int (*fpa())[];`	funkcja 'fpa', która zwraca wskaźnik na tablicę liczb typu int
`int (*fpf())();`	funkcja 'fpf', która zwraca wskaźnik na funkcję, która zwraca dane typu int

Typowe błędy

Jednym z najczęstszych błędów, oprócz prób wykonania operacji na wskaźniku NULL, są odwołania się do obszaru pamięci po jego zwolnieniu. Po wykonaniu funkcji `free()` nie możemy już wykonywać żadnych odwołań do zwolnionego obszaru.

Inne błędy:

- odwołania do adresów pamięci, które są poza obszarem przydzielonym funkcją `malloc()`
- brak kontroli, czy wskaźnik ma wartość NULL
- wycieki pamięci, czyli niezwalnianie przydzielonej wcześniej pamięci.

Ciekawostki

W rozdziale "Zmienne" pisaliśmy o stałych. Normalnie nie mamy możliwości zmiany ich wartości, ale z użyciem wskaźników staje się to możliwe:

```
const int a=0;
int *c=&a;
*c = 1;
printf("%i\n",a); /* wypisuje 1 */
```

Konstrukcja taka może jednak wywołać ostrzeżenie kompilatora bądź nawet jego błąd.

Język C++ oferuje mechanizm podobny do wskaźników – **referencje**. Referencje służą do reprezentacji innych zmiennych w programie. Wykorzystuje się je do przekazywania argumentów do funkcji. Deklaracja połączona jest zawsze z inicjalizacją:

```
int a, &b = a; // b to referencja
```

Użycie:

```
b = 7; // skutek taki sam jak a = 7
```

Język C++ dostarcza też innego sposobu dynamicznej alokacji i zwalniania pamięci – przez operatory `new` i `delete`:

```
char *wsk3 = new char; //rezerwacja pamięci dla znaku; dla tablic:
new char[n]
...
delete wsk3; // zwolnienie pamięci
```

15. NAPISY

Do przechowywania ciągów znaków w C służą tzw. "łańcuchy" (ang. string). Napis jest zapisywany w kodzie programu jako ciąg znaków zawarty pomiędzy dwoma cudzysłowami, np.:

```
printf ("Napis w języku C");
```

W pamięci taki łańcuch jest ciągiem następujących po sobie znaków (char), który kończy się znakiem "null", zapisywanym jako '**\0**' (Uwaga: Nie należy mylić znaku null ze wskaźnikiem null, czy też NULL). Do poszczególnych znaków łańcucha odwołujemy się jak w tablicy:

```
const char *tekst = "Jakiś tam tekst";
printf("%c\n", "przykład"[0]); /* wypisze 'p' - znaki w napisach są
numerowane od zera */
printf("%c\n", tekst[2]);      /* wypisze k */
```

Ponieważ napis w pamięci kończy się zerem umieszczonym tuż za ostatnim znakiem napisu, odwołanie się do znaku o indeksie równym długości napisu zwróci zero:

```
printf("%d", "test"[4]);       /* wypisze 0 */
```

Napisy możemy wczytywać z klawiatury i wypisywać na ekran przy pomocy dobrze znanych funkcji `scanf()`, `printf()` i pokrewnych. Formatem używanym dla napisów jest `%s`.

```
printf("%s", tekst);
```

Większość funkcji działających na napisach znajduje się w pliku nagłówkowym `string.h`.

Jeśli łańcuch jest zbyt długi, można zapisać go w kilku linijkach kodu, ale wtedy przechodząc do następnej linii musimy na końcu postawić znak "`\`".

```
printf("Ten napis zajmuje \
więcej niż jedną linię");
```

Aby wydrukować napis w kilku liniach należy wstawić do niego `\n`:

```
printf("Ten napis\nna ekranie\nzajmie więcej niż jedną linię.");
```

15.1. Jak łańcuchy są przechowywane w pamięci?

1	2	3	4	5	6	7	8
'J'	'e'	'z'	'y'	'k'	' '	'C'	'\0'

Rysunek: Napis "Jezyk C" przechowywany w pamięci.

Zmienna, która przechowuje łańcuch znaków jest tak naprawdę wskaźnikiem do ciągu znaków (bajtów) w pamięci. Możemy też myśleć o napisie jako o tablicy znaków (jak wyjaśnialiśmy wcześniej, tablice to też wskaźniki).

Sposoby deklaracji napisu:

```
const char *tekst  = "Jezyk C"; /* Umieszcza napis w obszarze danych
programu i przypisuje adres */
char tekst[] = "Jezyk C"; /* Umieszcza napis w tablicy */
char tekst[] = {'J','e','z','y','k',' ','C','\0'};
```

Kompilator automatycznie przydziela wtedy odpowiednią ilość pamięci (tyle bajtów, ile jest liter plus jeden dla kończącego nulla). Jeśli natomiast wiemy, że dany łańcuch powinien przechowywać określoną ilość znaków (nawet, jeśli w deklaracji tego łańcucha podajemy mniej znaków) deklarujemy go w taki sam sposób, jak tablicę jednowymiarową:

```
char tekst[80] = "Ten tekst musi być krótszy niż 80 znaków";
```

Należy pamiętać, że napis jest tak naprawdę tablicą. Jeśli zarezerwowaliśmy dla napisu 80 znaków, to przypisanie do niego dłuższego napisu jest niedopuszczalne, możemy zmienić w ten sposób przypadkowo wartość innych zmiennych. Program może wtedy zachowywać się nieprzewidywalnie.

Deklaracja `const char *tekst = "cokolwiek";` oraz `char tekst[] = "cokolwiek";` pomimo, że wyglądają bardzo podobnie bardzo się od siebie różnią. W przypadku pierwszej deklaracji próba zmodyfikowania napisu (np. `tekst[0] = 'C';`) jest niedopuszczalna.

Kończący napis znak null ('\0') jest kluczowy. W zasadzie wszystkie funkcje operujące na napisach opierają się właśnie na nim. Na przykład, `strlen()` szuka rozmiaru napisu idąc od początku napisu i zliczając znaki, aż nie natrafi na znak o kodzie zero. Jeśli nasz napis nie kończy się znakiem null, funkcja będzie szła dalej po pamięci. Wszystkie operacje podstawienia typu `tekst = "Tekst"` powodują automatyczne zakończenie napisu nullem (o ile jest na niego miejsce).

Oprócz znaku '\0' łańcuchy mogą zawierać inne znaki specjalne (sekwencje sterujące). Była o tym mowa w rozdziale dotyczącym typu `char`.

15.2. Operacje na łańcuchach

Porównywanie łańcuchów

Zmienne reprezentujące napisy to tak naprawdę wskaźniki. Tak więc używając zwykłego operatora porównania ==, otrzymamy wynik porównania adresów, a nie tekstów.

Do porównywania dwóch ciągów znaków należy użyć funkcji `strcmp()` zadeklarowanej w pliku nagłówkowym `string.h`. Jako argument przyjmuje ona dwa napisy i zwraca wartość 0, jeżeli napisy są jednakowe. Jeśli pierwszy niepasujący znak w obu napisach ma kod ASCII o wartości większej od kodu odpowiadającego mu znaku w napisie drugim, funkcja zwraca wartość dodatnią, w przeciwnym przypadku – wartośc ujemną.

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

int main(void) {
      char str1[100], str2[100];
      int cmp;

      puts("Podaj dwa ciagi znakow: ");
      fgets(str1, sizeof str1, stdin);
      fgets(str2, sizeof str2, stdin);

      cmp = strcmp(str1, str2);
      if (cmp<0) {
            puts("Pierwszy napis jest 'mniejszy'.");
      } else if (cmp>0) {
            puts("Pierwszy napis jest 'wiekszy'.");
      } else {
            puts("Napisy sa takie same.");
      }
      return 0;
}
```

Jeśli chcemy porównać tylko początkowe fragmenty dwóch napisów pomocna jest funkcja `strncmp()`. W porównaniu do `strcmp()` przyjmuje ona jeszcze jeden argument, oznaczający maksymalną liczbę znaków do porównania:

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

int main(void) {
      char str[100];
      int cmp;

      fputs("Podaj ciag znakow: ", stdout);
      fgets(str, sizeof str, stdin);
      if (!strncmp(str, "Pro", 3)) {
            puts("Podany ciag zaczyna sie od 'Pro'.");
      }
      return 0;
```

```
}
```

Kopiowanie napisów

Do kopiowania ciągów znaków służy funkcja `strcpy()`, która kopiuje drugi napis w miejsce pierwszego. Musimy pamiętać, by w pierwszym łańcuchu było wystarczająco dużo miejsca.

```
char napis[100];
strcpy(napis, "Ala ma kota.");
```

Znacznie bezpieczniej jest używać funkcji `strncpy()`, która kopiuje co najwyżej tyle bajtów ile podano jako trzeci parametr. Uwaga! Jeżeli drugi napis jest za długi, funkcja nie kopiuje znaku null na koniec pierwszego napisu, dlatego trzeba to zrobić ręcznie:

```
char napis[100];
strncpy(napis, "Ala ma kota.", sizeof(napis) - 1);
napis[12]= '\0';
```

Funkcja `strlen()` oblicza długość łańcucha. Jej działanie polega na zliczaniu znaków aż do napotkania znaku '\0'.

Łączenie napisów

Do łączenia napisów służy funkcja `strcat()`, która konkatenuje dwa napisy i wynik umieszcza w napisie przekazanym jako pierwszy argument. Ponownie jak w przypadku `strcpy()` musimy zagwarantować, by w pierwszym łańcuchu było wystarczająco dużo miejsca.

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

int main(void) {
      char napis1[80] = "hello ";
      char *napis2 = "world";
      strcat(napis1, napis2);
      puts(napis1);
      return 0;
}
```

I ponownie jak w przypadku `strcpy()` istnieje funkcja `strncat()`, która skopiuje co najwyżej tyle bajtów ile podano jako trzeci argument i dodatkowo dopisze znak null:

```
strncat(napis1, napis2, sizeof napis1 - 1);
```

Możemy też wykorzystać trzeci argument do zapewnienia bezpiecznego wywołania funkcji kopiującej. W przypadku zbyt małej tablicy skopiowany zostanie fragment tylko takiej długości, na jaki starczy miejsca (uwzględniając, że na końcu trzeba dodać znak '\0'). Przy podawaniu ilości znaków należy także pamiętać, że łańcuch, do którego kopiujemy nie musi być pu-

sty, a więc część pamięci przeznaczona na niego jest już zajęta. Dlatego od rozmiaru całego łańcucha, do którego kopiujemy należy odjąć długość napisu, który już się w nim znajduje.

```
strncat(napis1, napis2, sizeof(napis1)-strlen(napis1)- 1);
```

15.3. Konwersje

Czasami zdarza się, że łańcuch można interpretować nie tylko jako ciąg znaków, lecz np. jako liczbę. Jednak, aby dało się taką liczbę wykorzystywać w obliczeniach musimy skopiować ją do pewnej zmiennej typu numerycznego. Aby ułatwić programistom tego typu zamiany powstał zestaw funkcji bibliotecznych. Należą do nich:

- `atol()`, `strtol()` – konertuje łańcuch na liczbę całkowitą typu long
- `atoi()` – konwertuje łańcuch na liczbę całkowitą typu int
- `atoll()`, `strtoll()` – konwertuje łańcuch na liczbę całkowitą typu long long (64 bity)
- `atof()`, `strtod()` – konwertuje łańcuch na liczbę typu double

Przykłady użycia:

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

int main (void)
{
      char *str1 = "1 ABC", *str2 ="1.23 56.89";
      char *ptr;
      int a,b;
      double d,e,f;

      a=atoll(str1); // atoi i atoll działa podobnie
      b=strtol(str1,&ptr,10); //ptr wskazuje na pozycję za przekonwertowaną
      liczbą, 10 to system dziesiętny
      printf("%d   %d\n", a, b);
      printf("%s\n",ptr);

      d=atof(str2);
      e=strtod(str2, &ptr); //ptr wskazuje na pozycję za przekonwertowaną
      liczbą,
      printf("%f   %f\n", d, e);
      f=atof(ptr);
      printf("%s   %f\n", ptr, f);

      return 0;
}
```

Wynik działania:

```
1   1
 ABC
1.23   1.23
 56.89   56.89
```

Czasami przydaje się też konwersja w drugą stronę, tzn. z liczby na łańcuch. Do tego celu może posłużyć funkcja `sprintf()` lub `snprintf()`. `sprintf()` jest bardzo podobna do `printf()`, tyle, że wyniki jej pracy zwracane są do pewnego łańcucha, a nie wyświetlane np. na ekranie monitora. `snprintf` dodatkowo przyjmuje jako argument wielkość bufora docelowego. Jeżeli funkcje zakończą się sukcesem zwracają liczbę znaków w tekście.

Przykłady użycia:

```
#include <stdio.h>

int main ()
{
      char str [50];
      int n, a=5, b=3;
      n=sprintf (str, "%d + %d = %d", a, b, a+b);
      printf ("[%s] jest lancuchem %d znakow\n",str,n);

      n=snprintf (str, 50, "Polowa %d to %d", 60, 60/2 );
      snprintf (str+n, 50-n, ", a polowa tego jest %d.", 60/2/2 ); // dopi-
      sanie tekstu do łańcucha str
      puts (str);

      return 0;
}
```

15.4. Operacje na znakach

Warto też powiedzieć w tym miejscu o operacjach na samych znakach jako elementach składowych napisów. Spójrzmy na poniższy program:

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

int main()
{
      int znak;
      while ((znak = getchar())!=EOF) {
            if( islower(znak) ) {
                  znak = toupper(znak);
            } else if( isupper(znak) ) {
                  znak = tolower(znak);
            }
            putchar(znak);
      }
      return 0;
}
```

Program ten zmienia we wczytywanym tekście wielkie litery na małe i odwrotnie. Wykorzystujemy funkcje operujące na znakach z pliku nagłówkowego `ctype.h`. `isupper()` sprawdza, czy znak jest wielką literą, natomiast `toupper()` zmienia znak (o ile jest literą) na wielką literę. Analogicznie jest dla funkcji `islower()` i `tolower()`.

15.5. Częste błędy

- pisanie do niezaalokowanego miejsca

```
char *tekst;
scanf("%s", tekst);
```

- zapominanie o kończącym napis nullu

```
char test[4] = "test"; /* nie zmieścił się null kończący napis */
```

- nieprawidłowe porównywanie łańcuchów

```
char tekst1[] = "jakis tekst";
char tekst2[] = "jakis tekst";
if( tekst1 == tekst2 ) { /* tu zawsze będzie fałsz bo == porównuje
adresy, należy użyć strcmp() */
...
}
```

15.6. Unicode

> Unicode – komputerowy zestaw znaków mający w zamierzeniu obejmować wszystkie pisma używane na świecie. Kody pierwszych 128 znaków pokrywają się z ASCII.

Do przechowywania znaków zakodowanych w Unicode powinno się korzystać z typu `wchar_t`. Jego domyślny rozmiar jest zależny od użytego kompilatora, lecz w większości zaktualizowanych kompilatorów powinny to być 2 bajty. Typ ten jest częścią języka C++, natomiast w C znajduje się w pliku nagłówkowym `stddef.h`.

Jaki rozmiar i jakie kodowanie

Unicode określa jedynie jakiej liczbie odpowiada jaki znak, nie mówi zaś nic o sposobie dekodowania. Jako że Unicode obejmuje 918 tys. znaków, zmienna zdolna pomieścić go w całości musi mieć przynajmniej 3 bajty. Niestety procesory nie funkcjonują na zmiennych o tym rozmiarze, pracują jedynie na zmiennych o wielkościach: 1, 2, 4 oraz 8 bajtów (kolejne potęgi liczby 2). Popularnym kodowaniem na 4 bajtach jest UTF-32. Ten typ kodowania po prostu przydziela każdemu znakowi Unicode kolejne liczby. Jest to najbardziej intuicyjny i wygodny typ kodowania, ale jak widać ciągi znaków zakodowane w nim są bardzo obszerne, co zajmuje dostępną pamięć, spowalnia działanie programu oraz drastycznie pogarsza wydajność podczas transferu przez sieć. Poza UTF-32 istnieje jeszcze wiele innych kodowań, np. UCS-2,

gdzie znak kodowany jest na dwóch bajtach, przez co znaki z numerami powyżej 65 535 nie są uwzględnione.

Co należy zrobić, by zacząć korzystać z kodowania UCS-2 (domyślne kodowanie dla C):

- powinniśmy korzystać z typu `wchar_t` (ang. *wide character*), jednak jeśli chcemy udostępniać kod źródłowy programu do kompilacji na innych platformach, powinniśmy ustawić odpowiednie parametry dla kompilatorów, by rozmiar był identyczny niezależnie od platformy

- korzystamy z odpowiedników funkcji operujących na typie `char` pracujących na `wchar_t` (z reguły składnia jest identyczna z tą różnicą, że w nazwach funkcji zastępujemy "`str`" na "`wcs`" np. `strcpy()` – `wcscpy()`, `strcmp()` – `wcscmp()`)

- jeśli przyzwyczajeni jesteśmy do korzystania z klasy `string` (tylko C++), powinniśmy zamiast niej korzystać z `wstring`, która posiada zbliżoną składnię, ale pracuje na typie `wchar_t`.

Przykład użycia kodowania UCS-2:

```
#include <stddef.h> /* jeśli używamy C++, możemy opuścić tę linijkę */
#include <stdio.h>
#include <string.h>

int main() {
      wchar_t* wcs1 = L"Ala ma kota.";
      wchar_t* wcs2 = L"Kot ma Ale.";
      wchar_t calosc[25];

      wcscpy(calosc, wcs1);
      *(calosc + wcslen(wcs1)) = L' ';
      wcscpy(calosc + wcslen(wcs1) + 1, wcs2);

      printf("lancuch wyjsciowy: %ls\n", calosc);
      return 0;
}
```

Zauważ, że gdy używamy typu `wchar_t` przed łańcuchem dostawiamy literę `L`. Dla typu `wchar_t` używamy sekwencji formatującej `%ls`.

16. TYPY ZŁOŻONE

16.1. typedef

`typedef` to słowo kluczowe, które służy do definiowania typów pochodnych, np.:

```
typedef stara_nazwa  nowa_nazwa;
typedef int mojInt;
```

```
typedef int* WskNaInt;
```

Od tej pory można używać typów `mojInt` i `WskNaInt`, np. deklarując zmienne:

```
mojInt a,b,*c;
WskNaInt x,y; //x i y to wskaźniki
```

16.2. Typ wyliczeniowy

Służy do tworzenia zmiennych, które powinny przechowywać tylko pewne z góry ustalone wartości:

```
enum Nazwa_typu {WARTOSC_1, WARTOSC_2, WARTOSC_N};
```

Na przykład można w ten sposób stworzyć zmienną przechowującą kierunek:

```
enum Kierunek {W_GORE, W_DOL, W_LEWO, W_PRAWO};
enum Kierunek ruch = W_GORE;
```

Gdzie `Kierunek` to typ zmiennej, a `ruch` to nazwa zmiennej, o takim typie. Zmienną tę można teraz wykorzystać na przykład w instrukcji switch:

```
switch(ruch)
{
      case W_GORE:
            printf("w górę\n");
            break;
      case W_DOL:
            printf("w dół\n");
            break;
      default:
            printf("gdzieś w bok\n");
}
```

Tradycyjnie wartości wyliczeniowe zapisuje się wielkimi literami (`W_GORE`, `W_DOL`, ...).

Tak naprawdę C przechowuje wartości typu wyliczeniowego jako **liczby całkowite** (zakres typu `signed int`), o czym można się łatwo przekonać:

```
ruch = W_DOL;
printf("%i\n", ruch); /* wypisze 1 */
```

Kolejne wartości wyliczeniowe to po prostu liczby całkowite. Pierwsza to zero, druga jeden itp. Możemy przy deklarowaniu typu wyliczeniowego zmienić domyślne przyporządkowanie:

```
enum Kierunek { W_GORE, W_DOL = 8, W_LEWO, W_PRAWO };
printf("%i %i\n", W_DOL, W_LEWO); /* wypisze 8 9 */
```

Co więcej liczby mogą się powtarzać i wcale nie muszą być ustawione w kolejności rosnącej:

```
enum Kierunek { W_GORE = 5, W_DOL = 5, W_LEWO = 2, W_PRAWO = -1 };
printf("%i %i\n", W_DOL, W_LEWO); /* wypisze 5 2 */
```

Traktowanie przez kompilator typu wyliczeniowego jako liczby pozwala na wydajną ich obsługę, ale stwarza niebezpieczeństwa – można przypisywać pod zmienne wyliczeniowe liczby

nie mające odpowiednika w wartościach wyliczeniowych, a kompilator może o tym nawet nie ostrzec:

```
ruch = 40;
```

16.3. 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.

Przykład definicji struktury:

```
struct Struktura {
      int pole1;
      int pole2;
      char pole3;
};
```

gdzie `Struktura` to nazwa tworzonej struktury. Nazwy zmiennych, ilość i typ pól definiuje programista według potrzeb.

Zmienną posiadającą strukturę tworzy się podając jako jej typ nazwę struktury:

```
struct Struktura zmiennaS;
```

Dostęp do poszczególnych pól uzyskuje się przy pomocy operatora wyboru składowej – kropki (`.`):

```
zmiennaS.pole1 = 60;    /* przypisanie liczb do pól */
zmiennaS.pole2 = 2;
zmiennaS.pole3 = 'a';   /* a teraz znaku */
```

Jeśli zmienna strukturalna jest wskaźnikiem, aby dostać sie do pola zamiast kropki używamy operatora `->`, np. `wskaznikS->pole1 = 60`.

Przykłady struktur:

```
struct punkt
{
      int x,y;
};

struct ksiazka
{
      char *tytul, *autor;
      unsigned int rok_wyd;
} a, b = {"Potop","Sienkiewicz Henryk",1988}, *c;
```

W drugim przykładzie pokazano, że zmienne można deklarować zaraz za definicją struktury. Przykłady deklaracji zmiennych strukturalnych i dostępu do składowych struktury:

```
struct punkt *p, p1 = {23,45}, t_p[10];
struct ksiazka d = {"Genomy","Brown"}, e[200];
```

```
p=&p1;
p->x = 66; // można też tak (*p).x = 66;
p1.y = 98;
t_p[0].x = 88;

b.tytul = "Ogniem i mieczem";
c=&a;
c->rok_wyd = 1999;
e[2].autor = "Nowak";
```

Pole struktury może być dowolnego typu (także strukturalnego), z pewnymi wyjątkami.

Przykład użycia typu strukturalnego:

```
#include <stdio.h>
#define PI 3.14

struct punkty {
      int x, y;
};

struct prostokat {
      int a;
      int b;
} pr1 = {3, 12}, pr2;

struct {
      float r,o,p;
} 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("Pole %d\n", pr1.a*pr1.b);

      kolo.r = 3.45;
      kolo.o = 2*PI*kolo.r;
      kolo.p = PI*kolo.r*kolo.r;
      printf("Obwód: %.2f\tPole: %.2f\n", kolo.o, kolo.p);
      return 0;
}
```

Zauważ, że nazwę typu strukturalnego można pominąć (trzecia struktura), wystarczy podać nazwę zmiennej po zamykającym nawiasie klamrowym. Lecz w tym przypadku nie zadeklarujemy więcej zmiennych tego typu strukturalnego

Poniżej pokazano przykład funkcji operującej na zmienenych strukturalnych:

```
struct prostokat fun_p(int  x, int y)
{
```

```
        struct prostokat w;
        w.a=x;
        w.b=y;
        return w;
    }
```

Przykładwe wywołanie funkcji:

```
    pr2 = fun_p(3,5);
```

Funkcja `fun_p` zwraca wartość zmiennej strukturalnej typu `prostokat` – `w`. Polom tej zmiennej (`a` i `b`) przypisuje się wartości argumentów `x` i `y`. Tę funkcję można zdefiniować w inny sposób:

```
    void fun_p(struct prostokat *w, int  x, int y)
    {
        w->a=x;
        w->b=y;
    }
```

Wywołanie:

```
    fun_p(&pr2,3,5);
```

Struktury można zagnieżdżać. Demonstruje to poniższy przykład.

```
#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;
}
```

Używając typedef możemy skrócić zapis programu, definiując krótką nazwę dla typu strukturalnego:

```
#include <stdio.h>

typedef struct pole_ab {
      int a, b;
} Bok; // krótka nazwa stosowana dalej zamiast struct pole_ab

int f_pole (Bok pole){ // definicja funkcji
      return pole.a * pole.b;
}

int main (void)
{
      typedef int Licznik;
      typedef int Dlugosc;

      Bok pole; // to samo co struct pole_ab pole;
      Licznik i; // to samo co int i;
      Dlugosc max = 15; // to samo co 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;
}
```

16.4. Unie

Definicja uni jest podobna do definicji struktury:

```
union Nazwa {
      typ1 nazwa1;
      typ2 nazwa2;
      /* ... */
};
```

Na przykład:

```
union LiczbaLubZnak {
      int calkowita;
      char znak;
      double rzeczywista;
};
```

Struktura zajmuje w pamięci sumaryczną ilość bajtów zajmowanych przez jej poszczególne pola. Z unią jest trochę inaczej, ponieważ unia zajmuje tyle miejsca ile zajmuje największe z jej pól. W jednym czasie może przechowywać wartość tylko jednego pola. W powyższym przypadku unia będzie miała prawdopodobnie rozmiar typu double, czyli np. 8 bajtów. Pole

`całkowita` wykorzystuje pierwsze cztery bajty, a pole znak – pierwsze dwa bajty. Do pól uni odwołujemy się tak samo jak do pól struktury. Unia w jednej chwili może posiadać tylko jedną wartość konkretnego pola, co zostało pokazane na poniższym listingu.

```
#include <stdio.h>

union dane {
      unsigned char i;
      double x;
      unsigned char tab[6];
};

int main (void)
{
      int i;
      union dane u_data = {10}; // inicjalizacja pola x: u_data = {x:2.7};
      // inicjalizacja pola tab: u_data = {tab:{'a','b','c','d','e','f'}};
      printf("Addr u data: %p Rozm: %d\n", &u data, sizeof (u data));
      printf("Addr u_data.i: %p Rozm: %d\n", &u_data.i, sizeof (u_data.i));
      printf("Addr u_data.x: %p Rozm: %d\n", &u_data.x, sizeof (u_data.x));
      printf("Addr u_data.tab: %p Rozm: %d\n\n", &u_data.tab, sizeof
      (u_data.tab));

      printf("Co kryje sie w u_data.i: %d\n", u_data.i);

      u_data.x = 813.14;
      printf("Co kryje sie w u_data.x: %.2f\n", u_data.x);
      printf("a co teraz jest w u_data.i: %d - smieci\n\n", u_data.i);

      for (i = 0; i < sizeof (u_data.tab)/sizeof (u_data.tab[0]); ++i){
            u_data.tab[i] = 10*i;
            printf("Co kryje sie w u_data.tab[%d]: %d\n", i,
      u_data.tab[i]);
      }
      printf("a co teraz jest w u_data.x: %.2f - smieci\n", u_data.x);
      printf("a w u_data.i: %d - smieci\n\n", u_data.i);

      u_data.x = 1e-8;
      printf("u_data.x = %.8f\n", u_data.x);
      printf("Zawartosc poszczegolnych bajtow:\n");
      for (i = 0; i<8; ++i)
            printf("%d - smieci\n", *(&u_data.i+i));

      return 0;
}
```

Sprawdź jak ten program zadziała, gdy słowo `union` zamienisz na `struct`.

Zazwyczaj użycie unii ma na celu zmniejszenie zapotrzebowania na pamięć.

16.5. Pola bitowe

Struktury mają pewne dodatkowe możliwości w stosunku do zmiennych. Rozmiar pola struktury może mieć dowolną liczbę bitów, nawet 1 bit! Pola bitowe deklarowane są jako liczby

całkowite bez znaku (`unsigned`) lub ze znakiem (`signed`). Przykładowa definicja struktury z polami bitowymi:

```
struct X
{
      signed a: 2; // 2 bity, osiągane wartości: {-1, 0, 1}
      signed b: 5; // 5 bitów, {-16, -14, ..., 15}
      signed: 4;
      unsigned c: 3; // trzy bity, {0, 1, ..., 7}
};
```

Rozmieszczenie pól bitowych w pamięci w tym przypadku:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		c							b					a	

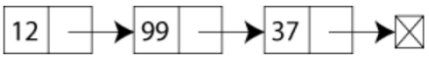
Wszystkie pola tej struktury mają w sumie rozmiar 14 bitów (maksymalny rozmiar struktury z polami bitowymi to 16 bitów). Możemy odwoływać się do nich w taki sam sposób, jak do innych elementów struktury. W ten sposób efektywniej wykorzystujemy pamięć.

Dostęp do pól bitowych:

```
Struci X z;
z.a=0;
z.b=12;
z.c=6;
```

16.6. Studium przypadku - implementacja listy wskaźnikowej

Lista to struktura danych służąca do reprezentacji zbiorów dynamicznych, w której elementy ułożone są w liniowym porządku. Rozróżniane są dwa podstawowe rodzaje list: lista jednokierunkowa, w której z każdego elementu możliwe jest przejście do jego następnika oraz lista dwukierunkowa, w której z każdego elementu możliwe jest przejście do jego poprzednika i następnika.



Rysunek: Lista jednokierunkowa.

Każdy element listy składa się z co najmniej dwóch pól: klucza oraz pola wskazującego na następny element listy. W przypadku list dwukierunkowych każdy element listy zawiera także pole wskazujące na poprzedni element listy. Pole wskazujące poprzedni i następny element listy są najczęściej wskaźnikami.

W języku C, aby stworzyć listę musimy użyć struktur, np.:

```
typedef struct element {
      struct element *next; /* wskaźnik na kolejny element listy */
      unsigned long val; /* przechowywana wartość, klucz */
} el_listy;
```

Rozważmy przykład – wyszukujemy kolejne liczby pierwsze i zapisujemy je w kolejnych elementach listy (Liczba pierwsza – liczba naturalna, która ma dokładnie dwa dzielniki natu-

ralne: jedynkę i siebie samą). Z matematyki wiemy, że dana liczba jest liczbą pierwszą, jeśli nie dzieli się przez żadną liczbę pierwszą ją poprzedzającą, mniejszą od jej pierwiastka. Pierwszą liczbą pierwszą jest liczba 2. Pierwszym elementem naszej listy będzie zatem struktura, która będzie przechowywała liczbę 2. Na co będzie wskazywało pole `next`? Ponieważ na początku działania programu będziemy mieć tylko jeden element listy, pole `next` powinno wskazywać na `NULL`. Po tym poznamy, że mamy do czynienia z ostatnim elementem listy.

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

typedef struct element {
      struct element *next;
      unsigned long val;
} el_listy;

el_listy *first;

//deklaracje funkcji
void dodaj_do_listy (el_listy *wsk, unsigned long liczba);
void wypisz_liste(el_listy *wsk);
int jest_pierwsza(el_listy *wsk, int liczba);

int main ()
{
      unsigned long i = 3; /* szukamy liczb pierwszych w zakresie od 3 do
      1000 */
      const unsigned long END = 1000;

      first = (el_listy*) malloc (sizeof(el_listy)); /* alokacja pamięci
      dla pierwszego elementu */
      first->val = 2; /* wpisanie wartości do pierwszego elementu */
      first->next = NULL;
      for (;i!=END;++i) {
            if (jest_pierwsza(first, i))
                  dodaj_do_listy (first, i);
      }
      wypisz_liste(first);
      return 0;
}

int jest_pierwsza(el_listy *wsk, int liczba)
{
      while (wsk != NULL) {
            if (wsk->val<=sqrt(liczba) && (liczba%wsk->val)==0) return 0;
            /* jeśli reszta z dzielenia liczby przez którąkolwiek z po-
            przednio znalezionych liczb pierwszych, mniejszych od jej pier-
            wiastka jest równa zero, to znaczy, że liczba ta nie jest licz-
            bą pierwszą */
            wsk = wsk->next;
      }
      return 1;
}

void dodaj_do_listy (el_listy *wsk, unsigned long liczba)
{
      el_listy *nowy;
      while (wsk->next != NULL)
      {
```

```
            wsk = wsk->next; /* przesuwamy wsk aż znajdziemy ostatni ele-
      ment */
      }
      nowy = (el_listy*) malloc (sizeof(el_listy)); /* alokacja pamięci dla
      nowego elementu */
      nowy->val = liczba; /* wpisanie wartości do nowego elementu */
      nowy->next = NULL;
      wsk->next = nowy; /* adres nowego elementu zapamiętujemy w elemenncie
      poprzedzającym */
}

void wypisz_liste(el_listy *wsk)
{
      while( wsk != NULL )
      {
            printf ("%lu\n", wsk->val);
            wsk = wsk->next;
      }
}
```

W celu wypisania listy na ekranie (`wypisz_liste()`) odwiedzamy każdy element listy. Elementy listy są połączone polem `next`. Aby przeglądnąć listę użyjemy następującego algorytmu:

1. Ustaw wskaźnik roboczy na pierwszym elemencie listy.
2. Jeśli wskaźnik ma wartość NULL, przerwij.
3. Wypisz element wskazywany przez wskaźnik.
4. Przesuń wskaźnik na element, który jest wskazywany przez pole `next`.
5. Wróć do punktu 2.

Agorytm dodawania do listy nowego elementu (`dodaj_do_listy ()`):

1. Znaleźć ostatni element (tj. element, którego pole `next` == NULL).
2. Przydzielić odpowiedni obszar pamięci.
3. Skopiować w pole `val` w nowo przydzielonym obszarze znalezioną liczbę pierwszą.
4. Nadać polu `next` ostatniego elementu listy wartość `NULL`.
5. W pole `next` przedostatniego elementu listy wpisać adres nowo przydzielonego obszaru.

A w jaki sposób usuwać element z listy? Aby usunąć element potrzebujemy wskaźnika do elementu go poprzedzającego (po to, by nie rozerwać listy). Popatrzmy na poniższą funkcję:

```
void usun_z_listy(el_listy *lista, int element)
{
      el_listy *wsk=lista;
      while (wsk->next != NULL)
      {
```

```
            if (wsk->next->val == element) /* musimy mieć wskaźnik do ele-
            mentu poprzedzającego */
            {
                  el_listy *usuwany=wsk->next; /* zapamiętujemy usuwany ele-
                  ment */
                  wsk->next = usuwany->next; /* przestawiamy wskaźnik next
                  by omijał usuwany element */
                  free(usuwany); /* usuwamy z pamięci */
            } else
            {
                  wsk = wsk->next; /* idziemy dalej tylko wtedy kiedy nie
                  usuwaliśmy */
            } /* bo nie chcemy zostawić duplikatów */
      }
}
```

Funkcja ta jest tak napisana, by usuwała z listy wszystkie wystąpienia danego elementu (w naszym programie nie ma to miejsca, ale lista jest zrobiona tak, że może przechowywać dowolne liczby). Zauważmy, że wskaźnik `wsk` jest przesuwany tylko wtedy, gdy nie kasowaliśmy. Gdybyśmy zawsze go przesuwali, przegapilibyśmy element, gdyby występował kilka razy pod rząd.

Cały powyższy przykład omawiał tylko jeden przypadek listy – listę jednokierunkową. Jednak istnieją jeszcze inne typy list, np.: lista jednokierunkowa cykliczna, lista dwukierunkowa oraz dwukierunkowa cykliczna. Różnią się one od siebie tylko tym, że:

- w przypadku list dwukierunkowych w strukturze `el_listy` znajduje się jeszcze pole, które wskazuje na element poprzedni,
- w przypadku list cyklicznych ostatni element wskazuje na pierwszy (nie rozróżnia się wtedy elementu pierwszego, ani ostatniego)

17. FUNKCJE MATEMATYCZNE

Funkcje matematyczne wymagają dołączenia zbioru nagłówkowego `math.h` [Pys]. Wszystkie funkcje opisane w tym rozdziale zwracają wartość typu `double`. Argumenty `x` i `y` są typu `double`, a argument `n` typu int. Wartości kątów funkcji trygonometryczny wyraża się w **radianach**.

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)	$\|x\|$
-	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.
-	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;

Przykład – Oblicz wyrażenie podane poniżej oraz wyświetl wynik zaokroglony w dół i w górę

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

```
#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 + 2.0*p2;
	mianownik = p3 + 2.0*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);
}
```

W pliku `math.h` zdefiniowane są pewne stałe, które mogą być przydatne do obliczeń. Są to m.in.:

- `M_E` – podstawa logarytmu naturalnego (e, liczba Eulera)
- `M_LOG2E` – logarytm o podstawie 2 z liczby e
- `M_LOG10E` – logarytm o podstawie 10 z liczby e
- `M_LN2` – logarytm naturalny z liczby 2
- `M_LN10` – logarytm naturalny z liczby 10
- `M_PI` – liczba π
- `M_PI_2` – liczba π/2
- `M_PI_4` – liczba π/4
- `M_1_PI` – liczba 1/π
- `M_2_PI` – liczba 2/π

W obliczeniach matematycznych przydatne mogą okazać się funkcje do generowania liczb pseudolosowych. Standardowa biblioteka języka C w pliku nagłówkowym `stdlib.h` posiada zadeklarowane dwie funkcje służące do tego celu `srand()` i `rand(void)`.

Funkcja `srand()` jako parametr przyjmuje tzw. ziarno (seed). Jest to liczba typu `unsigned int` na podstawie, której inicjowany jest generator liczb pseudolosowych.

Funkcja `rand()` służy do losowania liczb pseudolosowych. Zwraca ona liczbę całkowitą z przedziału miedzy 0 a RAND_MAX. RAND_MAX jest to stała zadeklarowana w pliku nagłówkowym.

Przykład – Wygeneruj 10 liczb pseudolosowych, całkowitych odpowiadających kodom znaków z zakresu A–Z.

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

int main()
{
      int i,c;
      /* Uruchamiamy generator liczb pseudolosowych jako
      parametr podajemy funkcje time(NULL), która zwraca czas
      w sekundach liczony do 1970 roku */
      srand(time(NULL));

      for(i=0; i<=9; i++)
      {
            c = (rand() % 25) + 65; //losowanie liczb z przedzialu 65 - 90,
            czyli kody ASCII A-Z
            printf("Wylosowana liczba nr [%d]: %d odpowiada jej znak ASCII
            %c\n",i,c,c);
      }
      return (0);
}
```

Aby wygenerować liczbę całkowitą z zakresu [x, y] używamy instrukcji:

```
rand()%(y-x)+x;
```

Aby wygenerować liczbę rzeczywistą z zakresu [x, y] używamy instrukcji:

```
(double)rand()/RAND_MAX*(y-x)+x;
```

18. PRZYKŁADOWE PROGRAMY W C

18.1. Pierwiastki równania kwadratowego

Program, który oblicza pierwiastki równania kwadratowego. Współczynniki wielomianu wprowadza użytkownik.

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

main()
{
  double a,b,c,delta,x1,x2;
```

```
  printf("Podaj a, b, c: ");
  scanf("%lf %lf %lf",&a,&b,&c);

  delta=b*b-4*a*c;
  if (delta<0) return 0;
  else
  {
      x1=(-b-sqrt(delta))/(2*a);
      x2=(-b+sqrt(delta))/(2*a);
  }
  printf("Pierwiastki: %lf, %lf\n",x1,x2);    // %10.2lf - wyświetlanie na
10 miejscach z dokładnością do 2 po przecinku

  return 0;
}
```

Druga wersja programu wykorzystuje makrodfinicje.

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

#define DEL    (b*b-4*a*c)
#define X1     ((-b-sqrt(DEL))/(2*a))
#define X2     ((-b+sqrt(DEL))/(2*a))

main()
{
  double a,b,c;

  printf("Podaj a, b, c: ");
  scanf("%lf %lf %lf",&a,&b,&c);

  a==0 ? printf("Blad a=0") : (DEL<0 ? printf("Brak pierwiastkow del-
ta=%.2lf",DEL) : printf("x1=%.2lf x2=%.2lf",X1,X2));

  getch();
  return 0;
}
```

18.2. Zapis daty i czasu w pliku tekstowym

Działanie programu:

1. Pobranie aktualnego czasu i daty.
2. Dopisanie daty i czasu do pliku tekstowego o podanej nazwie.
3. Odczyt z pliku tekstowego wcześniejszych wpisów.

Uwagi i wskazówki:

- `time_t czas2` – deklaracja wskaźnika na zmienną zawierającą czas jaki upłynął od godziny 00:00:00, 1 stycznia 1970 r.

- `struct tm *czas1` – deklaracja zmiennej strukturalnej `tm`

```
struct tm {
    int     tm_sec;         /* sekundy od pełnej minuty*/
    int     tm_min;         /* minuty  od pełnej godziny*/
    int     tm_hour;        /* godzina na 24-godzinnym zegarze*/
    int     tm_mday;        /* dzień miesiąca */
    int     tm_mon;         /* miesiąc licząc od zera */
    int     tm_year;        /* rok - 1900 */
    int     tm_wday;        /* dzień tygodnia, niedziela ma numer 0*/
    int     tm_yday;        /* dzień roku licząc od zera*/
    int     tm_isdst;       /* znacznik czasu letniego */
};
```

- `czas1=localtime(&czas2)` – konwersja czasu zapisanego w zmiennej typu `timt_t` na strukturę `tm`.

- `L = asctime(czas1)` – konwersja czasu zapisanego w strukturze `tm` na ciąg znaków.

- `fopen(scieżka,"a+")` – otwarcie pliku w trybie dopisywania.

- `fgets(L,n,file1)` – odczyt całego wiersza z pliku `file1` i umieszczenie go pod wskaźnikiem `L`. Maksymalna liczba znaków w wierszu – `n`. Jeśli wystąpił błąd lub napotkano koniec pliku funkcja zwraca adres pusty `NULL`.

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

 main() {
      time_t czas2;
      struct tm *czas1;
      FILE *file1;
      char *L,odp,nazwa[30]="data.txt";

      czas2 = time(NULL);
      czas1=localtime(&czas2);
      L = asctime(czas1);

      printf("Dzisiejsza data: %s",L );
      printf("\nDo jakiego pliku zapisac (data.txt)? ");
      if (*gets(nazwa)=='\0') nazwa[0]='d'; //gets(nazwa);
                                  // if (nazwa[0]=='\0') nazwa[0]='d';

      printf("\n Sciezka do pliku : %20s",nazwa);

      printf("\nData ta zostala zapisana do pliku textowego :%s",nazwa);
      file1=fopen(nazwa,"a+");
      fprintf(file1,"%s",L);
      fclose(file1);

      printf("\n\n\nCzy chcesz obejrzec wczesniejsze wpisy ? (t/n)");
```

```
      scanf("%s",&odp);

      if (odp=='t') {
        file1=fopen(nazwa,"r");
        while (fgets(L,80,file1)!=NULL)
          printf("%s",L);
        fclose(file1);
    }

      return 0;
}
```

18.3. Zliczanie znaków w pliku tekstowym

Działanie programu:

1. Podajemy ścieżkę dostępu do pliku tekstowego.
2. Program zlicza a) wszystkie znaki w tym pliku, b) białe znaki i c) słowa.
3. Podajemy znak z klawiatury, program zlicza ile takich znaków jest w pliku.
4. Wyświetlana jest statystyka znaków: kod znaku – znak – liczba wystąpień.

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

main()
{
  char znak,poprz='q',znak1,sciezka[100];
  int i;
  long l_zn=0,l_bz=0,l_sl=0,l_tz=0,tz[256];
  FILE *In;

  // p. 1
  printf("Podaj sciezke do pliku: ");
  scanf("%s",sciezka);
  if ((In=fopen(sciezka,"rb"))==NULL)
  {
    printf("Blad otwarcia strumienia");
    return 0;
  }

  // p. 2
  while ((znak=getc(In))!=EOF)
  {
    l_zn++;
    if (znak==' '||znak=='\n'||znak=='\t')
    {
      l_bz++;
      if (poprz!=' '&&poprz!='\n'&&poprz!='\t') l_sl++;
    }
    poprz=znak;
  }
  if (poprz!=' '&&poprz!='\n'&&poprz!='\t') l_sl++;
```

```
  printf("Liczba znakow: %ld, liczba bialych znakow: %ld,\
        \nliczba slow: %ld ",l_zn,l_bz,l_sl);

  // p. 3
  while (getch()==13)
  {
      rewind(In); //=fsek(In,0L,SEEK_SET) przesunięcie do początku pliku
      printf("\nPodaj znak: ");
      znak1=getch();
      while ((znak=getc(In))!=EOF)
            if (znak==znak1) l_tz++;
      printf("Liczba znakow \'%c\' - %ld",znak1,l_tz);
      l_tz=0;
  }

  // p. 4
  rewind(In);

  for(i=0;i<=255;++i) tz[i]=0;
  while ((znak=getc(In))!=EOF)
      for (i=0;i<=255;i++)
            if ((int)znak==i) tz[i]++;

  printf("\nStatystyka znakow\n");
  for (i=0;i<=255;i++)
      if (tz[i]!=0)
            printf("%3d  -  \'%1c\'  -  %d\n",i,i,tz[i]);

  fclose(In);
  return 0;
}
```

18.4. Sortowanie liczb

Działanie programu:

1. Generujemy losowo N liczb rzeczywistych z zakresu od 0 do zadanej wartości, zapamiętując je w tablicy dynamicznej.
2. Sortujemy te liczby dowolną metodą.
3. Mierzymy czas sortowania.
4. Zapisujemy liczby posortowane do pliku tekstowego.

Uwagi i wskazówki:

- `rand()` – funkcja losująca liczby z zakresu 0 – `RAND_MAX`,
- `czas=clock()` – pomiar czasu jaki upłynął od uruchomienia programu w impulsach (przeliczenie na sekundy: `(double)czas/CLOCKS_PER_SEC`. Zmienna `czas` musi być typu `clock_t`.

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

```
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <ctime>

double *tab,g;
int N,t,kier,i,j;
FILE *P;
char sciezka[100];
clock_t czas;

int sortf(double *a, double *b);

main()
{
  //losowanie liczb
  do{
      printf("Ile liczb posortowac (<100000): ");
      scanf("%d",&N);
  }while(N>100000);

  printf("Zakres liczb od 0 do: ");
  scanf("%lf",&g);

  tab=(double*)malloc(N*sizeof(double));//dynamiczny przydział pamięci

  for (i=0; i<=N-1; ++i){
    tab[i]=(double)rand()/RAND_MAX*g;
    printf("%lf\n",tab[i]);
  }

  // sortowanie
  puts("Podaj kierunek sortowania: 0 - od najmniejszej do najwiekszej, in-
na - od najwiekszej do najmniejszej:");
  scanf("%d",&t);
  kier=(t==0)?-1:1;

  czas=clock(); //początek odliczania czasu
  for (i=0;i<N;i++)
    for (j=i;j<N;j++)
      if (sortf(&tab[i],&tab[j])==kier)
      {
          g=tab[i];
          tab[i]=tab[j];
          tab[j]=g;
      }
  czas=clock()-czas; //koniec odliczania czasu
  printf("Czas sortowania: %.10lf sekund\n",(double)czas/CLOCKS_PER_SEC);

  //zapis do pliku
  printf("Podaj sciezke do pliku: ");
  scanf("%s",sciezka);
  if ((P=fopen(sciezka,"w"))==NULL)
  {
    printf("Blad otwarcia strumienia");
    return 0;
  }

  for (i=0; i<=N-1; ++i)
    fprintf(P,"%lf\n",tab[i]);
```

```
  free(tab);
  fclose(P);

  return 0;
}

int sortf(double *a, double *b)
{
      if (*a>*b) return -1;
      else if (*a<*b) return 1;
           else return 0;
      //return (*a>*b)?-1:((*a<*b)?1:0);
}
```

18.5. Generacja dziwnego atraktora

Program generuje atraktor Lorentza (http://pl.wikipedia.org/wiki/Układ_Lorenza):

$$dx/dt = -ax + ay$$

$$dy/dt = -xz + rx - y$$

$$dz/dt = xy - bz$$

dla a = 10, b = 8/3, r = 28.

Wyniki (kolejne punkty na płaszczyźnie XY, XZ, YZ – wyboru układu współrzędnych dokonuje użytkownik) zapisywane są do pliku graficznego .pbm.

Uwagi i wskazówki:

Z powyższych wzorów wyznacz wzory na dx, dy i dz (czyli przyrosty współrzędnych punktu). Załóż stały krok czasowy dt, załóż początkową pozycję punktu (x, y, z). W pętli generuj ciąg punktów (x + dx, y + dy, z + dz) zaznaczając je na płaszczyźnie. Zapisz obraz atraktora w pliku .pbm (http://pl.wikipedia.org/wiki/Portable_anymap).

```
#include <stdio.h>
#define M 300000 //liczba iteracji
#define DIM 2000 //rozmiary obrazu
#define MIN -50 //minimalna współrzędna
#define MAX 50 //maksymalna współrzędna

char T[DIM][DIM]; //tablica przechowujaca obraz

int main() {
      int i,x1,y1,z1;
      double dx,dy,dz,x,y,z;
      const double a = 10.0, b = 8.0/3, r = 99.96, dt=0.001;
      char u;

      //punkt poczatkowy
      x=1; y=0; z=0;
```

```
	do{
		puts("Podaj układ wspolrzednych: 1 - x/y, 2 - y/z, 3 - x/z ");
		scanf("%d",&u);
	}while((u<1)||(u>3));

	//iterowanie odwzorowania, wyznaczenie współrzędnych pikseli
	for (i=0;i<M;++i){
		dx = (-a*x + a*y)*dt;
		dy = (-x*z + r*x -y)*dt;
		dz = (x*y - b*z)*dt;

		x=x+dx;
		y=y+dy;
		z=z+dz;

		x1=(x-MIN)*DIM/(MAX-MIN);
		y1=(y-MIN)*DIM/(MAX-MIN);
		z1=(z-MIN)*DIM/(MAX-MIN);

		if (u==1 ) T[x1][y1]=1;
		else if (u==2) T[y1][z1]=1;
			else T[x1][z1]=1;
	}

	// zapisanie tablicy T w pliku graficznym .pbm
	FILE * fp = fopen("first.pbm", "w");
	fprintf(fp, "P1\n%d %d\n", DIM, DIM);
	for(int j=0; j<DIM; ++j){
		for(i=0; i<DIM; ++i)
			fprintf(fp,"%d ",T[i][j]);
		fputc('\n',fp);
	}

	fclose(fp);

	return 0;
}
```