Podstawy programowania – język C

Marian Soida

Obserwatorium Astronomiczne UJ

Astronomia, rok II

http://www.oa.uj.edu.pl/~soida/wyklady/C/wyklad.pdf

http://www.oa.uj.edu.pl/~soida/wyklady/C/progs/

wstęp organizacyjny

cel kursu

poznanie podstaw programowania w stopniu pozwalającym na samodzielne napisanie programu, mogącego służyć przetwarzaniu danych zebranych np. podczas ćwiczeń z pracowni fizycznej, obserwacyjnej.

literatura

- liczne źródła internetowe – samouczki, kursy, podręczniki
- liczne podręczniki drukowane
- B. W. Kernighan & D. M. Riche, „Jezyk C", WNT 1988
- S. Oualline, „Język C, Programowanie", O'Reilly 2003 (wyd. 3.)

wstęp organizacyjny

co potrzeba do zajęć

- edytor tekstowy do pisania tekstów programów
- kompilator C (jeśli Linux to mamy gcc)
- terminal do uruchamiannia edytora, kompilatora i programów
- chęć nauczenia się czegoś (!)

warunki zaliczenia

- obecność i aktywność na zajęciach
- pozytywny wynik sprawdzianu końcowego

dlaczego język C

- łatwo dostępny kompilator
- efektywny kod wynikowy
- prosta i zwięzła składnia
- wielość zastosowań
- wsteczna zgodność wersji

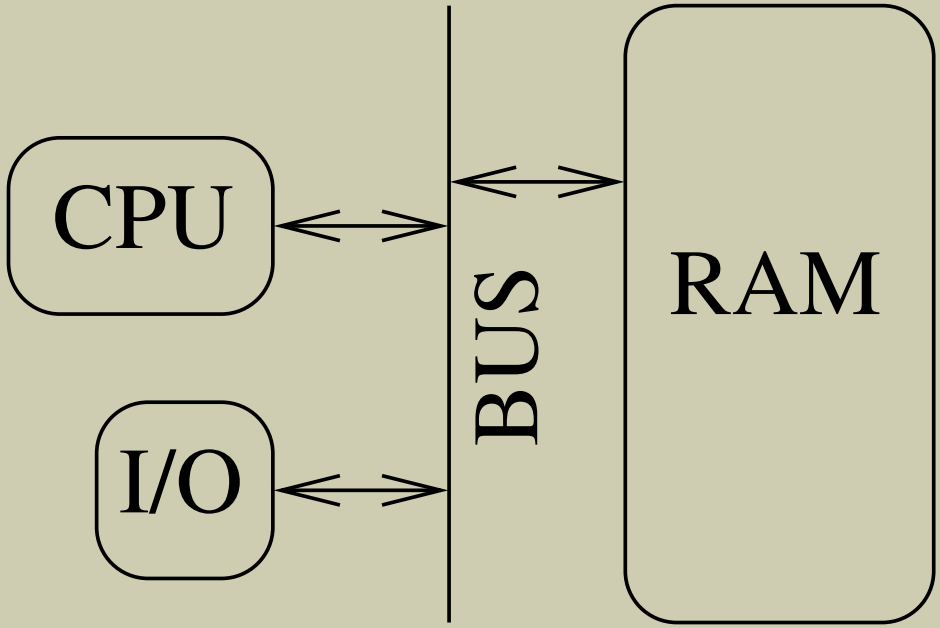
„W książce przedstawiono język C, będący nowoczesnym i uniwersalnym narzędziem programowania zrealizowanym na większość mini- i mikrokomputerów. Ze względu na swoją prostotę może być używany do pisania programów numerycznych, przetwarzających teksty lub obsługujących bazy danych. Ważną zaletą języka C jest uniezależnienie go od konkretnego komputera czy systemu operacyjnego. Programy napisane w języku C można więc bez zmian uruchamiać na różnych komputerach z różnymi systemami operacyjnymi." (K.&R., 1978)

co to jest i co robi kompilator

- tłumaczy program z języka zrozumiałego dla programisty do postaci „zrozumiałej” dla procesora (systemu operacyjnego)
  - znajduje błędy składni
  - generuje tzw. kod wynikowy
  - rezerwuje pamięć dla obiektów programu
  - łączy różne kawałki programu w całość
  - dołącza funkcje z bibliotek

C jest językiem niskiego poziomu

- dobrze jest znać budowę i działanie komputera



- dane są przetwarzane w procesorze i przechowywane w pamięci
- pobierane są z pamięci lub urządzeń wejścia/wyjścia

budowa języka C (składnia)

program

- zestaw funkcji, jedna z nich nazywa się „main”

funkcja

- nagłówek
  - typ zwracanej wartości
  - identyfikator
  - lista parametrów formalnych i ich typów – w nawiasach „()”, oddzielone przecinkami
- definicja
  - blok instrukcji – zero lub więcej instrukcji, w klamrach „{}”

instrukcja

- deklaracja zmiennej
- wyrażenie zakończone średnikiem

przykładowy program

program, który robi „nic”

```
main(){}
```

- to jest najkrótszy program w „C” – zawiera tylko deklarację wymaganej funkcji `main` i jej minimalną (pustą) definicję
- odstępy (w tym znaki tabulacji i łamanie lini) jedynie oddzielają elemnty składni – służą czytelności i przejrzystości kodu

co z nim zrobić

zapisać w pliku

użyć dowolny edytor tekstowy i jakoś nazwać (np. `pierwszy.c`)

skompilować

```
cc pierwszy.c
```

wykonać

`a.out` lub prawdopodobnie `./a.out`

- jeśli chcemy nazwać program wynikowy należy użyć jednej z licznych opcji kompilatora:

  ```
  cc pierwszy.c -o pierwszy
  ```

- po koncówce nazwy `.c` kompilator rozpoznaje co robić z takim plikiem

→ pełna lista opcji kompilatora: `man gcc`

lepszy przykładowy program

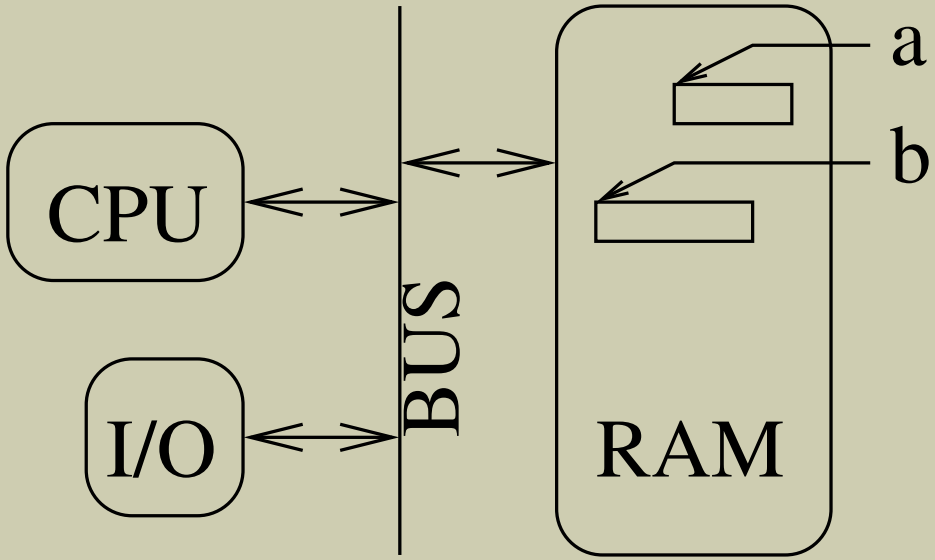
program, który już coś robi

```
#include <stdio.h>
/* program wypisujący coś */
int main(){
printf("Hallo World!\n"); // wypisz tekst
}
```

- znana już deklaracja funkcji `main()` (dodtakowo zadeklarowany typ zwracanej wartości)
- użyta instrukcja wywołania funkcji `printf()` ze swoim argumentem (zakończona średnikiem)
- użyta „makrodefinicja” `#include` powodująca włączenie w tekst programu tzw. pliku nagłówkowego (z deklaracjami różnych funkcji m. in. `printf()`)
- colwiek pomiędzy `/*` a `*/` oraz od `//` do końca lini to komentarz – kompilator traktuje go tak jak odstęp

co to jest zmienna

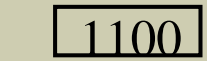
- nazwany obszar w pamięci zarezerwowany dla obiektu

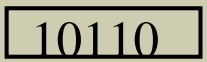


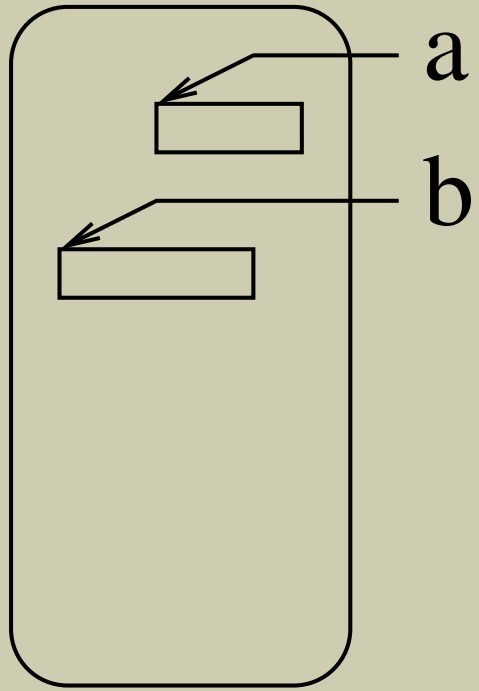
- jakiekolwiek dane umieścimy w takim obszarze będą tam dopóki nie umieścimy tam innych danych
- dane w takim miejscu pamięci umieszczamy instrukcją podstawienia

podstawienie

```
a = 12;
b = 22;
a = 6;
```

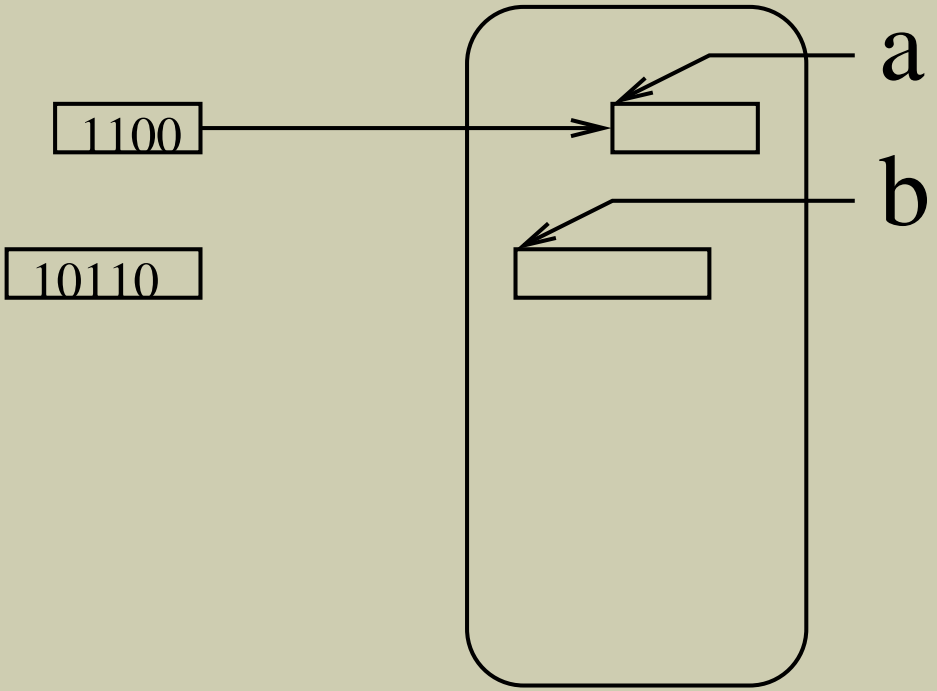






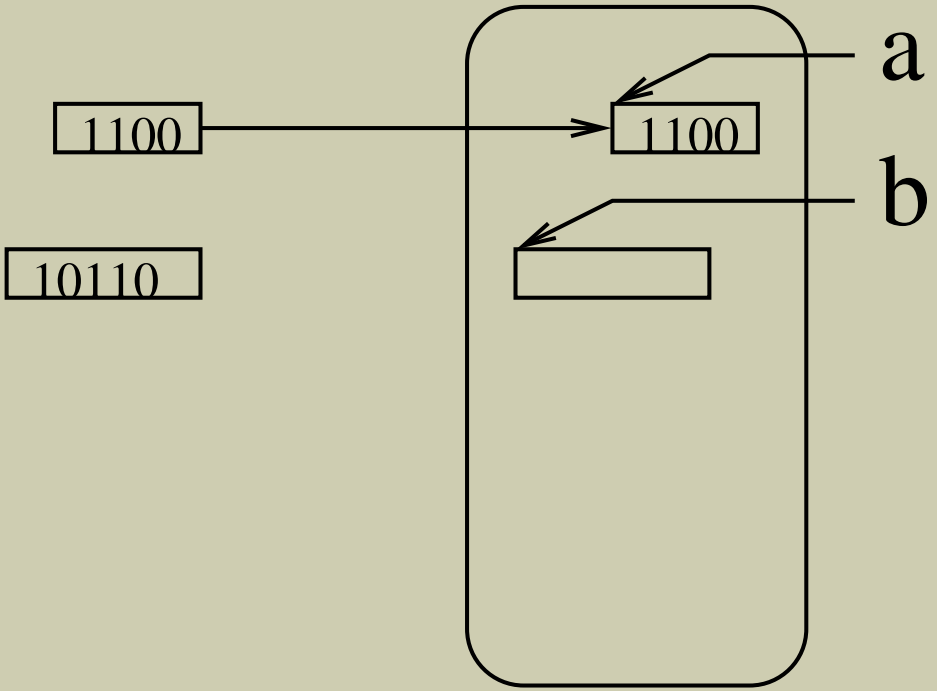
podstawienie

```
a = 12;
```



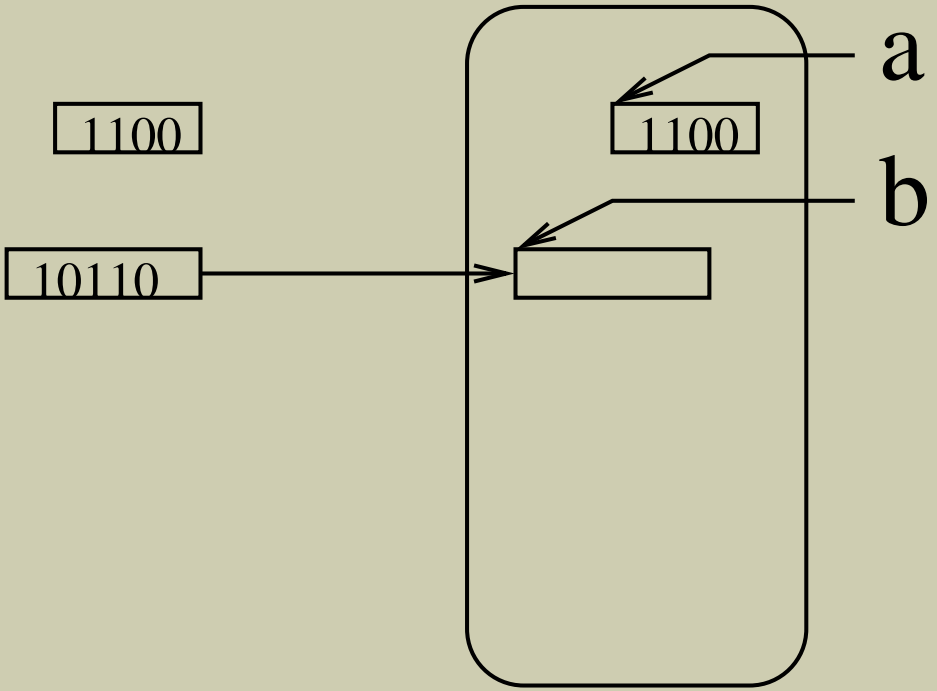
podstawienie

```
a = 12;
```



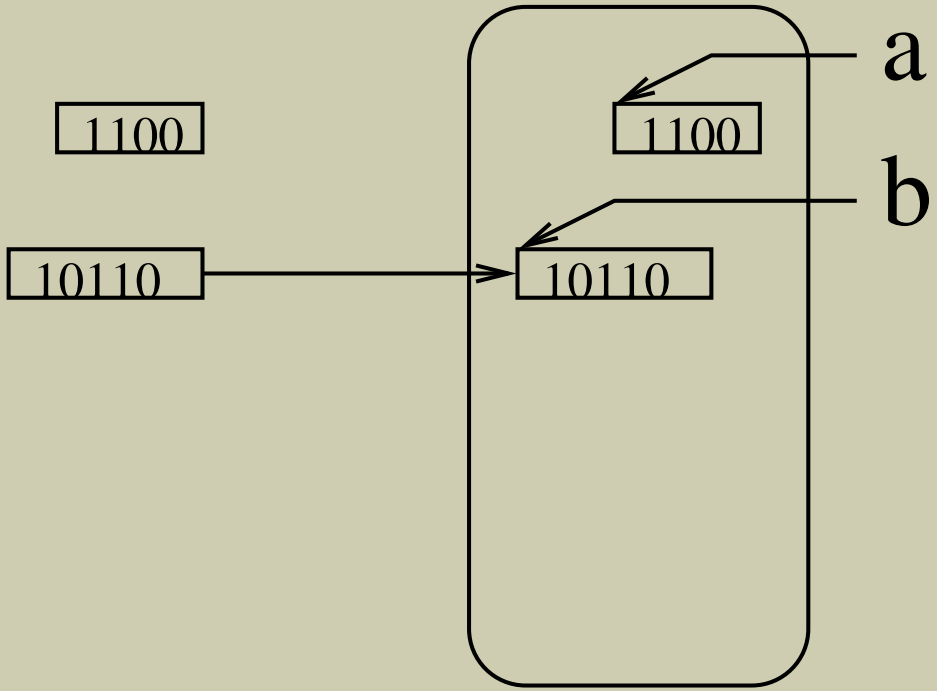
podstawienie

```
a = 12;
b = 22;
```



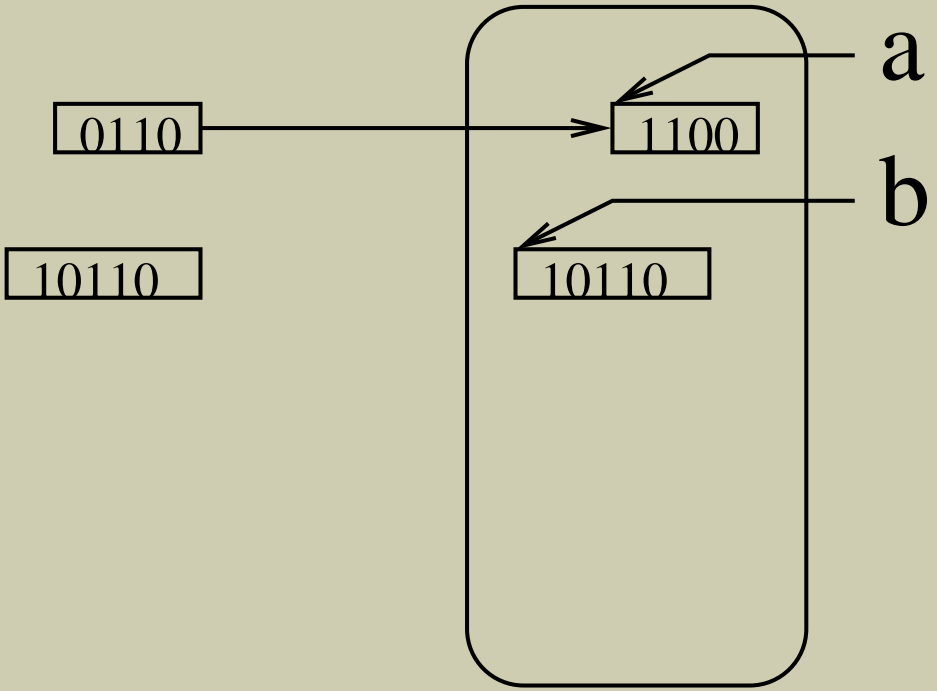
podstawienie

```
a = 12;
b = 22;
```



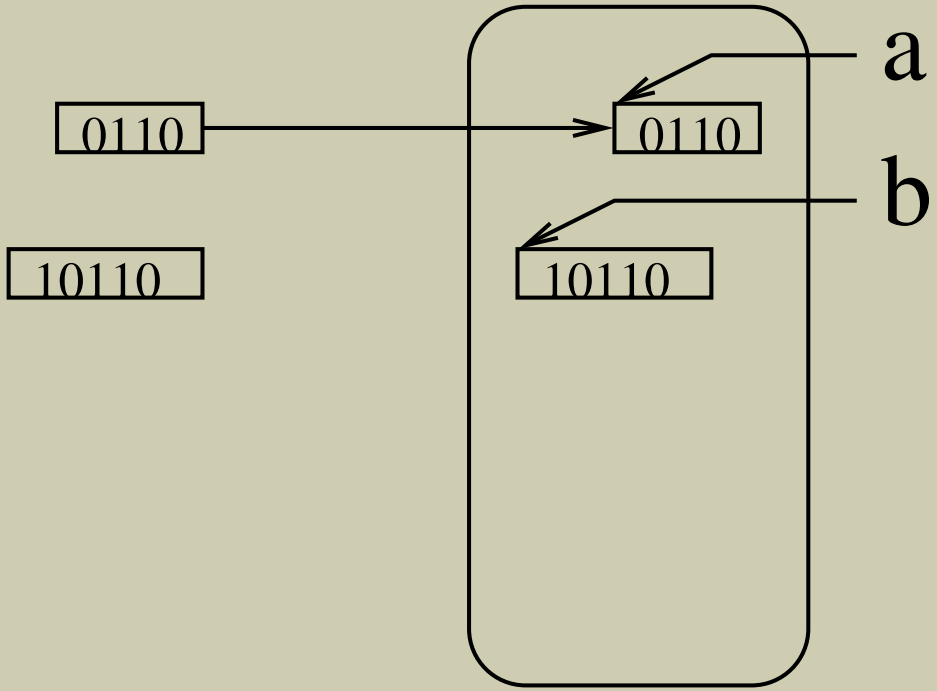
podstawienie

```
a = 12;
b = 22;
a = 6;
```



podstawienie

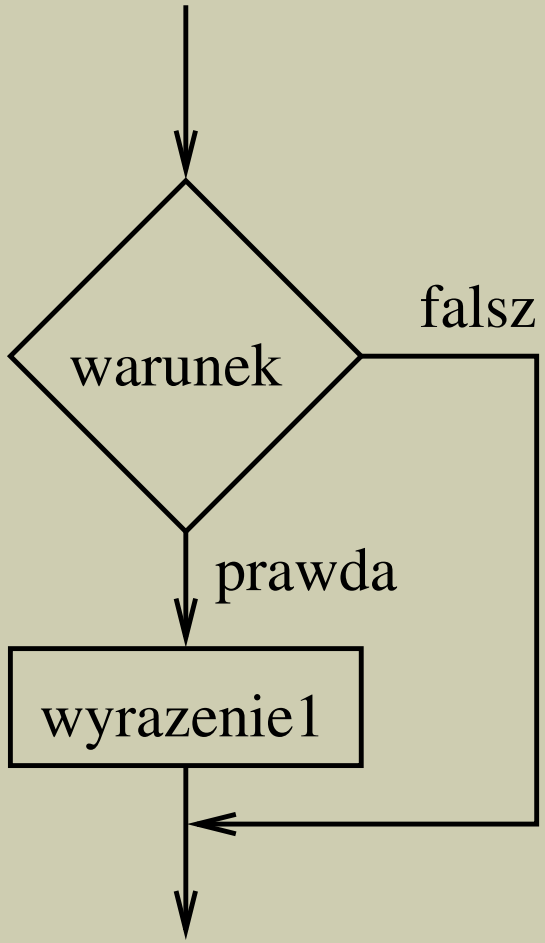
```
a = 12;
b = 22;
a = 6;
```



instrukcja warunkowa

`if ( <`*warunek*`> ) <`*wyrażenie1*`> ;`

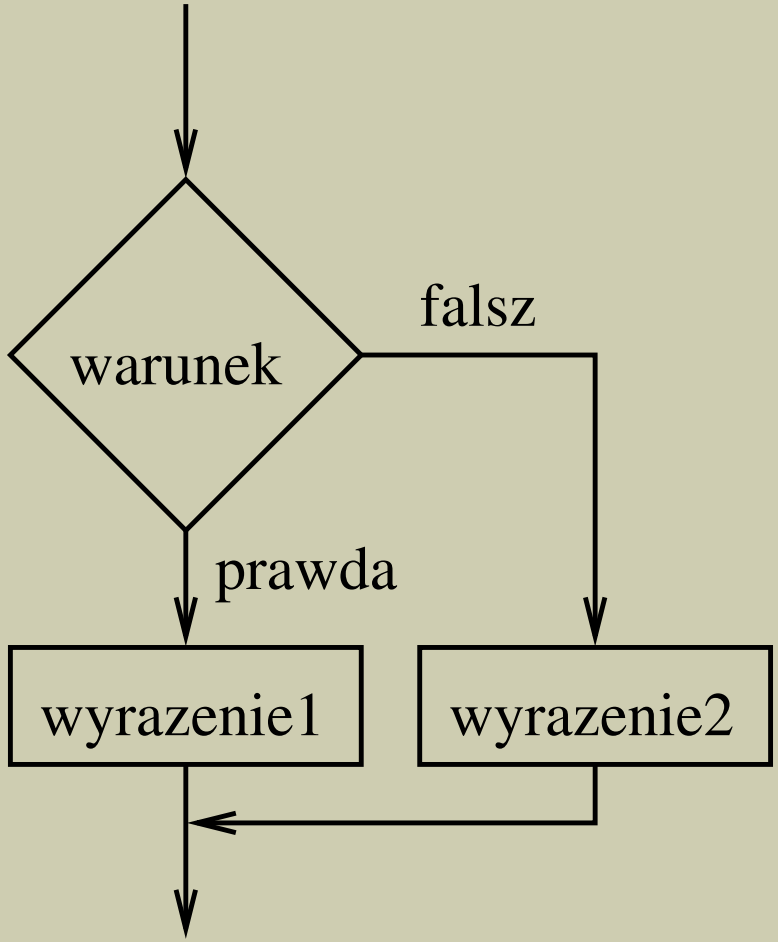
- *wyrażenie1* można pominąć
- *wyrażenie1* to zwykle blok instrukcji



instrukcja warunkowa

`if ( <`*warunek*`> ) <`*wyrażenie1*`> ;`
`else <`*wyrażenie2*`> ;`

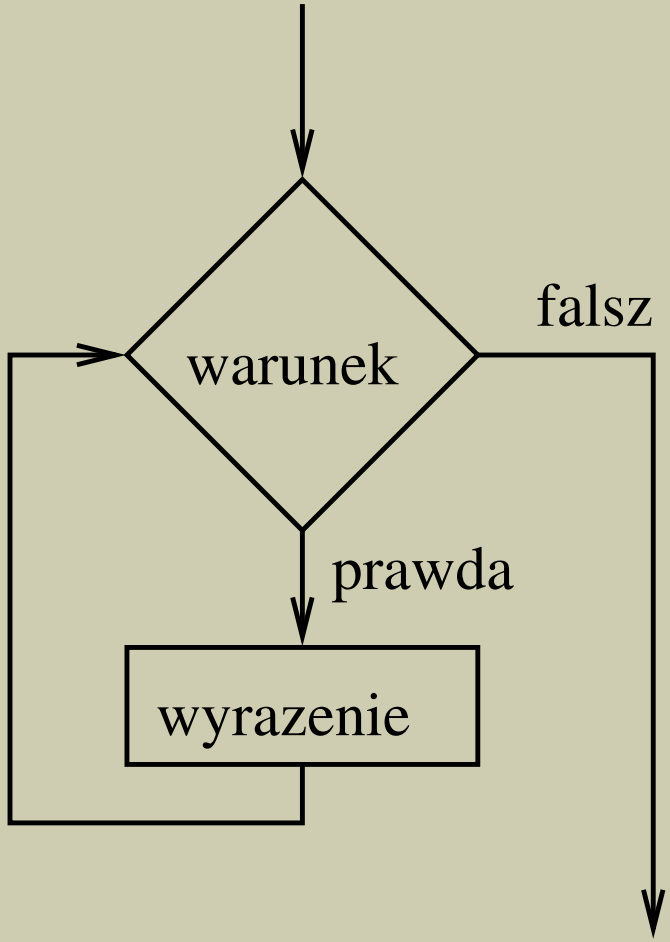
- oba wyrażenia można pominąć
- oba wyrażenia to zwykle bloki instrukcji



instrukcja powtarzania (pętla) „while”

```
while ( <warunek> )
  <wyrażenie> ;
```

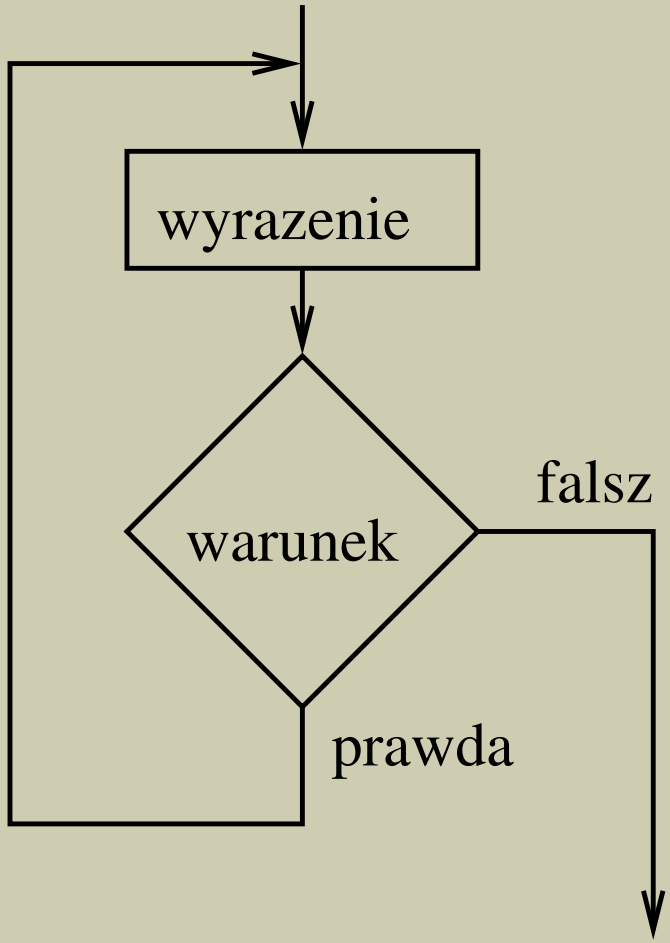
- *wyrażenie* można pominąć
- *wyrażenie* to zwykle blok instrukcji



pętla „do”

```
do
  <wyrażenie> ;
while( <warunek> );
```

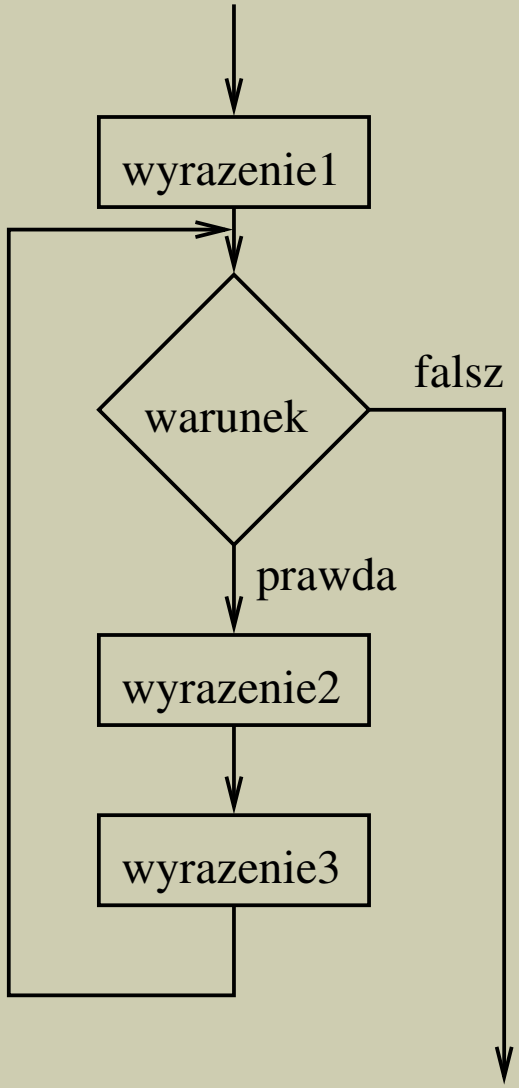
- *wyrażenie* można pominąć
- *wyrażenie* to zwykle blok instrukcji



pętla „for”

`for (`<*wyrażenie1*>` ;` <*warunek*>` ;` <*wyrazenie3*>` )`
  <*wyrażenie2*>` ;`

- oba wyrażenia w nagłówku, jak i *warunek* można pominąć – zostać muszą średniki
- *wyrażenie2* to zwykle blok instrukcji



uwagi do wszystkich pętli

- każdą pętlę można zastąpić inną – decyduje wygoda i przejrzystość zapisu
- najczęściej stosuje się pętlę `while` – gdy znamy warunek zakończenia/kontynuacji powtarzania sekwencji ciała pętli
- w zastosowaniach numerycznych najczęściej znamy z góry ilość powtórzeń – wtedy najwygodniejsza jest pętla `for`
  wyrażenie1 inicjalizuje licznik powtórzeń, *wyrażenie3* inkrementuje a *warunek* sprawdza (ogranicza) jego wartość
- pętla `do` zawsze wykonuje *wyrażenie* conajmniej raz
- pętle `while` i `for` mogą nie wykonać ciała pętli w ogóle
- należy zadbać, żeby warunek kontynuacji pętli kiedyś był fałszem

wyrażenia i warunki

wyrażenie

- kombinacja indentyfikatorów i operatorów

warunek

- kombinacja indentyfikatorów i operatorów

- wartość wyrażenia zwykle interpretujemy jako wartość numeryczną, wartość warunku jako logiczną (prawda, lub fałsz)
- „C” nie wyróżnia wartości logicznych – „prawda” to każda wartość różna od zera, wartość zero to „fałsz”

nazwy zmiennych (i nie tylko)

identyfikator

- ciąg liter lub cyfr zaczynający się od litery
  - → znak podkreślenia '_' to też litera
  - → małe i duże litery to różne litery
  - → odstęp (spacja) to nie jest litera

- niektóre identyfikatory są „zarezerowane” (tzw. słowa kluczowe) np. `while`, `for`, `if`, `int`, ...

typy wartości

- numeryczne
  - stałopozycyjne: `char`, `int`
  - i ich modyfikacje: `short`, `long`, `signed`, `unsigned`
  - zmiennopozycyjne: `float`, `double`
  - i ich modyfikacja: `long`
- napisy (tablice znakowe: `char *`, lub `char []`)
- złożone – struktury, unie, wskaźniki, tablice
- własne (definiowane w programie)

– różne inne „dodatki” `static`, `auto`, `register`, `extern`, `volatile`

→ określenie typu wartości (np. zmiennej) decyduje o tym, jak jest interpretowana zawartość komórek pamięci

przykłady wartości numerycznych (stałych)

- stałopozycyjne
  - dziesiętne: `0`, `-5`, `10`, `12ul`
  - znakowe: `'a'`, `'\n'`
  - ósemkowe: `072`, `02`
  - szesnastkowe: `0x3f`, `0X12`
- znmiennopozycyjne
  - `1.23`, `0.`, `.0123`, `1e+4`, `-2.E-2`, `.1e10`
- napisy
  - `"Ala ma kota"`, `"A"`, `"ai'bc\\54#ABC\"123"`

$\longrightarrow$ `0` to co innego niż `0.`, `.0` czy `0.0`, `'0'` to jeszcze coś innego

$\longrightarrow$ `1` to jeszcze bardziej coś innego niż `1.` czy `1e0`

$\longrightarrow$ `'A'` to co innego niż `"A"`

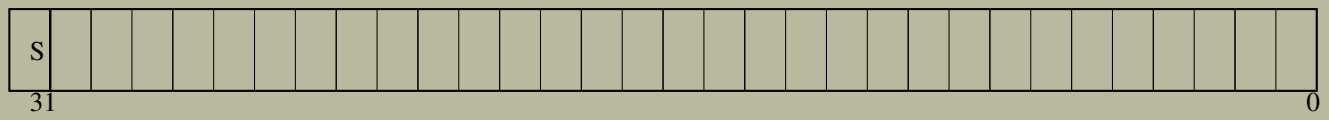
nieprawidłowe przykłady stałych

- stałopozycyjne
  - `1.0` – nie może być części dziesiętnej
  - `09` – '9' nie jest cyfrą ósemkową
  - `0x0h` – 'h' nie jest cyfrą szesnastkową
  - `'aa'` – dwa znaki to nie znak
  - `'ą'` – znaki spoza tablicy ASCII są zwykle kilkoma znakami
- zmiennopozycyjne
  - `0.1d5` – tylko 'e' lub 'E' może rozpoczynać mnożnik potęgowy
  - `1e5.1` – wykładnik musi być całkowity
- napisy
  - `"Ala ma "kota"` – już środkowy cudzysłów kończy napis

reprezentacja wartości numerycznych

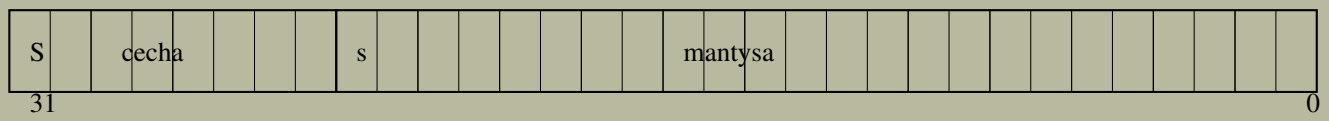
stałopozycyjne

- bit po bicie w reprezentacji dwójkowej z ew. (najstarszym) bitem znaku



zmiennoopozycyjne

- cecha (ze znakiem) i mantysa (też ze znakiem) – wartość numeryczna to $m \cdot 2^c$
- ilość bitów na cechę i mantysę definiuje kompilator (procesor)



$\rightarrow$ ułożenie bajtów (8 bitów) w całym słowie bywa odwrotne (lub jeszcze inne, bardziej zagmatwane kombinacje)

konsekwencje reprezentacji wartości numerycznej

- skończona ilość bitów na przechowywaną wartość
  - skończona ilość reprezentowanych wartości
  - istnienie największej i najmniejszej liczby w zbiorze wartości stałopozycynych
  - dodatkowo w zbiorze wartości zmiennopozycyjnych, istnienie liczby np. najmniejszej, niezerowej wartości dodatniej,

$\longrightarrow$ zbiór wartości stałopozycyjnych to **nie** zbiór wartości całkowitych

$\longrightarrow$ wynikiem np. dodania jedynki do **największej** wartości stałopozycyjnej jest **najmniejsza** wartość stałopozycyjna

$\longrightarrow$ zbiór wartości zmiennopozycyjnych to **nie** zbiór wartości rzeczywistych

$\longrightarrow$ zbiór wartości zmiennopozycyjnych jest bardzo „dziurawy” – im dalej od zera tym kolejne reprezentowane wartości są od siebie bardziej odległe

$\longrightarrow$ prawie wszystkie wartości rzeczywiste są reprezentowane niedokładnie (im większe tym bardziej)

czas na działanie

wypisz kolejne wartości od 1 do 10

```
#include <stdio.h>
int main(){
int a;              // deklaracja zmiennej stałopozycyjnej
a=1;                // nadanie wartości początkowej
while(a<=10){   // pętla wykonuje się aż a przekroczy 10
  printf("%d\n",a); // wypisz wartość zmiennej a
  a=a+1;              // zwiększ a o 1
  }
}
```

- a=a+1 to kluczowa instrukcja – do wartości zmiennej a jest dodana jedynka i wynik tej operacji jest zachowany w pamięci w miejscu przeznaczonym dla zmiennej a (podstawiony do zmiennej)
- wcięcia, łamanie lini, odstępy, komentarze służą tylko (i aż) przejrzystości i czytelności tekstu programu

to samo z użyciem pętli `for`

wypisz kolejne wartości od 1 do 10

```
#include <stdio.h>
int main(){
int a;              // deklaracja zmiennej stałopozycyjnej
for(a=1;a<=10;a++)  // pętla wykonująca się 10 razy
  printf("%d\n",a); // wypisz wartość zmiennej a
}
```

- zaczynamy od wartości a wynoszącej 1
- pętlę kontynuujemy póki wartość a nie przekroczy 10
- w pętli, po każdym wykonaniu ciała (`printf(...)`) zwiększamy a o jeden (wykorzystany operator inkrementacji ++)

funkcja `printf()`

- pierwszym argumentem jest tzw. format (napis – ciąg znaków zawarty pomiędzy cudzysłowami)
- w formacie, od znaku % to tzw. pole formatu – określa w jaki sposób interpretować wartość kolejnego argumentu
- tyle ile jest pól formatu tyle musi być kolejnych argumentów (po formacie) w wywołaniu funkcji `printf()`
- pole formatu musi być zgodne z typem wartości odpowiadającego argumentu – jeśli nie jest, efekty są nieprzewidywalne
- dla wypisania (dziesiętnie) wartości typu `int` używamy formatu 'd'
- dla wypisania wartości typu `double` używamy formatu 'lf', 'le' lub 'lg'
- dla wypisania znaku (typu `char`) używamy formatu 'c'
- każdy znak nie należący do pola formatu jest po prostu wypisywany
- pole formatu przewiduje różne sposoby formatowania wydruku np. `%10.4lf` wypisuje wartość `double` na 10 miejscach z 4 miejscami dziesiętnymi
- pełny opis np. w podręcznym *manualu*: `man 3 printf`

trochę bardziej skomplikowany przykład

wypisz kolejne wartości od 1 do 10 i ich kwadraty

```
#include <stdio.h>
int main(){
int a;              // deklaracja zmiennej stałopozycyjnej
for(a=1;a<=10;a++)  // pętla wykonująca się 10 razy
  printf("%d %d\n",a,a*a);// wypisz wartość a i jej kwadrat
}
```

- funkcja `printf()` wypisuje dwa argumenty – jednym jest wartość zmiennej, drugim wartość wyrażenia `a*a`
- oba są opisane takim samym polem formatu, oddzielone pojedynczym odstępem (zakończone znakiem konca lini `\n`)
- lepiej wygląda wydruk gdy format będzie np. `"%3d %3d\n"`

ćwiczenia

warto samemu się przekonać co się będzie działo ...

```
#include <stdio.h>
int main(){
int a=5;        // deklaracja z inicjalizacją
double b=3.2;
char c='8';
printf("%d %lf %c\n",a,b,c);// poprawne wypisanie wartości
printf("%d\n",c);        // też poprawne wypisanie wartości
printf("%lf %d\n",a,b);     // niezgodne typy zmiennych
                            // i opisu pola
}
```

- spróbować różnych postaci opisu pola formatu – %3d, %05d, %-5d, %5.0lf, %le, ...

bardziej skomplikowane ćwiczenie

wypisz tabliczkę mnożenia

	1	2	3	...	10
1	1	2	3	...	10
2	2	4	6	...	20
3	3	6	9	...	30
⋮	⋮	⋮	⋮		⋮
10	10	20	30	...	100

- do poziomej lini użyj znaku '−', a do pionowej '|'
- zastosuj odpowiedni opis pola formatu, żeby kolumny były wyrównane
- pracę ma za nas wykonać komputer, więc użyj pętli (jednej wewnątrz drugiej)

wypisz tabliczkę mnożenia

```
#include <stdio.h>
int main() {
int x,y;   // ,,współrzędne'' tabliczki

printf("     |");  // wypisz linię nagłówka
for(x=1;x<=10;x++) printf("%4d",x);
printf("\n  "); // zakończ linię i dodaj dwa odstępy

for(x=0;x<44;x++) printf("-"); // linia podkreślenia
printf("\n");                  // i jej koniec

for(y=1;y<=10;y++) {             // iteracja po wierszach
  printf("%4d |",y);
  for(x=1;x<=10;x++) printf("%4d",y*x); // i po kolumnach
  printf("\n");      // koniec lini po wierszu
  }
}
```

czytanie znaków

funkcja `getchar()`

```
int getchar()
```

- funkcja czyta jeden znak ze strumienia wejściowego (klawiatury)
- zwraca wartość stałopozycyjną odpowiadającą znakowi (kod znaku)
- w przypadku wystąpienia błędu – zwraca specjalną wartość `EOF` (zdefiniowaną w pliku `stdio.h`)

ćwiczenie – przeczytaj znak i wypisz go wraz z jego kodem

```
#include <stdio.h>
int main(){
int ch;   // deklaracja zmiennej dla przeczytanego znaku
ch=getchar();  // przeczytanie znaku
printf("%c %d\n",ch,ch); // wypisanie tej samej wartości
}                        // dwa razy ale w różny sposób
```

ćwiczenie – kopiowanie znaków z wejścia na wyjście

- wydaje się proste – trzeba w pętli przeczytać znak i go wypisać
- problem – kiedy zakończyć pętlę?
  - naturalne rozwiązanie – wykorzystać informację o braku danych w strumieniu wejściowym
  - wprowadzenie z klawiatury znaku `Ctrl-D` oznacza koniec danych (pliku)
  - funkcja `getchar()` zwraca wtedy wartość `EOF`
- jakiej pętli użyć?
  - naturalne wydaje się użycie pętli `do`

z użyciem pętli do

```
#include <stdio.h>
int main(){
int ch;          // deklaracja zmiennej dla znaku
do
  ch=getchar(); // przeczytanie znaku
  printf("%c",ch); // wypisanie znaku
while(ch!=EOF); // warunek kontynuacji pętli
}
```

drobne problemy

- wypisywany jest również znak końca pliku
- trzeba wypisywanie znaku obłożyć warunkiem
- efekt jest mało „elegancki”
- przy okazji – pojedynczy znak można wypisać inną (prostszą) funkcją putchar()

z użyciem pętli do

```
#include <stdio.h>
int main(){
int ch;          // deklaracja zmiennej dla znaku
do{
  ch=getchar(); // przeczytanie znaku
  if(ch!=EOF) putchar(ch); // wypisanie znaku
  }
while(ch!=EOF); // warunek kontynuacji pętli
}
```

a może jednak użyć pętlę `while`?

- żeby sprawdzić warunek trzeba raz przeczytać znak przed pętlą
- w pętli też trzeba przeczytać – po wypisaniu
- `getchar()` występuje dwukrotnie – też mało elegancko

z użyciem pętli `while`

```
#include <stdio.h>
int main(){
int ch;          // deklaracja zmiennej dla znaku
ch=getchar();    // przeczytanie pierwszego znaku
while(ch!=EOF){
  putchar(ch);  // wypisanie znaku
  ch=getchar(); // przeczytanie kolejnego znaku
  }
}
```

typowo robi się to tak

lepiej i z użyciem pętli `while`

```
#include <stdio.h>
int main(){
int ch;          // deklaracja zmiennej dla znaku
while((ch=getchar())!=EOF) // przeczytanie znaku i test na EOF
  putchar(ch);  // wypisanie znaku
}
```

- czytanie znaku jest w nagłówku pętli – wykonywane za każdym razem
- jeśli warunek nie jest prawdziwy – ciało pętli nie jest wykonane (nie wypisuje się wartości `EOF`
- konieczne są nawiasy, żeby najpierw wykonać podstawienie

$\longrightarrow$ wartością podstawienia jest podstawiana wartość

- ciało pętli stanowi jedna instrukcja – nie ma potrzeby (ale można) zawierać jej w klamry

ćwiczenie

- zlicz ile razy w strumieniu wejściowym wystąpiła litera 'a'
  - czytaj znak po znaku
  - jeśli przeczytany znak to 'a' to zwiększ licznik o 1

zlicz literki 'a'

```
#include <stdio.h>
int main(){
int ch; // zmienna dla czytanego znaku
int licznik=0;  // zmienna dla licznika
while((ch=getchar())!=EOF) // przeczytanie znaku i test na EOF
  if(ch=='a') licznik++;   // jeśli znak to 'a' - zwiększ licznik
printf("ilość a: %d\n",licznik);  // wypisz wynik
}
```

podobne ćwiczenie

- zlicz ile jest lini w strumieniu wejściowym
- co to jest linia?

→ ciąg liter zakończony końcem lini

- jak „wygląda” koniec lini?

→ '\n'

zlicz linie

```
#include <stdio.h>
int main(){
int ch; // zmienna dla czytanego znaku
int licznik=0;  // zmienna dla licznika
while((ch=getchar())!=EOF) // przeczytanie znaku i test na EOF
  if(ch=='\n') licznik++;  // jeśli koniec lini - zwiększ licznik
printf("ilość lini: %d\n",licznik);  // wypisz wynik
}
```

raz jeszcze podobne ćwiczenie

- zlicz ile jest lini i wszystkich znaków w strumieniu wejściowym

→ obie wartości musimy wyznaczać w tej samej pętli

zlicz linie i znaki

```
#include <stdio.h>
int main(){
int ch; // zmienna dla czytanego znaku
int lini=0, znakow=0;  // zmienne dla licznika lini i znaków
while((ch=getchar())!=EOF){ // przeczytanie znaku i test na EOF
  if(ch=='\n') lini++;  // jeśli koniec lini - zwiększ licznik
  znakow++;  // bezwarunkowo zwiększ licznik znaków
  }
printf("ilość lini: %d\nznaków: %d\n",lini,znaków);// wypisz wynik
}
```

- mamy dwie instrukcje w ciele pętli – potrzebne klamry
- w formacie są dwa pola na oba wyniki, po znaku nowej lini nie chcemy odstępu, więc go nie ma w formcie

podobne, ale jednak inne ćwiczenie

- zlicz ile razy w strumieniu wejściowym wystąpiły poszczególne znaki i wypisz te, które wystąpiły (i ile razy)

$\longrightarrow$ potrzebujemy wiele liczników – dla każdego znaku jeden

$\longrightarrow$ struktura danych mieszcząca wiele wartości tego samego typu to tablica

- tablica ma okeślony typ i rozmiar
- do elementu tablicy odwołujemy się podając identyfikator tablicy i numer indeksu (w nawiasach kwadratowych)
- elementy tablicy są numerowane od '0'
- np. piąty element tablicy `tab` to `tab[4]`
- ostatni element 100 elementowej tablicy to `tab[99]`

$\longrightarrow$ kompilator nie sprawdza odwołań do nieistniejących elementów tablic (zbyt duży indeks lub ujemny), takie (błędne) odwołania prowadzą do nieprzewidywalnego działania programu

podobne, ale jednak inne ćwiczenie

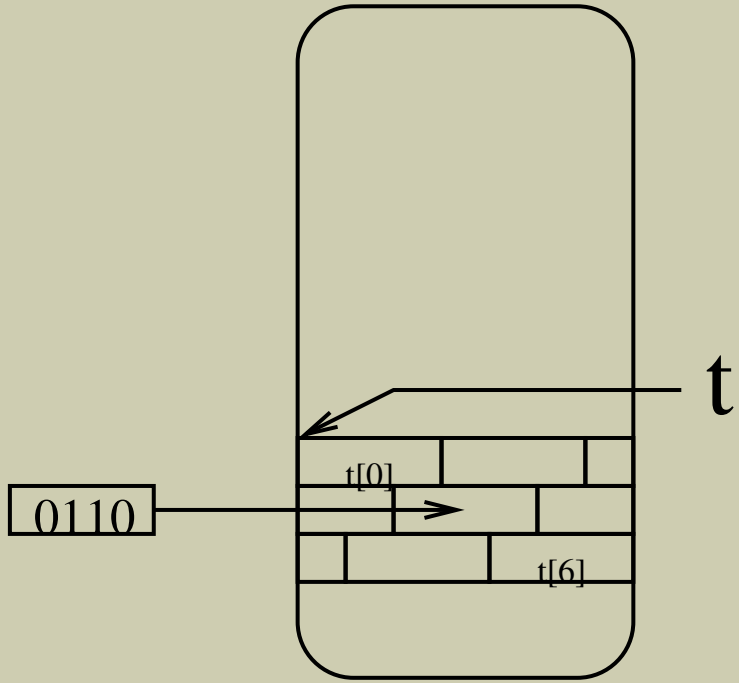
zlicz znaki

```
#include <stdio.h>
int main(){
int ch,i; // zmienna dla znaku i pomocnicza
int licznik[256]; // tablica liczników
for(i=0;i<256;i++) licznik[i]=0; // zeruj liczniki
while((ch=getchar())!=EOF) // przeczytanie znaku i test na EOF
  licznik[ch]++;  // zwiększ licznik znaku o wartości ch
for(i=32;i<127;i++)   // tylko znaki drukowalne
  if(licznik[i]>0)    // jeśli znak wystąpił
    printf("%c: %d\n",i,licznik[i]); // wypisz wynik
}
```

- znak jest identyfikowany przez jego wartość (kod)
- wypisujemy sam znak i ilość jego wystąpień
- żeby uniknąć niespodzianek, wypisujemy tylko tzw. drukowalne znaki (o kodach od 32 do 126 włącznie)

tablica

```
int t[7]; // deklaracja
t[3]=6; // podstawienie
```

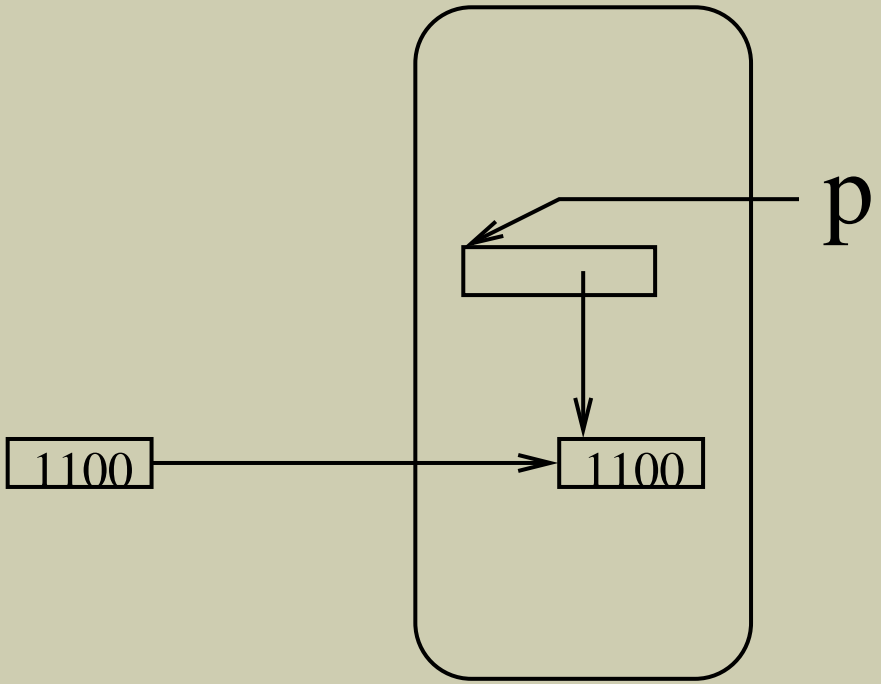


tablice i wskaźniki

- identyfikator tablicy (bez indeksu) to numer (adres) miejsca w pamięci, gdzie umieszczane są wartości elementów
- adres miejsca w pamięci nazywamy wskaźnikiem
- wskaźnik wskazuje miejsce zmiennej określonego typu – sam ma typ „wskaźnik do . . . ”
- do odwołania się do wskazywanego miejsca używa się operatora „wyłuskania” – praktycznie to samo co odwołanie się do pierwszego elementu tablicy: `*p` daje tą samą wartość co `t[0]`
- adres np. zmiennej uzyskujemy operatorem referencji `&`
- nazwa tablicy to stała wskaźnikowa (typu wskaźnik do . . .)
- operacja pobrania wartośći elementu jest równoważna pobraniu wartości wskazywanej przez nazwę tablicy, zwiększony o wartość indeksu: `t[i]` ma tą samą wartość co `*(t+i)`

podstawienie pod wskazywane miejsce w pamięci

```
int *p; // deklaracja
*p = 12; // lub p[0]=12;
```



podstawienie pod wskaźnik

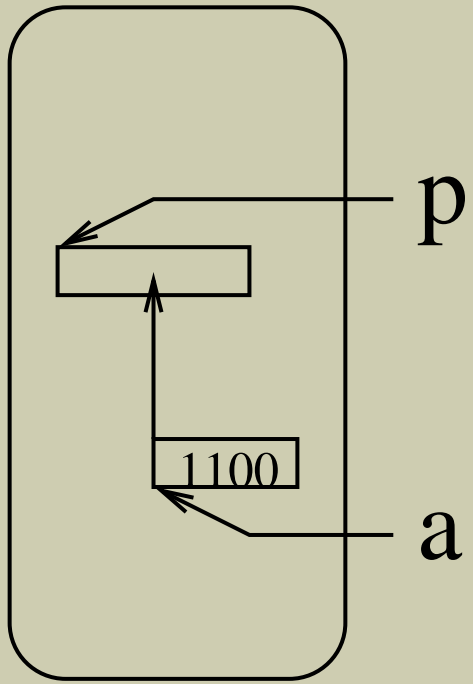
```
int *p, a;
p = &a;
```

- lub np.:

```
int t[10];
int *p;
p = t; // t to też wskaźnik
```

- ale już nie np.:

```
int t[10], *p;
t = p; // t to stały wskaźnik
```



tablice znaków – napisy

- napis (np. `"Ala ma kota"`) jest tablicą znaków (elementów typu `char`)
- ilość elementów takiej tablicy to ilość znaków **plus jeden**
- ten dodatkowy znak to tzw. znak końca napisu `'\0'` o wartości numerycznej (kodzie) rownym zero
- wykorzystują to liczne funkcje biblioteczne operujące na napisach
- tablicę znaków można inicjalizować stałym napisem podczas deklaracji (można pominąć rozmiar – wyliczy go kompilator):
  `char text[]="Ala ma kota";`
- przy okazji – inne tablice też można inicjalizować wraz z deklaracją:
  `int tab[]={1,2,3};`

formatowane czytanie np. liczb

funkcja `scanf()`

`int scanf(<format>, <argumenty>)`

- *format* jest podobny do formatu dla funkcji `printf()`
- *argumenty* muszą wskazywać miejsce w pamięci, gdzie funkcja `scanf()` ma wpisać przeczytaną wartość – zwykle wskaźnik do zmiennej
- typ wskazywanej zmiennej musi być zgodny z opisem pola formatu
- do pola formatu dopasowywany jest najdłuższy możliwy ciąg znaków „pasujący” do pola formatu, pierwszy niepasujący znak pozostaje do dopasowywania do kolejnego pola
- `scanf()` zwraca ilość poprawnie przeczytanych wartości
- np. przeczytanie jednej liczby zmiennopozycyjnej:
  `double x; scanf("%lf",&x);`

ćwiczenie

- przeczytaj liczbę i sprawdź, czy jest liczbą pierwszą
- sprawdzamy kolejno podzielniki, jeśli przez któryś się dzieli, to podana wartość nie jest liczbą pierwszą

sprawdź, czy podano liczbę pierwszą

```
#include <stdio.h>
int main(){
int x,i;  // zmienna dla przeczytanej liczby i pomocnicza
int p=0;  // 0 - oznacza,że mamy liczbę pierwszą, 1 - złożoną
scanf("%d",&x);   // przeczytaj liczbę całkowitą
for(i=2;i*i<=x;i++) // wystarczy sprawdzać do sqrt(x)
  if(x%i==0) p=1; // sprawdź podzielność przez i
if(p==0) printf("%d jest pierwsze\n",x);
else printf("%d nie jest pierwsze\n",x);
}
```

- % to operator wyliczania reszty z dzielenia

kilka modyfikacji

- wystarczy sprawdzać, do znalezienia pierwszego podzielnika
- sprawdzanie podzielności można teraz uprościć (choć to trochę mniej czytelne)
- warunek `p==0` można zastąpić `!p` lub samą wartością `p` i zamienić instrukcje za `if` i `else`

sprawdź, czy podano liczbę pierwszą

```
#include <stdio.h>
int main(){
int x,i;  // zmienna dla przeczytanej liczby i pomocnicza
int p=0;  // 0 - oznacza,że mamy liczbę pierwszą, 1 - złożoną
scanf("%d",&x);   // przeczytaj liczbę całkowitą
for(i=2;(i*i<=x)||p;i++) // sprawdź do sqrt(x) lub p==1
  p=(x%i==0);     // lub nawet: p=!(x%i);
if(p) printf("%d nie jest pierwsze\n",x);
else printf("%d jest pierwsze\n",x);
}
```

operatory języka C

operator	łączność
`() [] -> .`	L
`! ~ ++ -- - (`*typ*`) * & sizeof`	P
`* / %`	L
`+ -`	L
`<< >>`	L
`< <= > >=`	L
`== !=`	L
`&`	L
`^`	L
`|`	L
`&&`	L
`||`	L
`?:`	P
`= += -= ...`	P
`,`	L

- priorytet operatorów maleje z góry na dół tabeli
- to są wszystkie operatory języka „C”, nie wszystkie będą użyte w tym kursie

krótki opis niektórych operatorów

`()`	operator wywołania funkcji np. `funkcja(`*arg1, arg2*`)`
`[]`	operator pobrania elementu tablicy np. `tab[6]`
`!`	negacja logiczna – `!0` daje `1`, `!`*niezero* daje `0`
`++`	inkrementacja zmiennej np. `x++` lub `++x`
`--`	dekrementacja zmiennej np. `x--` lub `--x`
`-`	negacja arytmetyczna (zmiana znaku wartości)
(*typ*)	operator rzutowania typu (zmiana typu wartości)
`* &`	operacje na wskaźnikach
`* / %`	mnożenie, dzielenie, reszta z dzielenia
`+ -`	dodawanie, odejmowanie
`< <=`	porównaie (mniejszość) silna i słaba
`> >=`	porównaie (większość) silna i słaba
`== !=`	równość i nierówność
`&& ||`	logiczna koniunkcja i alternatywa
`?:`	trójargumentowy operator wyboru: `a=b?c:d;` działa jak: `if(b) a=c; else a=d;`
`=`	podstawienie (coś bardzo innego niż porównanie)
`+=`	dodanie prawej strony do zmiennej np. `a+=5` to (prawie) to samo co `a=a+5`

uwagi ogólne do operatorów i ich własności

- inny priorytet wykonywania operatorów można wymusić obejmując nawiasami ( ) operator z jego argumentami (argumentem)
- niekiedy warto dodać nawiasy dla przejrzystości
- łączność „L” ozancza, że z operatorów z równym priorytetem najpierw jest wykonywany ten z lewej strony („P” – ten z prawej)
- dla niektórych operatorów (np. porównania) łączność oznacza zupełnie co innego niż możnaby się spodziewać – należy unikać takich konstrukcji (jak np. `a<b<c`)

podsumowanie

- podane dotąd informacje w zupełności wystarczają do napisania prostego programu przetwarzającego wartości numeryczne
- to czego brakuje, to znajomość dostępnych funkcji bibliotecznych
- opisy funkcji bibliotecznych można znaleźć w trzeciej sekcji *manuala* (`man 3 <`*nazwa*`>`)