Министерство образования Республики Беларусь

«ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. ЕВФРОСИНИИ ПОЛОЦКОЙ»

Факультет информационных технологий

Кафедра технологий программирования

Методические указания для выполнения лабораторной работы №2 по курсу «Конструирование программного обеспечения»

«Работа с массивами в языке низкого уровня»

Полоцк, 2022 г.

ЦЕЛЬ РАБОТЫ

Приобретение навыков использования массивов при программировании на языке ассемблера.

.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Массивы

Языки высокого уровня обычно являются типизированными. Каждая переменная имеет тип, который накладывает ограничения на операции над переменной и на использование в одном выражении переменных разных типов. Кроме того, языки высокого уровня позволяют работать со сложными типами, таким как указатели, записи/структуры, классы, массивы, строки, множества и т.п.

Язык Паскаль имеет достаточно жёсткую структуру типов. Присваивания между переменными разных типов минимальны, над указателями определены только операции присваивания, взятия значения и получение адреса. Поддерживается много сложных типов.

Язык C, который создавался как высокоуровневая замена языку ассемблера, имеет гораздо менее жёсткую структуру типов. Все целочисленные типы совместимы, тип char, конечно, хранит символы, но также сопоставим с целыми типами, логический тип отсутствует в принципе (для языка C это именно так!), над указателями определены операции сложения и вычитания. Сложные типы, такие как массивы, строки и множества, не поддерживаются.

Что касается языка ассемблера, то тут вообще вряд ли можно говорить о какой-либо структуре типов. Команды языка ассемблера оперируют объектами, существующими в оперативной памяти, т.е. байтом и его производными (слово, двойное слово и т.д.). Символьный, логический тип? Какая глупость! Указатели? Вот тебе 4 байта и делай с ними, что хочешь. В итоге, конечно, и можно сделать, что хочешь, только предварительно стоит хорошо подумать, что из этого получится.

Соответственно, в языке ассемблера существует 5 (!) директив для определения данных:

- DB (define byte) – определяет переменную размером в 1 байт;
- DW (define word) – определяет переменную размеров в 2 байта (слово);
- DD (define double word) – определяет переменную размером в 4 байта (двойное слово);
- DQ (define quad word) – определяет переменную размером в 8 байт (учетверённое слово);
- DT (define ten bytes) – определяет переменную размером в 10 байт.

Все директивы могут быть использованы как для объявления простых переменных, так и для объявления массивов. Хотя для определения строк, в принципе,

можно использовать любую директиву, в связи с особенностями хранения данных в оперативной памяти лучше использовать директиву DB.

Синтаксис директив определения данных следующий:

<имя> DB <операнд> [, <операнд>]
<имя> DW <операнд> [, <операнд>]
<имя> DD <операнд> [, <операнд>]
<имя> DQ <операнд> [, <операнд>]
<имя> DT <операнд> [, <операнд>]

Операнд задаёт начальное значение переменной. В качестве операнда может использоваться число, символ или знак вопроса, с помощью которого определяются неинициализированные переменные.

Если в качестве операнда указывается строка или если указано несколько операндов через запятую, то память отводится под несколько переменных указанного типа, т.е. получается массив. При этом именованным оказывается только первый элемент, а доступ к остальным элементам массива осуществляется с помощью выражения <имя> + <смещение>.

Для того чтобы не указывать несколько раз одно и то же значение, при инициализации массивов можно использовать конструкцию повторения DUP.

```
a db 10011001b ; Определяем переменную размером 1 байт с начальным
значением, заданным в двоичной системе счисления
b db '!' ; Определяем переменную в 1 байт, инициализируемую символом
'!'
d db 'string',13,10 ; Определяем массив из 8 байт
e db 'string',0 ; Определяем строку из 7 байт, заканчивающую нулём
f dw 1235o ; Определяем переменную размером 2 байта с начальным
значением, заданным в восьмеричной системе счисления
g dd -345d ; Определяем переменную размером 4 байта с начальным
значением, заданным в десятичной системе счисления
h dd 0f1ah ; Определяем переменную размером 4 байта с начальным
значением, заданным в шестнадцатеричной системе счисления
i dd ? ; Определяем неинициализированную переменную размером 4 байта
j dd 100 dup (0) ; Определяем массив из 100 двойных слов,
инициализированных 0
k dq 10 dup (0, 1, 2) ; Определяем массив из 30 учетверённых слов,
инициализированный повторяющимися значениями 0, 1 и 2
l dd 100 dup (?) ; Определяем массив из 100 неинициализированных
двойных слов
```

К переменным можно применить две операции – offset и type. Первая определяет адрес переменной, а вторая – размер переменной. Однако размер переменной определяется по директиве, и даже если с директивой,

например, DD определён массив из нескольких элементов, размер всё равно будет равен 4.

Модификация адресов

Как уже было сказано, массивы в языке ассемблера описываются по директивам определения данных с использованием конструкции повторения. Для того чтобы обратиться к элементу массива, необходимо так или иначе указать адрес начала массива и смещение элемента в массиве. Смещение первого элемента массива всегда равно 0. Смещения остальных элементов массива зависят от размера элементов.

Пусть X – некий массив. Тогда адрес элемента массива можно вычислить по следующей формуле:

```
адрес(X[i]) = X + (type X) * i, где i – номер элемента массива,
начинающийся с 0
```

Напомним, что имя переменной эквивалентно её адресу (для массива – адресу начала массива), а операция type определяет размер переменной (для массива определяется размер элемента массива в соответствии с использованной директивой).

Для удобства в языке ассемблера введена операция модификации адреса, которая схожа с индексным выражением в языках высокого уровня – к имени массива надо приписать целочисленное выражение или имя регистра в квадратных скобках:

```
x[4]
x[ebx]
```

Однако принципиальное отличие состоит в том, в программе на языке высокого уровня мы указываем индекс элемента массива, а компилятор умножает его на размер элемента массива, получая смещение элемента массива. В программе на языке ассемблера указывается именно смещение, т.е. программист должен сам учитывать размер элемента массива. Компилятор же языка ассемблера просто прибавляет смещение к указанному адресу. Приведённые выше команды можно записать по-другому:

```
x + 4
[x + 4]
[x] + [4]
[x][4]
[x + ebx]
[x] + [ebx]
[x][ebx]
```

Обратите внимание, что при использовании регистра для модификации адреса наличие квадратных скобок обязательно. В противном случае компилятор зафиксирует ошибку.

Адрес может вычисляться и по более сложной схеме:

```
<база> + <множитель> * <индекс> + <смещение>
```

База – это регистр или имя переменной. Индекс должен быть записан в некотором регистре. Множитель – это константа 1 (можно опустить), 2, 4 или 8. Смещение – целое положительное или отрицательное число.

```
mov eax, [ebx + 4 * ecx - 32]
mov eax, [x + 2 * ecx]
```

Команда LEA

Команда LEA осуществляет загрузку в регистр так называемого эффективного адреса:

```
LEA <регистр>, <ячейка памяти>
```

Команда не меняет флаги. В простейшем случае с помощью команды LEA можно загрузить в регистр адрес переменной или начала массива:

```
x dd 100 dup (0)
lea ebx, x
```

Однако поскольку адрес может быть вычислен с использованием операций сложения и умножения, команда LEA имеет также ряд других применений.

Обработка массивов

Пусть есть массив x и переменная n, хранящая количество элементов этого массива.

```
x dd 100 dup(?)
n dd ?
```

Для обработки массива можно использовать несколько способов.

1. В регистре можно хранить смещение элемента массива.

```
mov eax, 0
mov ecx, n
```

```
mov ebx, 0
L: add eax, x[ebx]
add ebx, type x
dec ecx
cmp ecx, 0
jne L
```

2. В регистре можно хранить номер элемента массива и умножать его на размер элемента.

```
mov eax, 0
mov ecx, n
L: dec ecx
add eax, x[ecx * type x]
cmp ecx, 0
jne L
```

3. В регистре можно хранить адрес элемента массива. Адрес начала массива можно записать в регистр с помощью команды LEA.

```
mov eax, 0
mov ecx, n
lea ebx, x
L: add eax, [ebx]
add ebx, type x
dec ecx
cmp ecx, 0
jne L
```

4. При необходимости можно в один регистр записать адрес начала массива, а в другой – номер или смещение элемента массива.

```
mov eax, 0
mov ecx, n
lea ebx, x
L: dec ecx
add eax, [ebx + ecx * type x]
cmp ecx, 0
jne L
```

Модификацию адреса можно производить также по двум регистрам: x[ebx][esi]. Это может быть удобно при работе со структурами данных, которые рассматриваются как матрицы. Рассмотрим для примера подсчёт количества строк матриц с положительной суммой элементов.

```
mov esi, 0 ; Начальное смещение строки
```

```
mov ebx, 0 ; EBX будет содержать количество строк, удовлетворяющих условию
mov ecx, m ; Загружаем в ECX количество строк
L1: mov edi, 0 ; Начальное смещение элемента в строке
mov eax, 0 ; EAX будет содержать сумму элементов строки
mov edx, n ; Загружаем в EDX количество элементов в строке
L2: add eax, y[esi][edi] ; Прибавляем к EAX элемент массива
add edi, type y ; Прибавляем к смещению элемента в строке размер элемента
dec edx ; Уменьшаем на 1 счётчик внутреннего цикла
cmp edx, 0 ; Сравниваем EDX с нулём
jne L2 ; Если EDX не равно 0, то переходим к началу цикла
cmp eax, 0 ; После цикла сравниваем сумму элементов строки с нулём
jle L3 ; Если сумма меньше или равна 0, то обходим увеличение EBX
inc ebx ; Если же сумму больше 0, то увеличиваем EBX
L3: mov eax, n ; Загружаем в EAX количество элементов в строке
imul eax, type y ; Умножаем количество элементов в строке на размер элемента
add esi, eax ; Прибавляем к смещению полученный размер строки
dec ecx ; Уменьшаем на 1 счётчик внешнего цикла
cmp ecx, 0 ; Сравниваем ECX с нулём
jne L1 ; Если ECX не равно 0, то переходим к началу цикла
```

Сортировка массивов

Метод пузырька

```
; Процедура bubble_sort
; сортирует массив слов методом пузырьковой сортировки
; ввод: DS:DI = адрес массива
;    DX = размер массива (в словах)
bubble_sort      proc near
     pusha
     cld
     cmp dx,1
     jbe sort_exit     ; выйти, если сортировать нечего
     dec dx
sb_loop1:
     mov cx,dx          ; установить длину цикла
     xor bx,bx        ; BX будет флагом обмена
     mov si,di        ; SI будет указателем на
                       ; текущий элемент
sn_loop2:
     lodsw                  ; прочитать следующее слово
     cmp ax,word ptr [si]
     jbe no_swap         ; если элементы не
                       ; в порядке,
     xchg ax,word ptr [si] ; поменять их местами
```

```
     mov word ptr [si-2],ax
     inc bx         ; и установить флаг в 1,
no_swap:
     loop on_loop2
     cmp bx,0           ; если сортировка не закончилась,
     jne sn_loop1         ; перейти к следующему элементу
sort_exit:
     popa
     ret
bubble_sort endp
```

Пузырьковая сортировка осуществляется так медленно потому, что сравнения выполняются лишь между соседними элементами. Чтобы получить более быстрый метод сортировки перестановкой, следует выполнять сравнение и перестановку элементов, отстоящих далеко друг от друга. На этой идее основан алгоритм, который называется «быстрая сортировка». Он работает следующим образом: делается предположение, что первый элемент является средним по отношению к остальным. На основе такого предположения все элементы разбиваются на две группы – больше и меньше предполагаемого среднего. Затем обе группы отдельно сортируются таким же методом. В худшем случае быстрая сортировка массива из N элементов требует N2 операций, но в среднем случае - только 2n*log2n сравнений и еще меньшее число перестановок.

Метод быстрой сортировки

```
; Процедура quick_sort
; сортирует массив слов методом быстрой сортировки
; ввод: DS:BX = адрес массива
;     DX = число элементов массива
quicksort proc near
cmp dx,1 ; Если число элементов 1 или 0,
jle qsort_done ; то сортировка уже закончилась
xor di,di      ; индекс для просмотра сверху (DI = 0)
mov si,dx ; индекс для просмотра снизу (SI = DX)
dec si ; SI = DX-1,        так как элементы нумеруются с нуля,
shl si,1 ; и умножить на 2,             так как это массив слов
mov ax,word ptr [bx] ; AX = элемент X1, объявленный средним
step_2: ; просмотр массива снизу, пока не встретится
           ; элемент, меньший или равный X1
cmp word ptr [bx][si],ax         ; сравнить XDI и X1
jle step_3           ; если XSI больше,
sub si,2            ; перейти к следующему снизу элементу
jmp short step_2 ; и продолжить просмотр
step_3:                          ; просмотр массива сверху, пока не
встретится
           ; элемент меньше X1                  или оба просмотра не
придут
```

```
           ; в одну точку
cmp si,di      ; если просмотры встретились,
je step_5                    ; перейти к шагу 5,
add di,2 ; иначе: перейти
                                        ; к следующему сверху элементу,
cmp word ptr [bx][di],ax         ; если он меньше X1,
jl step_3                    ; продолжить шаг 3
step_4:
; DI указывает на элемент, который не должен быть
; в верхней части, SI указывает на элемент,
; который не должен быть в нижней. Поменять их местами
mov cx,word ptr [bx][di]          ; CX = XDI
xchg cx,word ptr [bx][si]      ; CX = XSI, XSI = XDI
mov word ptr [bx][di],cx        ; XDI = CX
jmp short step_2
step_5: ; Просмотры встретились. Все элементы в нижней
     ; группе больше X1, все элементы в верхней группе
     ; и текущий - меньше или равны X1 Осталось
     ; поменять местами X1 и текущий элемент:
xchg ax,word ptr [bx][di]      ; AX = XDI, XDI = X1
mov word ptr [bx],ax          ; X1 = AX
; теперь можно отсортировать каждую из полученных групп
push dx
push di
push bx
mov    dx,di   ; длина массива X1...XDI-1
         shr dx,1      ; в DX
         call quick_sort ; сортировка
          pop bx
          pop di
          pop dx
          add bx,di      ; начало массива XDI+1...XN
          add bx,2       ; в BX
          shr di,1     ; длина массива XDI+1...XN
          inc di
          sub dx,di      ; в DX
          call quicksort ; сортировка
qsort_done: ret
quicksort     endp
```

Кроме того, что быстрая сортировка - самый известный пример алгоритма, использующего рекурсию, то есть вызывающего самого себя. Это еще и самая быстрая из сортировок «на месте», то есть сортировка, использующая только ту память, в которой хранятся элементы сортируемого массива. Можно доказать, что сортировку нельзя выполнить быстрее, чем за n*log2n операций, ни в худшем, ни в среднем случаях; и быстрая сортировка достаточно хорошо приближается к этому пределу в среднем случае. Сортировки, достигающие теоретического предела, тоже

существуют – это сортировки турнирным выбором и сортировки вставлением в сбалансированные деревья, но для их работы требуется резервирование дополнительной памяти. Так что, например, работа со сбалансированными деревьями будет происходить медленно из-за дополнительных затрат на поддержку сложных структур данных в памяти.

Порядок выполнения работы

1. Подсчитать в массиве количество элементов, равных введённому N;
2. Заменить в массиве все четные элементы нулями;
3. Определить номер первого равного нулю элемента;
4. Вывести на экран все нечетные элементы массива;
5. Отсортировать массив по возрастанию, используя любой известный алгоритм сортировки.

Содержание отчета

Отчет должен включать:

а) титульный лист;

б) формулировку цели работы;

в) описание результатов выполнения задания:

- листинги программ;
- результаты выполнения программ;

г) выводы, согласованные с целью работы.