

# Programowanie niskopoziomowe

Prowadzący:  
Piotr Kisała

LABORATORIUM 4

# Przegląd tematów

- ◆ Tablice
- ◆ Procedury
  - Zmienne lokalne
  - Zmienne globalne
  - Parametry procedury
    - ◆ przekazywane przez wartość
    - ◆ przekazywane przez adres
- ◆ Funkcje
- ◆ Złożenie instrukcji
- ◆ Zarządzanie dużymi projektami

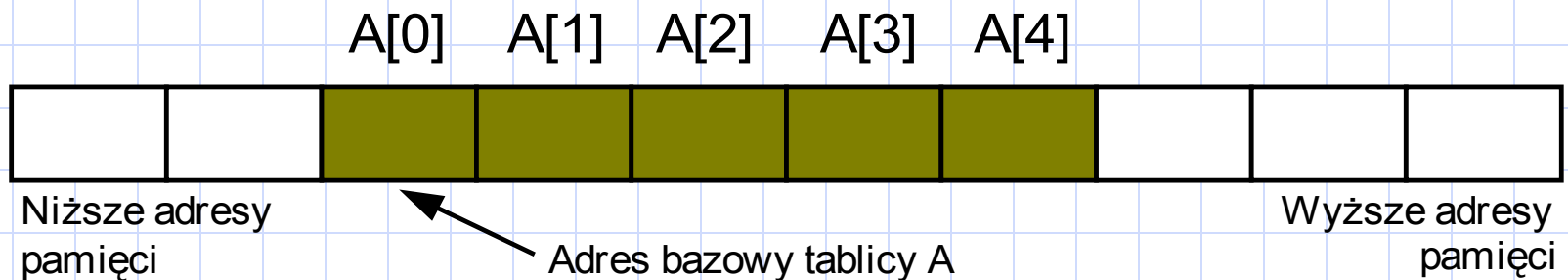
# Tablice

- ◆ Tablica jest agregatem, którego wszystkie składowe (elementy) mają ten sam typ.
- ◆ Wyboru elementu tablicy dokonuje się określając indeks elementu będący liczbą całkowitą.
- ◆ Zawsze kiedy odnośnie tablicy stosowany jest operator indeksowania, działanie operatora powoduje wybranie z tablicy elementu o zadanym indeksie, np. odwołanie w postaci  $A[i]$  powoduje wybranie z tablicy  $A$   $i$ -tego elementu.

# Adres bazowy tablicy

- ◆ Jest to adres, pod którym znajduje się w pamięci pierwszy element tablicy. Jest to zawsze najmniejszy co do wartości adres w tablicy.
- ◆ Drugi z elementów tablicy sąsiaduje w pamięci z elementem pierwszym, trzeci z drugim i tak dalej.
- ◆ Tablice indeksowane są od zera (chyba ,że w tekście wprost określony zostanie inny sposób indeksowania).

# 5-elementowa tablica w pamięci



- Do realizacji odwołań do poszczególnych elementów tablicy potrzebna jest funkcja, która indeks tablicy będzie tłumaczyła na adres indeksowanego elementu. W przypadku tablicy jednowymiarowej (wektor) funkcja wygląda następująco:

$$\text{adres-elementu} = \text{adres-bazowy} + ((\text{indeks} - \text{indeks-początkowy}) * \text{rozmiar-elementu})$$

- Wartość indeksu początkowego określa wartość indeksu pierwszego elementu tablicy, jeśli tablica indeksowana jest od zera, można tę wartość usunąć z funkcji.

# Tablice w assemblerach

- ◆ Przed rozpoczęciem korzystania z tablicy należy zarezerwować dla jej elementów pewien obszar pamięci. Aby przydzielić do tablicy pamięć dla  $n$  elementów, należy w jednej z sekcji deklaracji zmiennych umieścić następującą deklarację:

**NazwaTablicy: TypElementu[n];**

- ◆ Aby odwołać się do adresu bazowego tak zadeklarowanej tablicy, wystarczy odwołać się do zmiennej NazwaTablicy.
- ◆ Przyrostek  $[n]$  instruuje kompilator, że przydzielona do tablicy pamięć powinna pomieścić  $n$  obiektów typu TypElementu.

# Przykłady deklaracji tablic

static

```
znaki: char[128];
```

```
//tablica znaków o indeksach od 0 do 127
```

```
bajty: byte[10];
```

```
//tablica bajtów o indeksach od 0 do 9
```

```
słowa: dword[4];
```

```
//tablica podwójnych słów o indeksach od 0 do 3
```

# Tablice inicjalizowane

- ◆ Przy deklaracji tablicy można określić wyrażenie inicjalizacji, podając jawnie wartości wszystkich elementów tablicy:

```
tablica1:    real32[7] := [ 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,  
1.0 ];
```

```
tablica2:    int32[8]      := [ 1, 1, 1, 1, 1, 1, 1, 1 ];
```

- ◆ Powyższe definicje wprowadzają do programu tablice siedmio- i ośmioelementową zainicjalizowane statycznie. Wszystkie elementy pierwszej tablicy zawierają wartość 1.0, elementy tablicy drugiej zawierają wyłącznie jedynki. Liczba stałych występujących w wyrażeniu inicjalizującym musi dokładnie odpowiadać liczbie elementów tablicy.



# Tablice inicjalizowane

- Można również statycznie zainicjalizować elementy tablicy różnymi wartościami:

```
tablica1:    real32[7] := [ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0,  
7.0 ];
```

```
tablica2:    int32[8]      := [ 1, 2, 3, 4, 5, 6, 7, 8 ];
```

- W przypadku inicjalizacji tablicy tą samą wartością, przy dużej liczbie elementów tablicy, proces inicjalizacji można skrócić:

```
duzatablica:    int32[1000] := 1000 dup [1];
```

- Za pośrednictwem operatora *dup* można nawet inicjalizować tablice seriami wartości:

```
szesnastka:    int32[16] := 4 dup [1, 2, 3, 4];
```

# Odwołania do elementów tablicy jednowymiarowej

- ◆ Aby odwołać się do elementu tablicy jednowymiarowej indeksowanej od zera, należy skorzystać z następującego wzoru:

$\text{adres-elementu} = \text{adres-bazowy} + \text{indeks-elementu} * \text{rozmiar-elementu}$

- ◆ W miejsce adresu bazowego wystarczy określić nazwę zmiennej tablicowej w asemblerach z nazwą zmiennej tablicowej kojarzony jest adres pierwszego elementu tablicy).
- ◆ Aby odwołać się do elementu w zadeklarowanej wcześniej tablicy „szesnastka”, należałoby skorzystać ze wzoru:  
 $\text{adres-elementu} = \text{szesnastka} + \text{indeks} * 4$

Jeżeli mowa o efektywności kodu – procesory z rodziny 80x86 udostępniają tryb adresowania indeksowego skalowanego, pozwalającego automatycznie mnożyć wyznaczony indeks przez jeden, dwa, cztery bądź osiem.

W naszym przykładzie przesunięcie elementu wyznaczone jest iloczynem indeksu i liczby cztery, więc kod odwołania `EAX := szesnastka[index]` do elementu tablicy można zapisać:

```
mov( index, ebx);  
mov(szesnastka[ebx*4], eax);
```

# Tablice wielowymiarowe

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Pamięć

15	A[0,15]
14	A[0,14]
13	A[0,13]
12	A[0,12]
11	A[0,11]
10	A[0,10]
9	A[0,9]
8	A[0,8]
7	A[0,7]
6	A[0,6]
5	A[0,5]
4	A[0,4]
3	A[0,3]
2	A[0,2]
1	A[0,1]
0	A[0,0]

W układzie wielowierszowym pod kolejnymi adresami pamięci odwzorowywane są elementy tablicy w kolejności, w jakiej występują w kolejnych wierszach.

# Pobieranie adresu obiektu

- ◆ Do pobierania adresu obiektu statycznego służy operand (&). Niestety, nie może być on wykorzystany do zmiennych automatycznych (deklarowanych w sekcji var). Operator nie nadaje się też do pobrania adresu odwołania do pamięci realizowanego w trybie indeksowym albo indeksowym skalowanym (nawet jeśli częścią wyrażenia adresowego jest zmienna statyczna). Operator pobrania adresu (&) nadaje się więc wyłącznie do określania adresów prostych obiektów statycznych.
- ◆ W zestawie instrukcji procesorów z rodziny 80x86 przewidziana jest instrukcja załadowania adresu efektywnego lea (load effective adres)  
**lea(rejestr<sub>32</sub>, operand-pamięciowy);**
- ◆ Pierwszym z operandów musi być 32-bitowy rejestr. Operand drugi może być dowolnym dozwolonym odwołaniem do pamięci przy użyciu dowolnego z dostępnych trybów adresowania. Wykonanie instrukcji powoduje załadowanie określonego rejestru obliczonym adresem efektywnym. Instrukcja nie wpływa przy tym w żaden sposób na wartość operandu znajdującego się pod obliczonym adresem.
- ◆ Po załadowaniu adresu efektywnego do 32-bitowego rejestru ogólnego przeznaczenia można wykorzystać adresowanie pośrednie przez rejestr, adresowanie indeksowe, indeksowe skalowane.

# Procedury

- ◆ Procedura to zestaw instrukcji realizujących pewne obliczenie, bądź inną funkcję programu, jak wyprowadzanie danych na zewnątrz, czy odczyt pojedynczego znaku z wejścia.
- ◆ Procedura to zestaw reguł, którego zastosowanie powinno doprowadzić do uzyskania pewnego wyniku czy efektu.
- ◆ W pewnym miejscu kodu następuje wywołanie procedury, podejmowane jest wykonanie kodu procedury, po czym sterowanie jest przekazywane z powrotem z doku procedury do programu.
- ◆ Procedurę wywołuje się za pomocą instrukcji maszynowej **call**.
- ◆ W kodzie procedury powrót do miejsca wywołania realizuje się za pośrednictwem instrukcji maszynowej **ret**.
- ◆ Na przykład procedura biblioteczna `stdout.newln` wywoływana jest w języku assemblerowym procesora 80x86 następująco:

**call stdout.newln;**

# Procedury - składnia

Najprostsza deklaracja procedury przyjmuje postać:

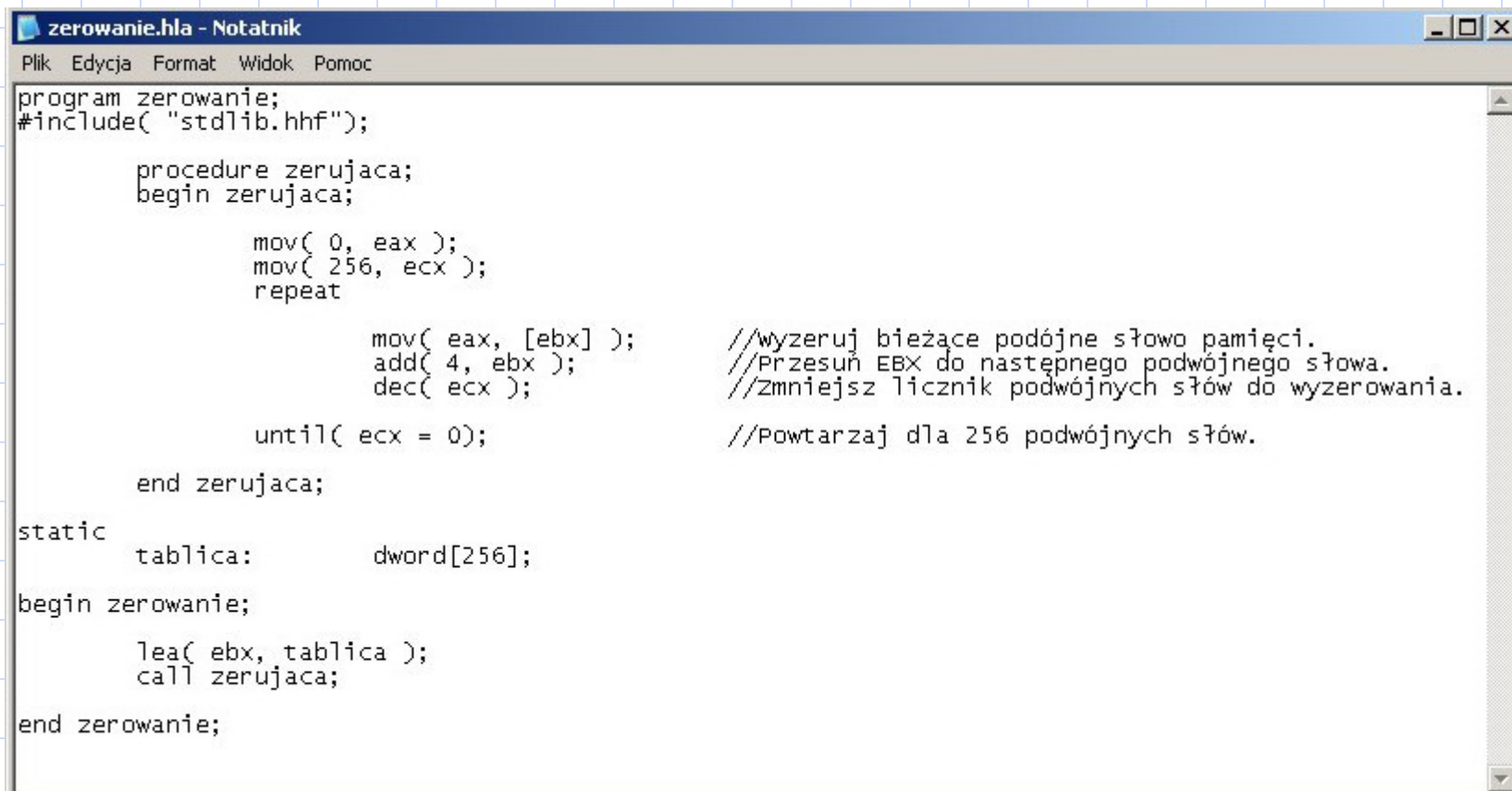
```
procedure nazwa-procedury;  
    //Deklaracje lokalne względem procedury;  
begin nazwa-procedury;  
    //Instrukcje tworzące kod procedury;  
end nazwa-procedury;
```

Deklaracje procedur powinny być umieszczane w części deklaracyjnej programu.

# Pierwsza procedura

Napisać program zawierający procedurę, która powoduje wyzerowanie tablicy 256 podwójnych słów,  
począwszy od adresu przechowywanego w rejestrze  
EBX.

# Jeden z możliwych sposobów realizacji zadania



```
zerowanie.hla - Notatnik
Plik Edycja Format Widok Pomoc

program zerowanie;
#include( "stdlib.hhf");

    procedure zerujaca;
    begin zerujaca;

        mov( 0, eax );
        mov( 256, ecx );
        repeat

            mov( eax, [ebx] );    //wyzeruj bieżące podójne słowo pamięci.
            add( 4, ebx );        //Przesuń EBX do następnego podwójnego słowa.
            dec( ecx );           //Zmniejsz licznik podwójnych słów do wyzerowania.

        until( ecx = 0);        //Powtarzaj dla 256 podwójnych słów.

    end zerujaca;

static
    tablica:          dword[256];

begin zerowanie;
    lea( ebx, tablica );
    call zerujaca;

end zerowanie;
```

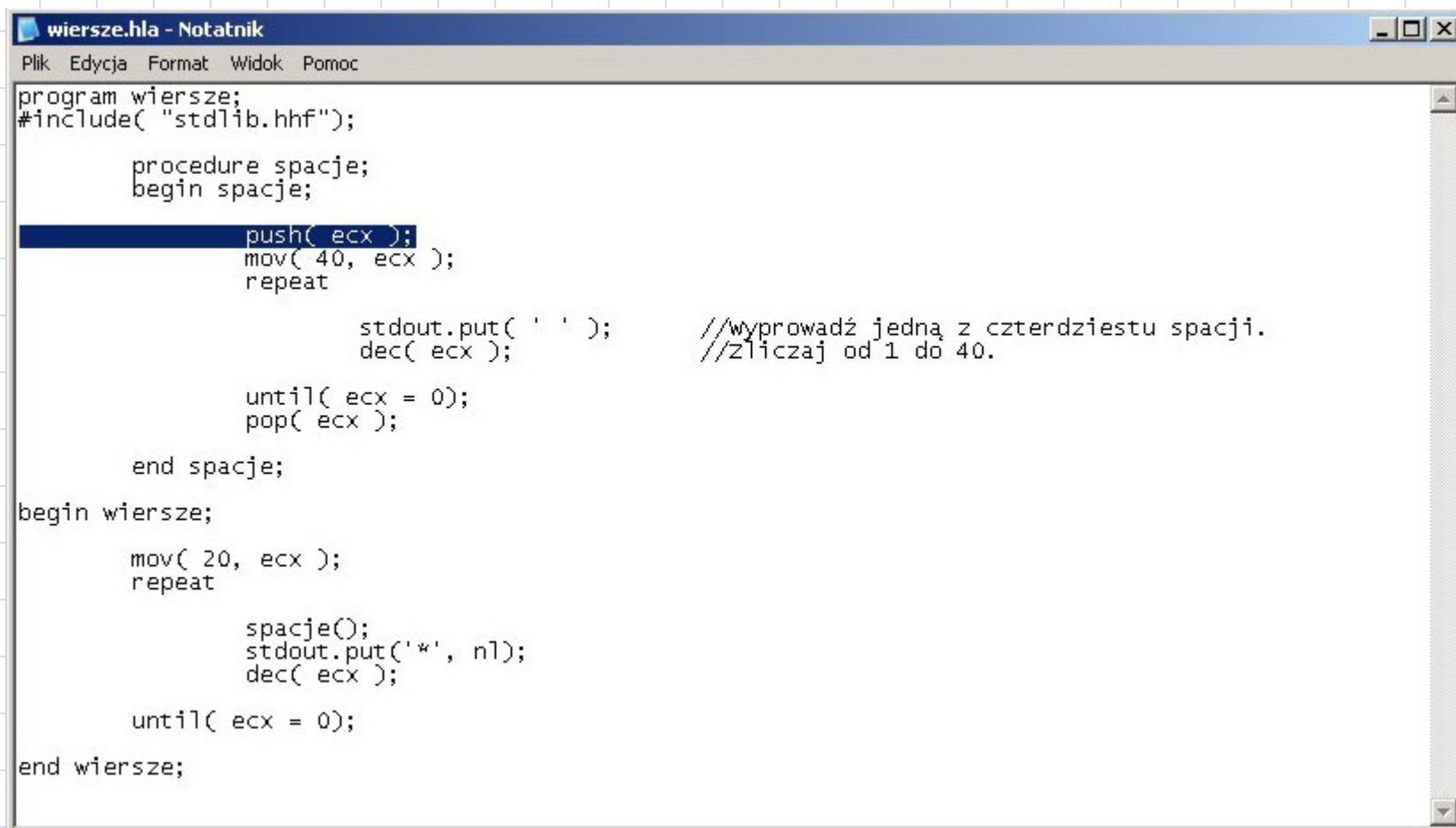


# Zachowanie stanu systemu

- ◆ Napisać procedurę, która wypisuje na ekranie 1-20 wierszy po 1-40 spacji uzupełnianych pojedynczym znakiem gwiazdki w każdym wierszu.
- ◆ Liczbę wierszy i spacji podaje użytkownik.
- ◆ Do wykorzystania jest tylko rejestr ECX.
- ◆ Procedura odlicza spacje, program główny odlicza wiersze.



# Jeden z możliwych sposobów realizacji zadania



```
wiersze.hla - Notatnik
Plik Edycja Format Widok Pomoc

program wiersze;
#include( "stdlib.hhf");

    procedure spacje;
    begin spacje;

        push( ecx );
        mov( 40, ecx );
        repeat

            stdout.put( ' ' );    //wyprowadź jedną z czterdziestu spacji.
            dec( ecx );           //Zliczaj od 1 do 40.

        until( ecx = 0);
        pop( ecx );

    end spacje;

begin wiersze;

    mov( 20, ecx );
    repeat

        spacje();
        stdout.put( '* ', nl);
        dec( ecx );

    until( ecx = 0);

end wiersze;
```

# Zachowywanie rejestrów

- ◆ Odpowiedzialność za zachowanie wartości modyfikowanych rejestrów może przyjąć
  - wywołujący (tak nazywany jest kontekst, z poziomu którego nastąpiło wywołanie procedury),
  - wywoływany (czyli sam kod procedury).

# Przedwczesny powrót z procedury

- ◆ Możliwe jest opuszczenie kody procedury przed osiągnięciem przez program właściwego końca procedury – opuszczenie takie umożliwiają instrukcje **exit** i **exitif**.
- ◆ W kodzie źródłowym programu wykorzystuje się je następująco:
  - **exit nazwa-procedury;**
  - **exitif( wyrażenie-logiczne ) nazwa-procedury;**

# Zmienne lokalne

- ◆ Są to zmienne dostępne wyłącznie w ramach kodu procedury, nie będąc dostępnymi z poziomu kodu wywołującego procedurę.
- ◆ Deklaracje zmiennych lokalnych są identyczne z deklaracjami zmiennych programu głównego – sekcję deklaracji takich zmiennych osadza się w części deklaracyjnej procedury (a nie programu).
- ◆ W części deklaracyjnej procedury można deklarować dowolne elementy, których deklarowanie jest dozwolone w części deklaracyjnej programu głównego: typy, stałe, zmienne, a nawet inne procedury.

# Zmienne lokalne

- ◆ Zmienne lokalne charakteryzują się dwoma atrybutami odróżniającymi je od zmiennych programu głównego (zmiennych globalnych):
  - zasięgiem leksykalnym,
    - ◆ określa on widoczność, (a tym samym możliwość wykorzystania) w kodzie źródłowym programu.
  - czasem życia.
    - ◆ określa momenty, w których dla zmiennej przydzielana jest pamięć i w których pamięć ta jest zwalniana – zmienna może przechowywać wartości jedynie pomiędzy tymi momentami.

# Zmienne globalne i zmienne lokalne

- ◆ W procedurach można odwoływać się do zmiennych lokalnych.
- ◆ Deklaracje procedur mogą występować w dowolnym miejscu części deklaracyjnej programu, a więc potencjalnie również przed miejscem zadeklarowania wykorzystywanej w procedurze zmiennej czy zmiennych.
- ◆ W kodzie procedury można odwołać się do dowolnego obiektu deklarowanego w sekcji *static*, *storage* bądź *readonly*, dostęp do takich obiektów realizowany jest identycznie jak z poziomu kodu programu głównego, a więc sprowadza się do określenia nazwy obiektu.
- ◆ Zmienne lokalne, deklarowane wewnątrz deklaracji procedury są dostępne wyłącznie z poziomu tej procedury.



# Free your mind...

◆ Zaproponować dowolny program, wykorzystujący obydwa rodzaje zmiennych:

- lokalne
- globalne

◆ ...

# Czas życia zmiennych

- ◆ Okres pomiędzy przydzieleniem pamięci dla zmiennej, a tej samej pamięci zwolnieniem.
- ◆ Czas życia zmiennej jest przy tym atrybutem dynamicznym
  - (kontrolowanym w fazie wykonania programu)
- ◆ w przeciwieństwie do zasięgu zmiennej, który jest atrybutem statycznym
  - (kontrolowanym w czasie kompilacji).

# Parametry

- ◆ W przypadku procedur sparametryzowanych, wykonanie kodu procedury uzależnione jest od określenia pewnych danych wejściowych.
- ◆ Istnieją dwa sposoby przekazywania do procedury wartości reprezentujących poszczególne parametry:
  - przekazywanie przez wartość
  - przekazywanie przez adres

# Przekazywanie przez wartość

- ◆ Argumenty przekazywane przez wartość pełnią zawsze rolę parametrów jednokierunkowych – konkretnie wejściowych.

**procedura1(I);**

- ◆ Jeśli zmienna I jest do procedury przekazywana przez wartość, to kod procedury nie może modyfikować zmiennej I.
- ◆ Argumenty wywołania procedury są przez domniemanie przekazywane przez wartość

# Przykłady wywołań z parametrami

- ◆ Procedurę pierwszaprooc można wywołać następująco:

pierwszaprooc( stała );

pierwszaprooc( rejestr32 );

- ◆ Przykłady poprawnych wywołań procedury pierwszaprooc:

pierwszaprooc( 40 );

pierwszaprooc( eax );

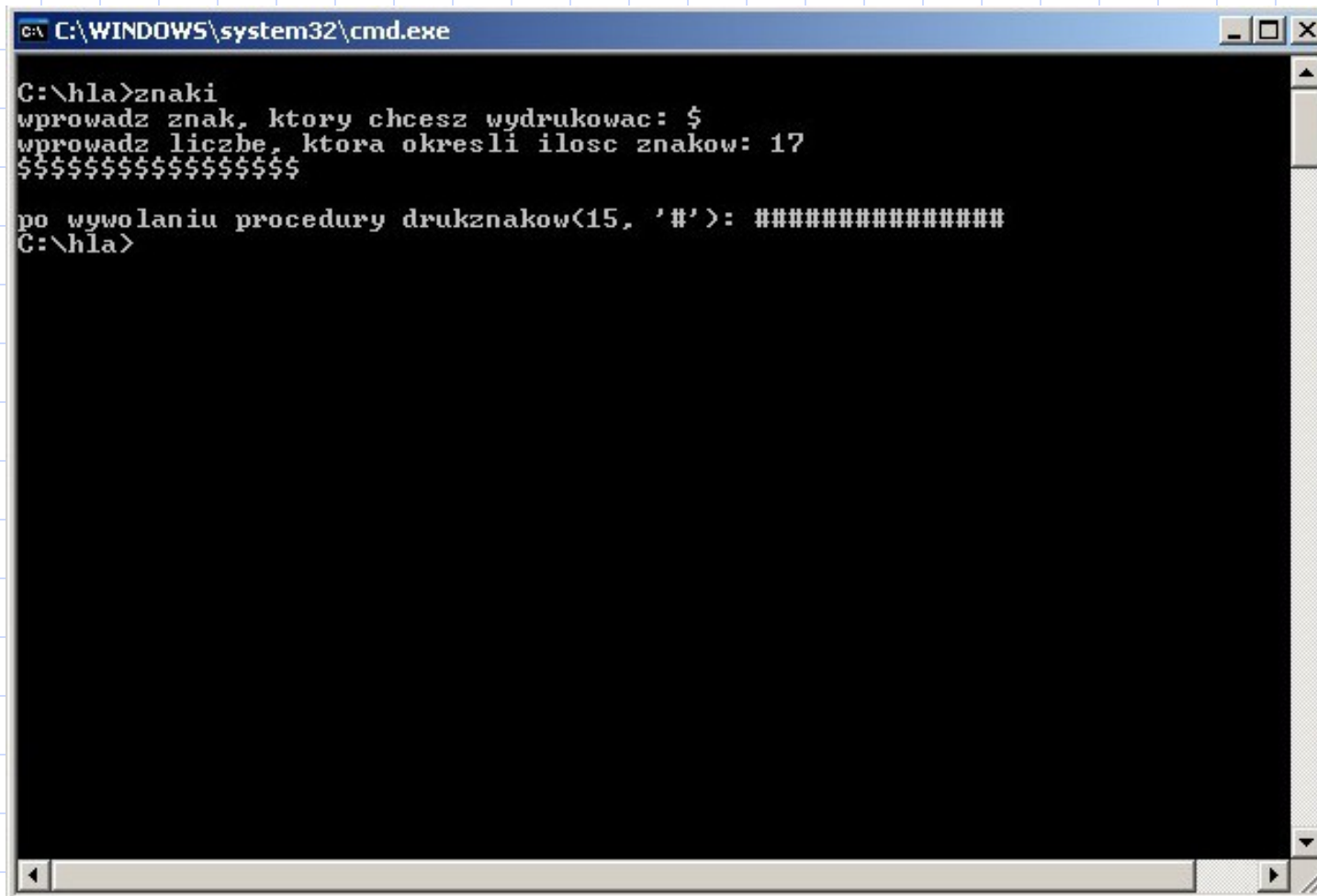
# Wiele parametrów wywołania

- ◆ Wykorzystywany jest zapis pozycyjny parametrów.
- ◆ Kolejne argumenty wywołania zostaną przez kompilator skojarzone z kolejnymi parametrami procedury, zgodnie z ich pozycją na liście parametrów procedury.
- ◆ Przy deklaracji procedury parametry oddzielane są znakiem ;
- ◆ Przy wywołaniu procedury parametry oddzielane są znakiem ,

## Procedura o dwóch parametrach wywołania

- ◆ Stworzyć dwie procedury, które drukują na ekranie określoną liczbę znaków (np. w zakresie od 1-20).
- ◆ Liczba znaków i wygląd znaku powinny być parametrami wywołania procedury.
- ◆ Liczba znaków i wygląd znaku wprowadzane są z klawiatury.

# Program znaki



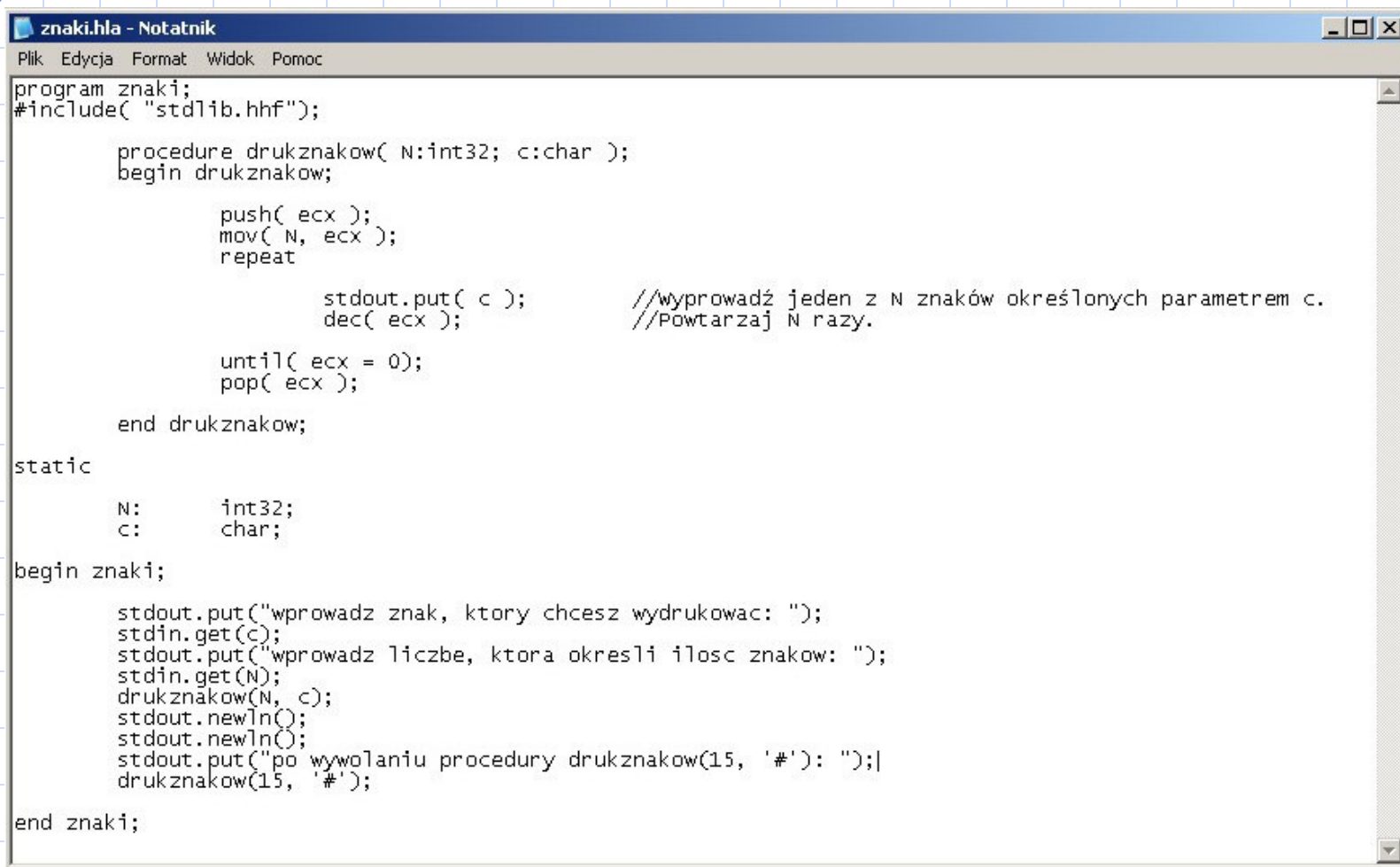
```
C:\WINDOWS\system32\cmd.exe

C:\hla>znaki
wprowadz znak, ktory chcesz wydrukowac: $
wprowadz liczbe, ktora okresli ilosc znakow: 17
$$$$$$$$$$$$$$$$$$$$

po wywołaniu procedury drukznakow(15, '#'): #####
C:\hla>
```



# Jeden z możliwych sposobów realizacji zadania:



```
znaki.hla - Notatnik
Plik Edycja Format Widok Pomoc

program znaki;
#include( "stdlibb.hhf");

    procedure drukznakow( N:int32; c:char );
    begin drukznakow;

        push( ecx );
        mov( N, ecx );
        repeat

            stdout.put( c );           //wyprowadź jeden z N znaków określonych parametrem c.
            dec( ecx );               //Powtarzaj N razy.

        until( ecx = 0);
        pop( ecx );

    end drukznakow;

static
    N:      int32;
    c:      char;

begin znaki;

    stdout.put("wprowadz znak, ktory chcesz wydrukowac: ");
    stdin.get(c);
    stdout.put("wprowadz liczbe, ktora okresli ilosc znakow: ");
    stdin.get(N);
    drukznakow(N, c);
    stdout.newln();
    stdout.newln();
    stdout.put("po wywołaniu procedury drukznakow(15, '#'): ");|
    drukznakow(15, '#');

end znaki;
```

# Przekazywanie przez adres

- ◆ Zwane również przekazywaniem przez referencję polega na przekazaniu – w miejsce wartości – adresu obiektu.
- ◆ Do procedury nie są przekazywane dane, a jedynie wskaźnik na te dane.
- ◆ Aby dla danego parametru określić tryb przekazywania przez adres, należy deklarację owego parametru poprzedzić słowem var:

`procedure nowaproc( var zmienna: int32 );`

Wywołanie procedury z przekazaniem argumentu przez referencję nie różni się od analogicznego wywołania z przekazaniem argumentu przez wartość, tyle że argument musi być określony jako adres w pamięci – argumentem nie może być ani stała, ani rejestr.

# Przykładowe wywołania procedur z przekazywaniem przez adres

Odwołania prawidłowe:

```
procedura1( i32 );    //zmienna i32 ma tu typ int32  
procedura1(type int32 [ebx]);
```

Odwołania nieprawidłowe:

```
procedura1( 40 );    //Niedozwolona w wywołaniu stała.  
procedura1( eax );   //Niedozwolony rejestr.  
procedura1( znak );  //Argument wywołania musi określać  
obiekt  
                     //typu zgodnego z typem parametru
```

# Przekazywanie przez adres

- ◆ Przekazywanie przez adres jest zwykle mniej efektywne niż przekazywanie przez wartość, ponieważ w kodzie procedury należy uwzględnić konieczność wyłuskiwania parametru w każdym odwołaniu do przekazanej wartości.
- ◆ Pośrednie odwołanie wymaga zwykle co najmniej dwóch instrukcji maszynowych, jednak w przypadku większych struktur danych efektywność przekazywania argumentów przez adres rośnie ponieważ alternatywą jest potencjalnie czasochłonne kopiowanie wszystkich bajtów takich struktur do pamięci parametrów procedury.

# Funkcje

- ◆ Funkcje to procedury, które zwracają wyniki.
- ◆ Efektem wykonania kodu procedury jest realizacja konkretnego zadania, w ramach funkcji podobna sekwencja instrukcji maszynowych służy do ustalenia pewnego wyniku (wartości funkcji), który jest następnie zwracany do wywołującego.
- ◆ Procedura staje się funkcją, kiedy programista zdecyduje o przekazaniu pewnej określonej i mającej jakieś umowne znaczenie wartości poza kod procedury.
- ◆ Sposób deklaracji funkcji nie różni się niczym od deklaracji procedury – składniowo są to obiekty identyczne.

# Przykład funkcji

- ◆ Funkcja **cs.member**.
- ◆ Wymaga ona określenia dwóch argumentów: wartości znakowej i zbioru znaków.
- ◆ Funkcja zwraca za pośrednictwem rejestru EAX wartość logiczną „prawda” (1), jeżeli znak należy do zbioru. Kiedy przekazany znak nie należy do określonego zbioru znaków, funkcja umieszcza w rejestrze EAX wartość logiczną „fałsz” (0).

# Zwracanie wartości funkcji

- ❖ Wartości funkcji najczęściej przekazywane są do wywołującego za pośrednictwem rejestrów ogólnego przeznaczenia.
- ❖ Obowiązuje ogólnie przyjęta konwencja w przypadku assemblerów dla procesorów rodziny 80x86, zakładająca zwracanie 8-bitowych, 16-bitowych i 32-bitowych wartości całkowitych za pośrednictwem rejestrów AL., AX i EAX (odpowiednio).
- ❖ Bardzo często funkcje zwracają wartości 64-bitowe w parze rejestrów EDX:EAX (na przykład funkcja `stdin.geti64` biblioteki standardowej języka HLA zwraca w tej parze rejestrów wczytaną z wejścia 64-bitową wartość całkowitą).

# Dowolna funkcja

- ◆ Stworzyć funkcję, efektem działania której będzie umieszczenie w rejestrach AI, AX, EAX wartości 0 lub 1 w zależności od wykonania programu lub w zależności od wprowadzonych danych przez użytkownika.



# Złożenie instrukcji

- ◆ Pozwala na wykorzystanie instrukcji w roli operandu innej instrukcji.
- ◆ Na przykład instrukcja *mov* przyjmuje dwa operandy: źródłowy i docelowy. Złożenie instrukcji polegałoby tu na zastąpieniu jednego z operandów (bądź obu) instrukcją maszynową z zestawu instrukcji procesora 80x86, np.:  
`mov( mov( 0, eax ), ebx );`
- ◆ Większość instrukcji maszynowych zwraca wartości, które przez asembler są odbierane w czasie asemblacji programu. Dla większości instrukcji wartością zwracaną jest **operand docelowy**.
- ◆ Z tego względu instrukcja `mov( 0, eax );` ma dla asemblera wartość „eax”.
- ◆ Wartości instrukcji – zwłaszcza tych występujących samodzielnie w wierszu programu – są przez kompilator ignorowane. Jednak kiedy instrukcja zostanie określona w miejsce operandu innej instrukcji, kompilator może skompletować ową instrukcję właśnie z pomocą wartości zwracanych instrukcji.
- ◆ Powyższa instrukcja *mov* jest równoważna następującej sekwencji instrukcji:  
`mov( 0, eax );`  
`mov( eax, ebx );`
- ◆ Instrukcje wewnętrzne złożenia są asemblowane w pierwszej kolejności.

# Złożenie instrukcji

- ◆ Przy przetwarzaniu instrukcji złożonych (czyli instrukcji zagnieżdżonych w miejscu operandu innych instrukcji) asembler przyjmuje strategię kompilacji „od lewej do prawej” oraz „od środka do zewnątrz”.

`add( sub( mov( i, eax ), mov( j, ebx )), mov(k, ecx ));`

- ◆ W wyniku asemblacji złożenia instrukcji asembler wygeneruje następującą sekwencję instrukcji:

`mov( i, eax );`

`mov( j, ebx);`

`sub( eax, ebx);`

`mov( k, ecx );`

`add( ebx, ecx);`

# Złożenie instrukcji języka HLA

- ◆ Niektóre z funkcji biblioteki standardowej są wykorzystywane jako operandy instrukcji całego języka.
- ◆ Przykładem jest kod:  

```
if( cs.member( al., { 'a' .. 'z' } ) ); then  
-  
endif;
```
- ◆ Funkcja *cs.member* sprawdza czy znak przechowywany w rejestrze AL reprezentuje znak małej litery alfabetu. Funkcja powinna zwrócić wartość logiczną, która zostanie zinterpretowana jako wartość warunku instrukcji *if*. Jeśli funkcja *cs.member* będzie miała wartość *false*, blok kodu określony pomiędzy klauzulami *then* i *endif* powinien zostać pominięty.

# Atrybut @returns procedur

- Możliwe jest określenie w deklaracji procedury specjalnego atrybutu, który określać będzie napis interpretowany przez kompilator jako „wartość zwracana” instrukcji wywołania danej procedury, kiedy wywołanie to zostanie w wyniku złożenia osadzone w miejscu operandu innej instrukcji. Składnia deklaracji procedury rozszerzona o atrybut @ returns prezentuje się następująco:

```
procedure nazwa-procedury ( opcjonalna-lista-parametrów );  
    @returns( napis );  
    //Deklaracje lokalne względem procedury;  
begin nazwa-procedury;  
    //Instrukcje tworzące kod procedury;  
end nazwa-procedury;
```

- atrybut @returns wymaga określenia pomiędzy znakami nawiasów pojedynczego literału łańcuchowego; assembler będzie podstawiał ów literał wszędzie tam, gdzie instrukcja wywołania procedury zostanie wykorzystana w roli operandu innej instrukcji.
- zwykle napis określa nazwę jednego z rejestrów ogólnego przeznaczenia, dopuszczalne jest jednak określenie zupełnie dowolnego literału łańcuchowego, który będzie nadawał się do wykorzystania w roli operandu instrukcji, na przykład napis może określać nazwę zmiennej albo stałą reprezentującą konkretny adres w pamięci.

# Funkcja logiczna

- ◆ Stworzyć funkcję logiczną, zwracającą wartość logiczną „prawda” (true) bądź „fałsz” (false) za pośrednictwem rejestru EDX.
- ◆ Wartość true sygnalizuje, że określona w wywołaniu wartość znakowa reprezentuje literę alfabetu.
- ◆ Funkcję wywołać w programie, który wyświetli komentarz stosowny do tego, czy wprowadzona została litera alfabetu czy też inny znak.

# Wskazówka

◆ Jeden z możliwych sposobów realizacji funkcji:

```
procedure litery( c:char ); @returns( "EAX" );  
begin litery;
```

```
    cs.member( c, { 'a' .. 'z', 'A' .. 'Z' } );
```

```
end litery;
```

Po uzupełnieniu procedury od atrybut @returns można już swobodnie wykorzystywać wywołanie **litery** w roli operandu innych instrukcji, np.

```
mov( litery( al ), ebx );  
-  
if( litery( ch ) ) then  
-  
endif;
```

# Zarządzanie dużymi projektami

- ◆ Rzadko który plik źródłowy zawierający kod w języku assemblerowym jest samodzielny programem. Z reguły program taki musi być wspomagany z zewnątrz, choćby wywołaniami biblioteki standardowej języka HLA.
- ◆ W przypadku niewielkich, prostych programów dopuszczalne jest operowanie jednym tylko plikiem kodu źródłowego. Większe programy wymagają już wyodrębnienia poszczególnych części do osobnych plików (podobnie jak wymaga tego kod biblioteki standardowej języka HLA).
- ◆ Podział taki ma szczególną zaletę. Jeżeli dany fragment programu został już wszechstronnie przetestowany, to jego każdorazowa asemlacja po wprowadzeniu w innej części kodu najmniejszej nawet zmiany jest bezcelowe.
- ◆ Cała biblioteka standardowa języka HLA asemluje się nawet kilka minut.

# Asemlacja rozłączna

- ◆ Asemlacja rozłączna poszczególnych części kodu polega na podzieleniu kodu na możliwie niezależne, łatwiejsze do zarządzania części.
- ◆ Każdą z takich części należy następnie asemlować do postaci pliku kodu obiektowego. Celem stworzenia działającego, kompletnego pliku kodu wykonywalnego, wszystkie moduły obiektowe poddawane są konsolidacji.
- ◆ W takim układzie niewielka zmiana w obrębie jednej z części programu powoduje konieczność kompilacji tylko tej jednej części – nie trzeba od nowa kompilować całości programu.
- ◆ Dokładnie taki schemat kompilacji zastosowany jest w bibliotece standardowej języka HLA, która jest rozprowadzana w formie skompilowanych do postaci obiektowej modułów.



# Dyrektywa `#include`

- ◆ Sygnałizuje konieczność wstawienia w miejscu dyrektywy zawartości pliku określonego argumentem dyrektywy.
- ◆ W tak wstawianym do kodu źródłowego pliku można na przykład zgrupować definicje wykorzystywanych w programie stałych, procedur czy innych obiektów programu.
- ◆ Definicje te można następnie wykorzystywać w wielu różnych programach.
- ◆ Dyrektywy `#include` mogą być zagnieżdżane również w plikach włączanych, a dalej w plikach włączanych do plików włączanych i tak dalej. włączenie jednego pliku do kompilacji może pociągnąć za sobą konieczność włączenia do niej całego zestawu plików.

# Dyrektywa `#include`

- ◆ Znakiem przykładowym zagnieżdżenia dyrektyw `#include` jest włączanie często do programów w języku HLA pliku `stdlib.hhf`. Plik ów nie jest niczym więcej jak tylko listą kolejnych plików do włączenia.
- ◆ Włączając do pliku kodu źródłowego plik `stdlib.hhf`, automatycznie włączamy do kodu wszystkie moduły biblioteki HLA.

dziękuję za uwagę  
za tydzień termin odróbkowy