**ипы данных**

При программировании на языке ассемблера используются данные следующих типов:

1. ***Непосредственные*** данные, представляющие собой числовые или символьные значения, являющиеся частью команды.   
   Непосредственные данные формируются программистом в процессе написания программы для конкретной команды ассемблера.
2. Данные [***простого типа***](https://i-assembler.ru/25/Text/Data.htm#ПростогоТипа), описываемые с помощью ограниченного набора директив резервирования памяти, позволяющих выполнить самые элементарные операции по размещению и инициализации числовой и символьной информации. При обработке этих директив ассемблер сохраняет в своей таблице символов информацию о местоположении данных (значения сегментной составляющей адреса и смещения) и типе данных, то есть единицах памяти, выделяемых для размещения данных в соответствии с директивой резервирования и инициализации данных.

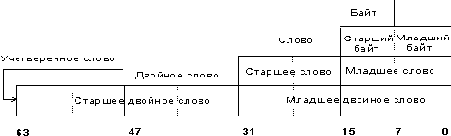
Эти два типа данных являются *элементарными*, или *базовыми*; работа с ними поддерживается на уровне системы команд микропроцессора. Используя данные этих типов, можно формализовать и запрограммировать практически любую задачу. Но насколько это будет удобно — вот вопрос.

1. Данные [***сложного типа***](https://i-assembler.ru/25/Text/Data.htm#СложногоТипа), которые были введены в язык ассемблера с целью облегчения разработки программ. Сложные типы данных строятся на основе базовых типов, которые являются как бы кирпичиками для их построения. Введение сложных типов данных позволяет несколько сгладить различия между языками высокого уровня и ассемблером. У программиста появляется возможность сочетания преимуществ языка ассемблера и языков высокого уровня (в направлении абстракции данных), что в конечном итоге повышает эффективность конечной программы.

Обработка информации, в общем случае, процесс очень сложный. Это косвенно подтверждает популярность языков высокого уровня. Одно из несомненных достоинств языков высокого уровня — поддержка развитых структур данных. При их использовании программист освобождается от решения конкретных проблем, связанных с представлением числовых или символьных данных, и получает возможность оперировать информацией, структура которой в большей степени отражает особенности предметной области решаемой задачи. В то же самое время, чем выше уровень такой абстракции данных от конкретного их представления в компьютере, тем большая нагрузка ложится на компилятор с целью создания действительно эффективного кода. Ведь нам уже известно, что в конечном итоге все написанное на языке высокого уровня в компьютере будет представлено на уровне машинных команд, работающих только с базовыми типами данных. Таким образом, самая эффективная программа — программа, написанная в машинных кодах, но писать сегодня большую программу в машинных кодах — занятие не имеющее слишком большого смысла.

Понятие *простого* типа данных носит двойственный характер. С точки зрения размерности (***физическая интерпретация***), микропроцессор аппаратно поддерживает следующие основные типы данных (*[рис. 1](https://i-assembler.ru/25/Text/Data.htm" \l "Рис_1)*):

* *байт* — восемь последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом;
* *слово* — последовательность из двух байт, имеющих последовательные адреса. Размер слова — 16 бит; биты в слове нумеруются от 0 до 15. Байт, содержащий нулевой бит, называется младшим байтом, а байт, содержащий 15-й бит - старшим байтом. Микропроцессоры Intel имеют важную особенность — ***младший байт всегда хранится по меньшему адресу. Адресом слова считается адрес его младшего байта. Адрес старшего байта может быть использован для доступа к старшей половине слова***.
* *двойное слово* — последовательность из четырех байт (32 бита), расположенных по последовательным адресам. Нумерация этих бит производится от 0 до 31. Слово, содержащее нулевой бит, называется младшим словом, а слово, содержащее 31-й бит, - старшим словом. Младшее слово хранится по меньшему адресу. Адресом двойного слова считается адрес его младшего слова. Адрес старшего слова может быть использован для доступа к старшей половине двойного слова.
* *учетверенное слово* — последовательность из восьми байт (64 бита), расположенных по последовательным адресам. Нумерация бит производится от 0 до 63. Двойное слово, содержащее нулевой бит, называется младшим двойным словом, а двойное слово, содержащее 63-й бит, — старшим двойным словом. Младшее двойное слово хранится по меньшему адресу. Адресом учетверенного слова считается адрес его младшего двойного слова. Адрес старшего двойного слова может быть использован для доступа к старшей половине учетверенного слова.



***Рис. 1.*** *Основные типы данных микропроцессора*

Кроме трактовки типов данных с точки зрения их разрядности, микропроцессор на уровне команд поддерживает ***логическую интерпретацию*** этих типов (*[рис. 2](https://i-assembler.ru/25/Text/Data.htm" \l "Рис_2)*):

* *Целый тип со знаком* — двоичное значение со знаком, размером 8, 16 или 32 бита. Знак в этом двоичном числе содержится в 7, 15 или 31-м бите соответственно. Ноль в этих битах в операндах соответствует положительному числу, а единица — отрицательному. Отрицательные числа представляются в дополнительном коде. Числовые диапазоны для этого типа данных следующие:
  + 8-разрядное целое — от –128 до +127;
  + 16-разрядное целое — от –32 768 до +32 767;
  + 32-разрядное целое — от –231 до +231–1.
* *Целый тип без знака* — двоичное значение без знака, размером 8, 16 или 32 бита. Числовой диапазон для этого типа следующий:
  + байт — от 0 до 255;
  + слово — от 0 до 65 535;
  + двойное слово — от 0 до 232–1.
* *Указатель на память двух типов:*
  + ближнего типа — 32-разрядный логический адрес, представляющий собой относительное смещение в байтах от начала сегмента. Эти указатели могут также использоваться в сплошной (плоской) модели памяти, где сегментные составляющие одинаковы;
  + дальнего типа — 48-разрядный логический адрес, состоящий из двух частей: 16-разрядной сегментной части — селектора, и 32-разрядного смещения.
* *Цепочка* — представляющая собой некоторый непрерывный набор байтов, слов или двойных слов максимальной длины до 4 Гбайт.
* *Битовое поле* представляет собой непрерывную последовательность бит, в которой каждый бит является независимым и может рассматриваться как отдельная переменная. Битовое поле может начинаться с любого бита любого байта и содержать до 32 бит.
* *Неупакованный двоично-десятичный тип* — байтовое представление десятичной цифры от 0 до 9. Неупакованные десятичные числа хранятся как байтовые значения без знака по одной цифре в каждом байте. Значение цифры определяется младшим полубайтом.
* *Упакованный двоично-десятичный тип* представляет собой упакованное представление двух десятичных цифр от 0 до 9 в одном байте. Каждая цифра хранится в своем полубайте. Цифра в старшем полубайте (биты 4–7) является старшей.

https://i-assembler.ru/25/Pic/Ris2_5.gif

***Рис. 2.*** *Основные логические типы данных микропроцессора*

Отметим, что **“Зн”** на рис. 2 означает знаковый бит.

После всего сказанного было бы логичным возникновение у читателя вопроса: как описать эти простые типы данных ассемблера, а затем и воспользоваться ими в программе? Ведь любая программа предназначена для обработки некоторой информации, поэтому вопрос о том, как описать данные с использованием средств языка обычно встает одним из первых.

TASM предоставляет очень широкий набор средств описания и обработки данных, который вполне сравним с аналогичными средствами некоторых языков высокого уровня.

Для описания простых типов данных в программе используются *специальные директивы резервирования и инициализации данных*, которые, по сути, являются указаниями транслятору на выделение определенного объема памяти. Если проводить аналогию с языками высокого уровня, то директивы резервирования и инициализации данных являются определениями переменных.   
*Машинного эквивалента этим директивам нет; просто транслятор, обрабатывая каждую такую директиву, выделяет необходимое количество байт памяти и при необходимости инициализирует эту область некоторым значением*.

Директивы резервирования и инициализации данных простых типов имеют формат, показанный на рис. 3.

https://i-assembler.ru/25/Pic/Ris5_17.gif

***Рис. 3.*** *Директивы описания данных простых типов*

На рис. 3 использованы следующие обозначения:

* **?** показывает, что содержимое поля не определено, то есть при задании директивы с таким значением выражения содержимое выделенного участка физической памяти изменяться не будет. Фактически, создается неинициализированная переменная;
* **значение инициализации** — значение элемента данных, которое будет занесено в память после загрузки программы. Фактически, создается инициализированная переменная, в качестве которой могут выступать константы, строки символов, константные и адресные выражения в зависимости от типа данных. Подробная информация приведена в приложении 1;
* **выражение** — итеративная конструкция с синтаксисом, описанным на рис. 5.17. Эта конструкция позволяет повторить последовательное занесение в физическую память выражения в скобках n раз.
* **имя** — некоторое символическое имя метки или ячейки памяти в сегменте данных, используемое в программе.

На рис. 3 представлены следующие поддерживаемые TASM директивы резервирования и инициализации данных:

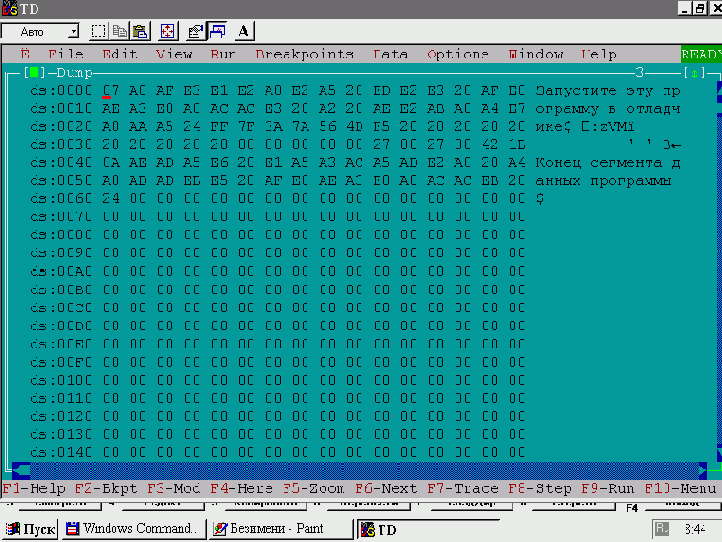
* **db** — резервирование памяти для данных размером 1 байт.   
  Директивой **db** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - для чисел со знаком –128...+127;
    - для чисел без знака 0...255;
  + 8-битовое относительное выражение, использующее операции HIGH и LOW;
  + символьную строку из одного или более символов. Строка заключается в кавычки. В этом случае определяется столько байт, сколько символов в строке.
* **dw** — резервирование памяти для данных размером 2 байта.   
  Директивой **dw** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - для чисел со знаком –32 768...32 767;
    - для чисел без знака 0...65 535;
  + выражение, занимающее 16 или менее бит, в качестве которого может выступать смещение в 16-битовом сегменте или адрес сегмента;
  + 1- или 2-байтовую строку, заключенная в кавычки.
* **dd** — резервирование памяти для данных размером 4 байта.   
  Директивой **dd** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - для i8086:
      * для чисел со знаком –32 768...+32 767;
      * для чисел без знака 0...65 535;
    - для i386 и выше:
      * для чисел со знаком –2 147 483 648...+2 147 483 647;
      * для чисел без знака 0...4 294 967 295;
  + относительное или адресное выражение, состоящее из 16-битового адреса сегмента и 16-битового смещения;
  + строку длиной до 4 символов, заключенную в кавычки.
* **df** — резервирование памяти для данных размером 6 байт;
* **dp** — резервирование памяти для данных размером 6 байт.   
  Директивами **df** и **dp** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - для i8086:
      * для чисел со знаком –32 768...+32 767;
      * для чисел без знака 0...65 535;
    - для i386 и выше:
      * для чисел со знаком –2 147 483 648...+2 147 483 647;
      * для чисел без знака 0...4 294 967 295;
  + относительное или адресное выражение, состоящее из 32 или менее бит (для i80386) или 16 или менее бит (для младших моделей микропроцессоров Intel);
  + адресное выражение, состоящее из 16-битового сегмента и 32-битового смещения;
  + константу со знаком из диапазона –247...247–1;
  + константу без знака из диапазона 0...248-1;
  + строку длиной до 6 байт, заключенную в кавычки.
* **dq** — резервирование памяти для данных размером 8 байт.   
  Директивой **dq** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - для МП i8086:
      * для чисел со знаком –32 768...+32 767;
      * для чисел без знака 0...65 535;
    - для МП i386 и выше:
      * для чисел со знаком –2 147 483 648...+2 147 483 647;
      * для чисел без знака 0...4 294 967 295;
  + относительное или адресное выражение, состоящее из 32 или менее бит (для i80386) или 16 или менее бит (для младших моделей микропроцессоров Intel);
  + константу со знаком из диапазона –263...263–1;
  + константу без знака из диапазона 0...264–1;
  + строку длиной до 8 байт, заключенную в кавычки.
* **dt** — резервирование памяти для данных размером 10 байт.   
  Директивой **dt** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - для МП i8086:
      * для чисел со знаком –32 768...+32 767;
      * для чисел без знака 0...65 535;
    - для МП i386 и выше:
      * для чисел со знаком –2 147 483 648...+2 147 483 647;
      * для чисел без знака 0...4 294 967 295;
  + относительное или адресное выражение, состоящее из 32 или менее бит (для i80386) или 16 или менее бит (для младших моделей);
  + адресное выражение, состоящее из 16-битового сегмента и 32-битового смещения;
  + константу со знаком из диапазона –279...279-1;
  + константу без знака из диапазона 0...280-1;
  + строку длиной до 10 байт, заключенную в кавычки;
  + упакованную десятичную константу в диапазоне 0...99 999 999 999 999 999 999.

Очень важно уяснить себе порядок размещения данных в памяти. Он напрямую связан с логикой работы микропроцессора с данными. Микропроцессоры Intel требуют следования данных в памяти по принципу: **младший байт по младшему адресу**.

Для иллюстрации данного принципа рассмотрим листинг 1, в котором определим сегмент данных. В этом сегменте данных приведено несколько директив описания простых типов данных.

|  |
| --- |
| Листинг 1. Пример использования директив резервирования и инициализации данных  masm  model small  .stack 100h  .data  message db 'Запустите эту программу в отладчике’,’$'  perem\_1 db 0ffh  perem\_2 dw 3a7fh  perem\_3 dd 0f54d567ah  mas db 10 dup (' ')  pole\_1 db 5 dup (?)  adr dw perem\_3  adr\_full dd perem\_3  fin db 'Конец сегмента данных программы $'  .code  start:  mov ax,@data  mov ds,ax  mov ah,09h  mov dx,offset message  int 21h  mov ax,4c00h  int 21h  end start |

Теперь наша цель — посмотреть, как выглядит сегмент данных программы листинга 1 в памяти компьютера. Это даст нам возможность обсудить практическую реализацию обозначенного нами принципа размещения данных. Для этого запустим отладчик TD.EXE, входящий в комплект поставки TASM. Результат показан на рис. 4.



***Рис. 4.*** *Окно дампа памяти для программы листинга 1*

Обсудим рис. 4. На нем вы видите данные вашего сегмента в двух представлениях: шестнадцатеричном и символьном. Видно, что со смещением **0000** расположены символы, входящие в строку *message*. Она занимает 34 байта. После нее следует байт, имеющий в сегменте данных символическое имя *perem\_1*, содержимое этого байта **offh**.   
Теперь обратите внимание на то, как размещены в памяти байты, входящие в слово, обозначенное символическим именем *perem\_2*. Сначала следует байт со значением **7fh**, а затем со значением **3ah**. Как видите, в памяти действительно сначала расположен младший байт значения, а затем старший.   
Теперь посмотрите и самостоятельно проанализируйте размещение байтов для поля, обозначенного символическим именем *perem\_3*.   
Оставшуюся часть сегмента данных вы можете теперь проанализировать самостоятельно.   
Остановимся лишь на двух специфических особенностях использования директив резервирования и инициализации памяти. Речь идет о случае использования в поле операндов директив **dw** и **dd** символического имени из поля имя этой или другой директивы резервирования и инициализации памяти. В нашем примере сегмента данных это директивы с именами *adr* и *adr\_full*.   
Когда транслятор встречает директивы описания памяти с подобными операндами, то он формирует в памяти значения адресов тех переменных, чьи имена были указаны в качестве операндов. В зависимости от директивы, применяемой для получения такого адреса, формируется либо полный адрес (директива **dd**) в виде двух байтов сегментного адреса и двух байтов смещения, либо только смещение (директива **dw**). Найдите в дампе на рис. 4 поля, соответствующие именам *adr* и *adr\_full*, и проанализируйте их содержимое.

Любой переменной, объявленной с помощью директив описания простых типов данных, ассемблер присваивает три атрибута:

1. Сегмент (**seg**) — адрес начала сегмента, содержащего переменную;
2. Смещение (**offset**) в байтах от начала сегмента с переменной;
3. Тип (**type**) — определяет количество памяти, выделяемой переменной в соответствии с директивой объявления переменной.

Получить и использовать значение этих атрибутов в программе можно с помощью рассмотренных нами операторов ассемблера **[seg](https://i-assembler.ru/25/Text/Structur.htm" \l "SEG)**, [**offset**](https://i-assembler.ru/25/Text/Structur.htm#OFFSET) и [**type**](https://i-assembler.ru/25/Text/Structur.htm).

TASM поддерживает следующие сложные типы данных:

* **[массивы](https://i-assembler.ru/25/Text/Data.htm" \l "массивы)**;
* [**структуры**](https://i-assembler.ru/25/Text/Data.htm#структуры);
* [**объединения**](https://i-assembler.ru/25/Text/Data.htm#объединения);
* [**записи**](https://i-assembler.ru/25/Text/Data.htm#записи).

Разберемся более подробно с тем, как определить данные этих типов в программе и организовать работу с ними.

**Массивы**

Дадим формальное определение:   
***массив*** *- структурированный тип данных, состоящий из некоторого числа элементов одного типа.*

Для того чтобы разобраться в возможностях и особенностях обработки массивов в программах на ассемблере, нужно ответить на следующие вопросы:

* *[Как описать массив](https://i-assembler.ru/25/Text/Data.htm" \l "КакОписатьМассив)* в программе?
* [*Как инициализировать массив*](https://i-assembler.ru/25/Text/Data.htm#КакОписатьМассив), то есть как задать начальные значения его элементов?
* [*Как организовать доступ*](https://i-assembler.ru/25/Text/Data.htm#КакОрганизоватьДоступ) к элементам массива?
* [*Как организовать массивы*](https://i-assembler.ru/25/Text/Data.htm#ДвумерныеМассивы) с размерностью более одной?
* [*Как организовать выполнение*](https://i-assembler.ru/25/Text/Data.htm#ТиповыеОперации) типовых операций с массивами?

**Описание и инициализация массива в программе**

Специальных средств описания массивов в программах ассемблера, конечно, нет. При необходимости использовать массив в программе его нужно моделировать одним из следующих способов:

1. Перечислением элементов массива в поле операндов одной из директив описания данных. При перечислении элементы разделяются запятыми. К примеру:

|  |
| --- |
| ;массив из 5 элементов.Размер каждого  элемента 4 байта:  mas dd 1,2,3,4,5 |

1. Используя оператор повторения **dup**. К примеру:

|  |
| --- |
| ;массив из 5 нулевых элементов.  ;Размер каждого элемента 2 байта:  mas dw 5 dup (0) |

Такой способ определения используется для резервирования памяти с целью размещения и инициализации элементов массива.

1. Используя директивы **label** и **rept**. Пара этих директив может облегчить описание больших массивов в памяти и повысить наглядность такого описания. Директива **rept** относится к макросредствам языка ассемблера и вызывает повторение указанное число раз строк, заключенных между директивой и строкой endm. К примеру, определим массив байт в области памяти, обозначенной идентификатором *mas\_b*. В данном случае директива **label** определяет символическое имя *mas\_b*, аналогично тому, как это делают директивы резервирования и инициализации памяти. Достоинство директивы **label** в том, что она не резервирует память, а лишь определяет характеристики объекта. В данном случае объект — это ячейка памяти. Используя несколько директив **label**, записанных одна за другой, можно присвоить одной и той же области памяти разные имена и разный тип, что и сделано в следующем фрагменте:

|  |
| --- |
| ...  n=0  ...  mas\_b label byte  mas\_w label word  rept 4  dw 0f1f0h  endm |

В результате в памяти будет создана последовательность из четырех слов **f1f0**. Эту последовательность можно трактовать как массив байт или слов в зависимости от того, какое имя области мы будем использовать в программе — *mas\_b* или *mas\_w*.

1. Использование цикла для инициализации значениями области памяти, которую можно будет впоследствии трактовать как массив.   
   Посмотрим на примере листинга 2, каким образом это делается.

|  |
| --- |
| Листинг 2 Инициализация массива в цикле  ;prg\_12\_1.asm  MASM  MODEL small  STACK 256  .data  mes db 0ah,0dh,'Массив- ','$'  mas db 10 dup (?) ;исходный массив  i db 0  .code  main:  mov ax,@data  mov ds,ax  xor ax,ax ;обнуление ax  mov cx,10 ;значение счетчика цикла в cx  mov si,0 ;индекс начального элемента в cx  go: ;цикл инициализации  mov bh,i ;i в bh  mov mas[si],bh ;запись в массив i  inc i ;инкремент i  inc si ;продвижение к следующему элементу массива  loop go ;повторить цикл  ;вывод на экран получившегося массива  mov cx,10  mov si,0  mov ah,09h  lea dx,mes  int 21h  show:  mov ah,02h ;функция вывода значения из al на экран  mov dl,mas[si]  add dl,30h ;преобразование числа в символ  int 21h  inc si  loop show  exit:  mov ax,4c00h ;стандартный выход  int 21h  end main ;конец программы |

**Доступ к элементам массива**

При работе с массивами необходимо четко представлять себе, что все элементы массива располагаются в памяти компьютера последовательно.   
Само по себе такое расположение ничего не говорит о назначении и порядке использования этих элементов. И только лишь программист с помощью составленного им алгоритма обработки определяет, как нужно трактовать эту последовательность байт, составляющих массив. Так, одну и ту же область памяти можно трактовать как одномерный массив, и одновременно те же самые данные могут трактоваться как двухмерный массив. Все зависит только от алгоритма обработки этих данных в конкретной программе. Сами по себе данные не несут никакой информации о своем *“смысловом”*, или *логическом*, типе. Помните об этом принципиальном моменте.

Эти же соображения можно распространить и на ***индексы элементов массива***. Ассемблер не подозревает об их существовании и ему абсолютно все равно, каковы их численные смысловые значения.   
Для того чтобы локализовать определенный элемент массива, к его имени нужно добавить *индекс*. Так как мы моделируем массив, то должны позаботиться и о моделировании индекса. В языке ассемблера индексы массивов — это обычные адреса, но с ними работают особым образом. Другими словами, когда при программировании на ассемблере мы говорим об индексе, то скорее подразумеваем под этим не номер элемента в массиве, а некоторый адрес.

Давайте еще раз обратимся к описанию массива. К примеру, в программе статически определена последовательность данных:

|  |
| --- |
| mas dw 0,1,2,3,4,5 |

Пусть эта последовательность чисел трактуется как одномерный массив. Размерность каждого элемента определяется директивой **dw**, то есть она равна **2** байта. Чтобы получить доступ к третьему элементу, нужно к адресу массива прибавить **6**. Нумерация элементов массива в ассемблере начинается с нуля.   
То есть в нашем случае речь, фактически, идет о **4**-м элементе массива — 3, но об этом знает только программист; микропроцессору в данном случае все равно — ему нужен только адрес.

В общем случае для получения адреса элемента в массиве необходимо начальный (базовый) адрес массива сложить с произведением индекса (номер элемента минус единица) этого элемента на размер элемента массива:

**база + (индекс\*размер элемента)**

Архитектура микропроцессора предоставляет достаточно удобные программно-аппаратные средства для работы с массивами. К ним относятся базовые и индексные регистры, позволяющие реализовать несколько режимов адресации данных. Используя данные режимы адресации, можно организовать эффективную работу с массивами в памяти. Вспомним эти режимы:

* ***индексная адресация со смещением*** — режим адресации, при котором эффективный адрес формируется из двух компонентов:
  + **постоянного (базового)** — указанием прямого адреса массива в виде имени идентификатора, обозначающего начало массива;
  + **переменного (индексного)** — указанием имени индексного регистра.   
    К примеру:

|  |
| --- |
| mas dw 0,1,2,3,4,5  ...  mov si,4  ;поместить 3-й элемент массива mas в регистр ax:  mov ax,mas[si] |

* ***базовая индексная адресация со смещением*** — режим адресации, при котором эффективный адрес формируется максимум из трех компонентов:
  + **постоянного** (необязательный компонент), в качестве которой может выступать прямой адрес массива в виде имени идентификатора, обозначающего начало массива, или непосредственное значение;
  + **переменного (базового)** — указанием имени базового регистра;
  + **переменного (индексного)** — указанием имени индексного регистра.

Этот вид адресации удобно использовать при обработке двухмерных массивов. Пример использования этой адресации мы рассмотрим далее при изучении особенностей работы с двухмерными массивами.

Напомним, что в качестве базового регистра может использоваться любой из восьми регистров общего назначения. В качестве индексного регистра также можно использовать любой регистр общего назначения, за исключением esp/sp.

Микропроцессор позволяет ***масштабировать*** индекс. Это означает, что если указать после имени индексного регистра знак умножения **“\*”** с последующей цифрой 2, 4 или 8, то содержимое индексного регистра будет умножаться на 2, 4 или 8, то есть *масштабироваться*.

Применение масштабирования облегчает работу с массивами, которые имеют размер элементов, равный 2, 4 или 8 байт, так как микропроцессор сам производит коррекцию индекса для получения адреса очередного элемента массива. Нам нужно лишь загрузить в индексный регистр значение требуемого индекса (считая от 0). Кстати сказать, возможность масштабирования появилась в микропроцессорах Intel, начиная с модели i486. По этой причине в рассматриваемом здесь примере программы стоит директива **.486**. Ее назначение, как и ранее использовавшейся директивы **.386**, в том, чтобы указать ассемблеру при формировании машинных команд на необходимость учета и использования дополнительных возможностей системы команд новых моделей микропроцессоров.

В качестве примера использования масштабирования рассмотрим листинг 3, в котором просматривается массив, состоящий из слов, и производится сравнение этих элементов с нулем. Выводится соответствующее сообщение.

|  |
| --- |
| Листинг 3. Просмотр массива слов с использованием  масштабирования  ;prg\_12\_2.asm  MASM  MODEL small  STACK 256  .data ;начало сегмента данных  ;тексты сообщений:  mes1 db 'не равен 0!$',0ah,0dh  mes2 db 'равен 0!$',0ah,0dh  mes3 db 0ah,0dh,'Элемент $'  mas dw 2,7,0,0,1,9,3,6,0,8 ;исходный массив  .code  .486 ;это обязательно  main:  mov ax,@data  mov ds,ax ;связка ds с сегментом данных  xor ax,ax ;обнуление ax  prepare:  mov cx,10 ;значение счетчика цикла в cx  mov esi,0 ;индекс в esi  compare:  mov dx,mas[esi\*2] ;первый элемент массива в dx  cmp dx,0 ;сравнение dx c 0  je equal ;переход, если равно  not\_equal: ;не равно  mov ah,09h ;вывод сообщения на экран  lea dx,mes3  int 21h  mov ah,02h ;вывод номера элемента массива на экран  mov dx,si  add dl,30h  int 21h  mov ah,09h  lea dx,mes1  int 21h  inc esi ;на следующий элемент  dec cx ;условие для выхода из цикла  jcxz exit ;cx=0? Если да — на выход  jmp compare ;нет — повторить цикл  equal: ;равно 0  mov ah,09h ;вывод сообщения mes3 на экран  lea dx,mes3  int 21h  mov ah,02h  mov dx,si  add dl,30h  int 21h  mov ah,09h ;вывод сообщения mes2 на экран  lea dx,mes2  int 21h  inc esi ;на следующий элемент  dec cx ;все элементы обработаны?  jcxz exit  jmp compare  exit:  mov ax,4c00h ;стандартный выход  int 21h  end main ;конец программы |

Еще несколько слов о соглашениях:

* Если для описания адреса используется только один регистр, то речь идет о *базовой адресации* и этот регистр рассматривается как *базовый*:

|  |
| --- |
| ;переслать байт из области данных, адрес  которой находится в регистре ebx:  mov al,[ebx] |

* Если для задания адреса в команде используется *прямая адресация* (в виде идентификатора) в сочетании с одним регистром, то речь идет об *индексной адресации*. Регистр считается *индексным*, и поэтому можно использовать масштабирование для получения адреса нужного элемента массива:

|  |
| --- |
| add eax,mas[ebx\*4]  ;сложить содержимое eax с двойным словом в памяти  ;по адресу mas + (ebx)\*4 |

* Если для описания адреса используются два регистра, то речь идет о *базово-индексной адресации*. Левый регистр рассматривается как *базовый*, а правый — как *индексный*. В общем случае это не принципиально, но если мы используем масштабирование с одним из регистров, то он всегда является *индексным*. Но лучше придерживаться определенных соглашений.   
  Помните, что применение регистров **ebp/bp** и **esp/sp** по умолчанию подразумевает, что сегментная составляющая адреса находится в регистре **ss**.

Заметим, что *базово-индексную адресацию* не возбраняется сочетать с *прямой адресацией* или указанием непосредственного значения. Адрес тогда будет формироваться как сумма всех компонентов.

К примеру:

|  |
| --- |
| mov ax,mas[ebx][ecx\*2]  ;адрес операнда равен [mas+(ebx)+(ecx)\*2]  ...  sub dx,[ebx+8][ecx\*4]  ;адрес операнда равен [(ebx)+8+(ecx)\*4] |

Но имейте в виду, что *масштабирование* эффективно лишь тогда, когда размерность элементов массива равна 2, 4 или 8 байт. Если же размерность элементов другая, то организовывать обращение к элементам массива нужно обычным способом, как описано ранее.

Рассмотрим пример работы с массивом из пяти трехбайтовых элементов (листинг 4). Младший байт в каждом из этих элементов представляет собой некий счетчик, а старшие два байта — что-то еще, для нас не имеющее никакого значения. Необходимо последовательно обработать элементы данного массива, увеличив значения счетчиков на единицу.

|  |
| --- |
| Листинг 4. Обработка массива элементов с нечетной длиной  ;prg\_11\_3.asm  MASM  MODEL small ;модель памяти  STACK 256 ;размер стека  .data ;начало сегмента данных  N=5 ;количество элементов массива  mas db 5 dup (3 dup (0))  .code ;сегмент кода  main: ;точка входа в программу  mov ax,@data  mov ds,ax  xor ax,ax ;обнуление ax  mov si,0 ;0 в si  mov cx,N ;N в cx  go:  mov dl,mas[si] ;первый байт поля в dl  inc dl ;увеличение dl на 1 (по условию)  mov mas[si],dl ;заслать обратно в массив  add si,3 ;сдвиг на следующий элемент массива  loop go ;повтор цикла  mov si,0 ;подготовка к выводу на экран  mov cx,N  show: ;вывод на экран содержимого  ;первых байт полей  mov dl,mas[si]  add dl,30h  mov ah,02h  int 21h  loop show  exit:  mov ax,4c00h ;стандартный выход  int 21h  end main ;конец программы |

**Двухмерные массивы**

С представлением *одномерных* массивов в программе на ассемблере и организацией их обработки все достаточно просто. А как быть если программа должна обрабатывать *двухмерный* массив? Все проблемы возникают по-прежнему из-за того, что специальных средств для описания такого типа данных в ассемблере нет. *Двухмерный* массив нужно моделировать. На описании самих данных это почти никак не отражается — память под массив выделяется с помощью директив резервирования и инициализации памяти.

Непосредственно моделирование обработки массива производится в сегменте кода, где программист, описывая алгоритм обработки ассемблеру, определяет, что некоторую область памяти необходимо трактовать как двухмерный массив.   
При этом вы вольны в выборе того, как понимать расположение элементов двухмерного массива в памяти: по строкам или по столбцам.

Если последовательность однотипных элементов в памяти трактуется как двухмерный массив, расположенный по строкам, то адрес элемента **(i, j)** вычисляется по формуле

**(база + количество\_элементов\_в\_строке \* размер\_элемента \* i+j)**

Здесь **i = 0...n–1** указывает *номер строки*, а **j = 0...m–1** указывает *номер столбца*.

Например, пусть имеется массив чисел (размером в 1 байт) *mas(i, j)* с размерностью 4 на 4   
**(i= 0...3, j = 0...3)**:

23 04 05 67

05 06 07 99

67 08 09 23

87 09 00 08

В памяти элементы этого массива будут расположены в следующей последовательности:

**23 04 05 67 05 06 07 99 67 08 09 23 87 09 00 08**

Если мы хотим трактовать эту последовательность как двухмерный массив, приведенный выше, и извлечь, например, элемент   
**mas(2, 3) = 23**, то проведя нехитрый подсчет, убедимся в правильности наших рассуждений:

**Эффективный адрес mas(2, 3) = mas + 4 \* 1 \* 2 + 3 = mas + 11**

Посмотрите на представление массива в памяти и убедитесь, что по этому смещению действительно находится нужный элемент массива.

Организовать адресацию двухмерного массива логично, используя рассмотренную нами ранее *базово-индексную адресацию*. При этом возможны два основных варианта выбора компонентов для формирования эффективного адреса:

* сочетание прямого адреса, как базового компонента адреса, и двух индексных регистров для хранения индексов:

|  |
| --- |
| mov ax,mas[ebx][esi] |

* сочетание двух индексных регистров, один из которых является и базовым и индексным одновременно, а другой — только индексным:

|  |
| --- |
| mov ax,[ebx][esi] |

В программе это будет выглядеть примерно так:

|  |
| --- |
| ;Фрагмент программы выборки элемента  ;массива mas(2,3) и его обнуления  .data  mas db  23,4,5,67,5,6,7,99,67,8,9,23,87,9,0,8  i=2  j=3  .code  ...  mov si,4\*1\*i  mov di,j  mov al,mas[si][di] ;в al элемент mas(2,3)  ... |

В качестве законченного примера рассмотрим программу поиска элемента в двухмерном массиве чисел (листинг 5). Элементы массива заданы статически.

|  |
| --- |
| Листинг 5. Поиск элемента в двухмерном массиве  ;prg\_11\_4.asm  MASM  MODEL small  STACK 256  .data  ;матрица размером 2x5 — если ее не инициализировать,  ;то для наглядности она может быть описана так:  ;array dw 2 DUP (5 DUP (?))  ;но мы ее инициализируем:  array dw 1,2,3,4,5,6,7,3,9,0  ;логически это будет выглядеть так:  ;array= {1 2}  ; {3 4}  ; {5 6}  ; {7 3}  ; {9 0}  elem dw 3 ;элемент для поиска  failed db 0ah,0dh,'Нет такого элемента в массиве!','$'  success db 0ah,0dh,'Такой элемент в массиве присутствует ','$'  foundtime db ? ;количество найденных элементов  fnd db ' раз(а)',0ah,0dh,'$'  .code  main:  mov ax,@data  mov ds,ax  xor ax,ax  mov si,0 ;si=столбцы в матрице  mov bx,0 ;bx=строки в матрице  mov cx,5 ;число для внешнего цикла (по строкам)  external: ;внешний цикл по строкам  mov ax,array[bx][si] ;в ax первый элемент матрицы  push cx ;сохранение в стеке счётчика внешнего цикла  mov cx,2 ;число для внутреннего цикла (по столбцам)  mov si,0  iternal: ;внутренний цикл по строкам  inc si ;передвижение на следующий элемент в строке  ;сравниваем содержимое текущего элемента в ax с искомым элементом:  cmp ax,elem  ;если текущий совпал с искомым, то переход на here для обработки,  ;иначе цикл продолжения поиска  je here  ;иначе — цикл по строке cx=2 раз  loop iternal  here:  jcxz move\_next ;просмотрели строку?  inc foundtime ;иначе увеличиваем счётчик совпавших  move\_next: ;продвижение в матрице  pop cx ;восстанавливаем CX из стека (5)  add bx,1 ;передвигаемся на следующую строку  loop external ;цикл (внешний)  cmp foundtime,0h ;сравнение числа совпавших с 0  ja eql ;если больше 0, то переход  not\_equal: ;нет элементов, совпавших с искомым  mov ah,09h ;вывод сообщения на экран  mov dx,offset failed  int 21h  jmp exit ;на выход  eql: ;есть элементы, совпавшие с искомым  mov ah,09h ;вывод сообщений на экран  mov dx,offset success  int 21h  mov ah,02h  mov dl,foundtime  add dl,30h  int 21h  mov ah,09h  mov dx,offset fnd  int 21h  exit: ;выход  mov ax,4c00h ;стандартное завершение программы  int 21h  end main ;конец программы |

При анализе работы программы не забывайте, что в языке ассемблера принято элементы массива нумеровать с 0. При поиске определенного элемента массив просматривается от начала и до конца.   
Приведенная программа сохраняет в поле *foundtime* количество вхождений искомого элемента в массив. В качестве индексных регистров используются **si** и **bx**.

**Типовые операции с массивами**

Для демонстрации основных приемов работы с массивами лучше всего подходят программы поиска или сортировки.

Рассмотрим одну такую программу, выполняющую сортировку массива по возрастанию (листинг 6).

|  |
| --- |
| Листинг 6. Сортировка массива  <1> ;prg\_12\_5.asm  <2> MASM  <3> MODEL small  <4> STACK 256  <5> .data  <6> mes1 db 0ah,0dh,'Исходный массив — $',0ah,0dh  <7> ;некоторые сообщения  <8> mes2 db 0ah,0dh,'Отсортированный массив — $',0ah,0dh  <9> n equ 9 ;количество элементов в массиве, считая с 0  <10> mas dw 2,7,4,0,1,9,3,6,5,8 ;исходный массив  <11> tmp dw 0 ;переменные для работы с массивом  <12> i dw 0  <13> j dw 0  <14> .code  <15> main:  <16> mov ax,@data  <17> mov ds,ax  <18> xor ax,ax  <19> ;вывод на экран исходного массива  <20> mov ah,09h  <21> lea dx,mes1  <22> int 21h ;вывод сообщения mes1  <23> mov cx,10  <24> mov si,0  <25> show\_primary: ;вывод значения элементов  <26> ;исходного массива на экран  <27> mov dx,mas[si]  <28> add dl,30h  <29> mov ah,02h  <30> int 21h  <31> add si,2  <32> loop show\_primary  <33>  <34> ;строки 40-85 программы эквивалентны следующему коду на языке С:  <35> ;for (i=0;i<9;i++)  <36> ; for (j=9;j>i;j--)  <37> ; if (mas[i]>mas[j])  <38> ; {tmp=mas[i];  <39> ; mas[i]=mas[j];  <40> ; mas[j]=tmp;}  <41> mov i,0 ;инициализация i  <42> ;внутренний цикл по j  <43> internal:  <44> mov j,9 ;инициализация j  <45> jmp cycl\_j ;переход на тело цикла  <46> exchange:  <47> mov bx,i ;bx=i  <48> shl bx,1  <49> mov ax,mas[bx] ;ax=mas[i]  <50> mov bx,j ;bx=j  <51> shl bx,1  <52> cmp ax,mas[bx] ;mas[i] ? mas[j] — сравнение элементов  <53> jle lesser ;если mas[i] меньше, то обмен не нужен и  ;переход на продвижение далее по массиву  <54> ;иначе tmp=mas[i], mas[i]=mas[j], mas[j]=tmp:  <55> ;tmp=mas[i]  <56> mov bx,i ;bx=i  <57> shl bx,1 ;умножаем на 2, так как элементы — слова  <58> mov tmp,ax ;tmp=mas[i]  <59>  <60> ;mas[i]=mas[j]  <61> mov bx,j ;bx=j  <62> shl bx,1 ;умножаем на 2, так как элементы — слова  <63> mov ax,mas[bx] ;ax=mas[j]  <64> mov bx,i ;bx=i  <65> shl bx,1 ;умножаем на 2, так как элементы — слова  <66> mov mas[bx],ax ;mas[i]=mas[j]  <67>  <68> ;mas[j]=tmp  <69> mov bx,j ;bx=j  <70> shl bx,1 ;умножаем на 2, так как элементы — слова  <71> mov ax,tmp ;ax=tmp  <72> mov mas[bx],ax ;mas[j]=tmp  <73> lesser: ;продвижение далее по массиву во внутреннем цикле  <74> dec j ;j--  <75>;тело цикла по j  <76> cycl\_j:  <77> mov ax,j ;ax=j  <78> cmp ax,i ;сравнить j ? i  <79> jg exchange ;если j>i, то переход на обмен  <80> ;иначе на внешний цикл по i  <81> inc i ;i++  <82> cmp i,n ;сравнить i ? n — прошли до конца массива  <83> jl internal ;если i  <85> ;вывод отсортированного массива  <86> mov ah,09h  <87> lea dx,mes2  <88> int 21h  <89> prepare:  <90> mov cx,10  <91> mov si,0  <92> show: ;вывод значения элемента на экран  <93> mov dx,mas[si]  <94> add dl,30h  <95> mov ah,02h  <96> int 21h  <97> add si,2  <98> loop show  <99> exit:  <100> mov ax,4c00h ;стандартный выход  <101> int 21h  <102> end main ;конец программы |

В основе программы лежит алгоритм, похожий на метод пузырьковой сортировки. Эта программа не претендует на безусловную оптимальность, так как существует целая теория, касающаяся подобного типа сортировок. Перед нами стоит другая цель — показать использование средств ассемблера для решения подобного рода задач.   
В программе два цикла.   
Внешний цикл определяет позицию в массиве очередного элемента, с которым производится попарное сравнение элементов правой части массива (относительно этого элемента). За каждую итерацию внешнего цикла на месте этого очередного элемента оказывается меньший элемент из правой части массива (если он есть).   
В остальном программа достаточно проста и на языке высокого уровня заняла бы около десятка строк.

**Структуры**

Рассмотренные нами выше массивы представляют собой совокупность однотипных элементов. Но часто в приложениях возникает необходимость рассматривать некоторую совокупность данных разного типа как некоторый единый тип.   
Это очень актуально, например, для программ баз данных, где необходимо связывать совокупность данных разного типа с одним объектом.   
К примеру, ранее мы рассмотрели [листинг 4](https://i-assembler.ru/25/Text/Data.htm" \l "Листинг_4), в котором работа производилась с массивом трехбайтовых элементов. Каждый элемент, в свою очередь, представлял собой два элемента разных типов: однобайтовое поле счетчика и двухбайтовое поле, которое могло нести еще какую-то нужную для хранения и обработки информацию. Если читатель знаком с одним из языков высокого уровня, то он знает, что такой объект обычно описывается с помощью специального типа данных — *структуры*.   
С целью повысить удобство использования языка ассемблера в него также был введен такой тип данных.

По определению ***структура*** — это тип данных, состоящий из фиксированного числа элементов разного типа.

Для использования структур в программе необходимо выполнить три действия:

1. [Задать](https://i-assembler.ru/25/Text/Data.htm" \l "ЗадатьШаблон) *[шаблон структуры](https://i-assembler.ru/25/Text/Data.htm" \l "ЗадатьШаблон)*.   
   По смыслу это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.
2. [Определить *экземпляр структуры*](https://i-assembler.ru/25/Text/Data.htm#ОпределитьЭкземпляр).   
   Этот этап подразумевает инициализацию конкретной переменной заранее определенной (с помощью шаблона) структурой.
3. Организовать [*обращение к элементам структуры*](https://i-assembler.ru/25/Text/Data.htm#РаботаСоСтруктурой).

Очень важно, чтобы вы с самого начала уяснили, в чем разница между *описанием* структуры в программе и ее *определением*.   
***Описать*** структуру в программе означает лишь указать ее схему или шаблон; память при этом не выделяется.   
Этот шаблон можно рассматривать лишь как информацию для транслятора о расположении полей и их значении по умолчанию.   
***Определить*** структуру — значит, дать указание транслятору выделить память и присвоить этой области памяти символическое имя.

Описать структуру в программе можно только один раз, а определить — любое количество раз.

**Описание шаблона структуры**

Описание шаблона структуры имеет следующий синтаксис:

|  |
| --- |
| имя\_структуры STRUC  <описание полей>  имя\_структуры ENDS |

Здесь ***<описание полей>*** представляет собой последовательность директив описания данных **db, dw, dd, dq** и **dt**.   
Их операнды определяют размер полей и, при необходимости, начальные значения. Этими значениями будут, возможно, инициализироваться соответствующие поля при определении структуры.

Как мы уже отметили при описании шаблона, память не выделяется, так как это всего лишь информация для транслятора.

*Местоположение* шаблона в программе может быть поизвольным, но, следуя логике работы однопроходного транслятора, он должен быть расположен до того места, где определяется переменная с типом данной структуры. То есть при описании в сегменте данных переменной с типом некоторой структуры ее шаблон необходимо поместить в начале сегмента данных либо перед ним.

Рассмотрим работу со структурами на примере моделирования базы данных о сотрудниках некоторого отдела.   
Для простоты, чтобы уйти от проблем преобразования информации при вводе, условимся, что все поля символьные.   
Определим структуру записи этой базы данных следующим шаблоном:

|  |
| --- |
| worker struc ;информация о сотруднике  nam db 30 dup (' ') ;фамилия, имя, отчество  sex db 'м' ;пол, по умолчанию 'м' — мужской  position db 30 dup (' ') ;должность  age db 2 dup(‘ ’) ;возраст  standing db 2 dup(‘ ’) ;стаж  salary db 4 dup(‘ ’) ;оклад в рублях  birthdate db 8 dup(‘ ’) ;дата рождения  worker ends |

**Определение данных с типом структуры**

Для использования описанной с помощью шаблона структуры в программе необходимо определить переменную с типом данной структуры. Для этого используется следующая синтаксическая конструкция:

|  |
| --- |
| [имя переменной] имя\_структуры <[список значений]> |

Здесь:

* *имя переменной* — идентификатор переменной данного структурного типа.   
  Задание имени переменной необязательно. Если его не указать, будет просто выделена область памяти размером в сумму длин всех элементов структуры.
* *список значений* — заключенный в угловые скобки список начальных значений элементов структуры, разделенных запятыми.   
  Его задание также необязательно.   
  Если список указан не полностью, то все поля структуры для данной переменной инициализируются значениями из шаблона, если таковые заданы.   
  Допускается инициализация отдельных полей, но в этом случае пропущенные поля должны отделяться запятыми. Пропущенные поля будут инициализированы значениями из шаблона структуры. Если при определении новой переменной с типом данной структуры мы согласны со всеми значениями полей в ее шаблоне (то есть заданными по умолчанию), то нужно просто написать угловые скобки.   
  К примеру: **victor worker <>**.

Для примера определим несколько переменных с типом описанной выше структуры.

|  |
| --- |
| data segment  sotr1 worker <’Гурко Андрей Вячеславович’,,’художник’,’33’,‘15’,‘1800’,’26.01.64’<  sotr2 worker <’Михайлова Наталья Геннадьевна’,’ж’,’программист’,’30’,’10’,’1680’,’27.10.58’<  sotr3 worker <’Степанов Юрий Лонгинович’,,’художник’,’38’,’20’,’1750’,’01.01.58’<  sotr4 worker <’Юрова Елена Александровна’,’ж’,’свяэист’,’32’,’2’,,’09.01.66’<  sotr5 worker <> ;здесь все значения по умолчанию  data ends |

**Методы работы со структурой**

Идея введения структурного типа в любой язык программирования состоит в объединении разнотипных переменных в один объект.   
В языке должны быть средства доступа к этим переменным внутри конкретного экземпляра структуры. Для того чтобы сослаться в команде на поле некоторой структуры, используется специальный оператор — *символ "****.****" (точка)*. Он используется в следующей синтаксической конструкции:

|  |
| --- |
| адресное\_выражение.имя\_поля\_структуры |

Здесь:

* *адресное\_выражение* — идентификатор переменной некоторого структурного типа или выражение в скобках в соответствии с указанными ниже синтаксическими правилами (рис. 1);
* *имя\_поля\_структуры* — имя поля из шаблона структуры.  
  Это, на самом деле, тоже адрес, а точнее, смещение поля от начала структуры.

Таким образом оператор "**.**" (точка) вычисляет выражение

|  |
| --- |
| (адресное\_выражение) + (имя\_поля\_структуры) |

https://i-assembler.ru/25/Pic/Ris12_1.gif

***Рис. 5.*** *Синтаксис адресного выражения в операторе обращения к полю структуры*

Продемонстрируем на примере определенной нами структуры ***worker*** некоторые приемы работы со структурами.   
К примеру, извлечь в **ax** значения поля с возрастом. Так как вряд ли возраст трудоспособного человека будет больше величины 99 лет, то после помещения содержимого этого символьного поля в регистр **ax** его будет удобно преобразовать в двоичное представление командой **[aad](https://i-assembler.ru/25/Text/Command.htm" \l "aad)**.   
Будьте внимательны, так как из-за принципа хранения данных *“младший байт по младшему адресу”* старшая цифра возраста будет помещена в **al**, а младшая — в **ah**.   
Для корректировки достаточно использовать команду **xchg al,ah**:

|  |
| --- |
| mov ax,word ptr sotr1.age ;в al возраст sotr1  xchg ah,al |

а можно и так:

|  |
| --- |
| lea bx,sotr1  mov ax,word ptr [bx].age  xchg ah,al |

Давайте представим, что сотрудников не четверо, а намного больше, и к тому же их число и информация о них постоянно меняются. В этом случае теряется смысл явного определения переменных с типом ***worker*** для конкретных личностей.   
Язык ассемблера разрешает определять не только отдельную переменную с типом структуры, но и массив структур.   
К примеру, определим массив из 10 структур типа ***worker***:

|  |
| --- |
| mas\_sotr worker 10 dup (<>) |

Дальнейшая работа с массивом структур производится так же, как и с одномерным массивом. Здесь возникает несколько вопросов:  
*Как быть с размером и как организовать индексацию элементов массива?*

Аналогично другим идентификаторам, определенным в программе, транслятор назначает имени типа структуры и имени переменной с типом структуры атрибут типа. Значением этого атрибута является размер в байтах, занимаемый полями этой структуры. Извлечь это значение можно с помощью оператор **type**.   
После того как стал известен размер экземпляра структуры, организовать индексацию в массиве структур не представляет особой сложности.   
К примеру:

|  |
| --- |
| worker struc  ...  worker ends  ...  mas\_sotr worker 10 dup (<>)  ...  mov bx,type worker ;bx=77  lea di,mas\_sotr  ;извлечь и вывести на экран пол всех сотрудников:  mov cx,10  cycl:  mov al,[di].sex  ...  ;вывод на экран содержимого поля sex структуры worker  add di,bx ;к следующей структуре в массиве mas\_sort  loop cycl |

Как выполнить копирование поля из одной структуры в соответствующее поле другой структуры? Или как выполнить копирование всей структуры? Давайте выполним копирование поля *nam* третьего сотрудника в поле *nam* пятого сотрудника:

|  |
| --- |
| worker struc  ...  worker ends  ...  mas\_sotr worker 10 dup (<>)  ...  mov bx,offset mas\_sotr  mov si,(type worker)\*2 ;si=77\*2  add si,bx  mov di,(type worker)\*4 ;si=77\*4  add di,bx  mov cx,30  rep movsb |

Мне кажется, что ремесло программиста рано или поздно делает человека похожим на хорошую домохозяйку. Он, подобно ей, постоянно находится в поиске, где бы чего-нибудь сэкономить, урезать и из минимума продуктов сделать прекрасный обед. И если это удается, то и моральное удовлетворение получается ничуть не меньше, а может и больше, чем от прекрасного обеда у домохозяйки. Степень этого удовлетворения, как мне кажется, зависит от степени любви к своей профессии.   
С другой стороны, успехи в разработке программного и аппаратного обеспечения несколько расслабляют программиста, и довольно часто наблюдается ситуация, похожая на известную пословицу про муху и слона, - для решения некоторой мелкой задачи привлекаются тяжеловесные средства, эффективность которых, в общем случае, значима только при реализации сравнительно больших проектов.

Наличие в языке следующих двух типов данных, наверное, объясняется стремлением “хозяйки” максимально эффективно использовать рабочую площадь стола (оперативной памяти) при приготовлении еды или для размещения продуктов (данных программы).

**Объединения**

Представим ситуацию, когда мы используем некоторую область памяти для размещения некоторого объекта программы (переменной, массива или структуры). Вдруг после некоторого этапа работы у нас отпала надобность в использовании этих данных. Обычно память останется занятой до конца работы программы. Конечно, в принципе, ее можно было бы использовать для хранения других переменных, но при этом без принятия специальных мер нельзя изменить тип и имя. Неплохо было бы иметь возможность переопределить эту область памяти для объекта с другим типом и именем. Язык ассемблера предоставляет такую возможность в виде специального типа данных, называемого *объединением*.

***Объединение*** — тип данных, позволяющий трактовать одну и ту же область памяти как имеющую разные типы и имена.

*Описание* объединений в программе напоминает описание структур, то есть сначала описывается **шаблон**, в котором с помощью директив описания данных перечисляются имена и типы полей:

|  |
| --- |
| имя\_объединения UNION  <описание полей>  имя\_объединения ENDS |

Отличие объединений от структур состоит, в частности, в том, что при определении переменной типа объединения память выделяется в соответствии с размером максимального элемента.   
Обращение к элементам объединения происходит по их именам, но при этом нужно, конечно, помнить о том, что все поля в объединении накладываются друг на друга.   
Одновременная работа с элементами объединения исключена. В качестве элементов объединения можно использовать и структуры.

Листинг 7, который мы сейчас рассмотрим, примечателен тем, что кроме демонстрации использования собственно типа данных **“объединение”** в нем показывается возможность взаимного вложения структур и объединений.   
Постарайтесь внимательно отнестись к анализу этой программы. Основная идея здесь в том, что указатель на память, формируемый программой, может быть представлен в виде:

* 16-битного смещения;
* 32-битного смещения;
* пары из 16-битного смещения и 16-битной сегментной составляющей адреса;
* в виде пары из 32-битного смещения и 16-битного селектора.

Какие из этих указателей можно применять в конкретной ситуации, зависит от *режима адресации* (**use16** или **use32**) и режима работы микропроцессора.   
Так вот, описанный в листинге 7 шаблон объединения позволяет нам облегчить формирование и использование указателей различных типов.

|  |
| --- |
| Листинг 7 Пример использования объединения  masm  model small  stack 256  .586P  pnt struc ;структура pnt, содержащая вложенное объединение  union ;описание вложенного в структуру объединения  offs\_16 dw ?  offs\_32 dd ?  ends ;конец описания объединения  segm dw ?  ends ;конец описания структуры  .data  point union ;определение объединения, содержащего вложенную структуру  off\_16 dw ?  off\_32 dd ?  point\_16 pnt <>  point\_32 pnt <>  point ends  tst db "Строка для тестирования"  adr\_data point <> ;определение экземпляра объединения  .code  main:  mov ax,@data  mov ds,ax  mov ax,seg tst  ;записать адрес сегмента строки tst в поле структуры adr\_data  mov adr\_data.point\_16.segm,ax  ;когда понадобится, можно извлечь значение из этого поля обратно, к примеру, в регистр bx:  mov bx,adr\_data.point\_16.segm  ;формируем смещение в поле структуры adr\_data  mov ax,offset tst ;смещение строки в ax  mov adr\_data.point\_16.offs\_16,ax  ;аналогично, когда понадобится, можно извлечь значение из этого поля:  mov bx,adr\_data.point\_16.offs\_16  exit:  mov ax,4c00h  int 21h  end main |

Когда вы будете работать в защищенном режиме микропроцессора и использовать 32-разрядные адреса, то аналогичным способом можете заполнить и использовать описанное выше объединение.

**Записи**

Наша “хозяйка-программист” становится все более экономной. Она уже хочет работать с продуктами на молекулярном уровне, без любых отходов и напрасных трат.   
Подумаем, зачем тратить под некоторый программный индикатор со значением “включено-выключено” целых восемь разрядов, если вполне хватает одного? А если таких индикаторов несколько, то расход оперативной памяти может стать весьма ощутимым.   
Когда мы знакомились с логическими командами, то говорили, что их можно применять для решения подобной проблемы. Но это не совсем эффективно, так как велика вероятность ошибок, особенно при составлении битовых масок.

TASM предоставляет нам специальный тип данных, использование которого помогает решить проблему работы с битами более эффективно. Речь идет о специальном типе данных — *записях*.

***Запись*** — структурный тип данных, состоящий из фиксированного числа элементов длиной от одного до нескольких бит.   
При описании записи для каждого элемента указывается его длина в битах и, что необязательно, некоторое значение.   
Суммарный размер записи определяется суммой размеров ее полей и не может быть более **8, 16** или **32** бит.   
Если суммарный размер записи меньше указанных значений, то все поля записи “прижимаются” к младшим разрядам.

Использование записей в программе, так же, как и структур, организуется в три этапа:

1. [Задание](https://i-assembler.ru/25/Text/Data.htm" \l "ЗадатьШаблонЗаписи) *[шаблона записи](https://i-assembler.ru/25/Text/Data.htm" \l "ЗадатьШаблонЗаписи)*, то есть определение набора битовых полей, их длин и, при необходимости, инициализация полей.
2. [*Определение экземпляра записи*](https://i-assembler.ru/25/Text/Data.htm#ОпределениеЭкземпляраЗаписи). Так же, как и для структур, этот этап подразумевает инициализацию конкретной переменной типом заранее определенной с помощью шаблона записи.
3. [Организация *обращения к элементам* записи](https://i-assembler.ru/25/Text/Data.htm#ОбращенияКЭлементамЗаписи).

Компилятор TASM, кроме стандартных средств обработки записей, поддерживает также и некоторые [*дополнительные возможности*](https://i-assembler.ru/25/Text/Data.htm#ДополнительныеВозможности) их обработки.

**Описание записи**

Описание шаблона записи имеет следующий синтаксис (рис. 6):

|  |
| --- |
| имя\_записи RECORD <описание элементов> |

Здесь:   
*<описание элементов>* представляет собой последовательность описаний отдельных элементов записи согласно синтаксической диаграмме (см. рис. 6):

https://i-assembler.ru/25/Pic/Ris12_2.gif

***Рис. 6.*** *Синтаксис описания шаблона записи*

При описании шаблона память не выделяется, так как это всего лишь информация для транслятора ассемблера о структуре записи.   
Так же, как и для структур, местоположение шаблона в программе может быть любым, но при этом необходимо учитывать логику работы однопроходного транслятора.

**Определение экземпляра записи**

Для использования шаблона записи в программе необходимо определить переменную с типом данной записи, для чего применяется следующая синтаксическая конструкция (рис. 7):

https://i-assembler.ru/25/Pic/Ris12_3.gif

***Рис. 7.*** *Синтаксис описания экземпляра записи*

Анализируя эту синтаксическую диаграмму, можно сделать вывод, что инициализация элементов записи осуществляется достаточно гибко. Рассмотрим несколько вариантов инициализации.

Если инициализировать поля не требуется, то достаточно указать **?** при определении экземпляра записи:

|  |
| --- |
| ...  iotest record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00  ...  flag iotest ? |

Если вы составите и исследуете в отладчике тестовый пример с данным определением записи, то увидите, что все поля переменной типа запись *flag* обнуляются. Это происходит несмотря на то, что в определении записи заданы начальные значения полей.

Если требуется частичная инициализация элементов, то они заключаются в *угловые* (**<** и **>**) или *фигурные* (**{** и **}**) скобки.   
Различие здесь в том, что в *угловых* скобках элементы должны быть заданы в том же порядке, что и в определении записи. Если значение некоторого элемента совпадает с начальным, то его можно не указывать, но обязательно обозначить его запятой. Для последних элементов идущие подряд запятые можно опустить.   
К примеру, согласиться со значениями по умолчанию можно так:

|  |
| --- |
| iotest record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00  ...  flag iotest <> ;согласились со значением по умолчанию |

Изменить значение поля i2 можно так:

|  |
| --- |
| iotest record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00  ...  flag iotest <,10,> ; переопределили i2 |

Применяя *фигурные* скобки, также можно указать выборочную инициализацию полей, но при этом необязательно обозначать запятыми поля, со значениями по умолчанию которых мы согласны:

|  |
| --- |
| iotest record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00  ...  flag iotest {i2=10} ;переопределили i2, не обращая внимания на порядок  ;следования других компонентов записи |

**Работа с записями**

Как организовать работу с отдельными элементами записи? Обычные механизмы адресации здесь бессильны, так как они работают на уровне ячеек памяти, то есть байтов, а не отдельных битов. Здесь программисту нужно приложить некоторые усилия.   
Прежде всего для понимания проблемы нужно усвоить несколько моментов:

* Каждому имени элемента записи ассемблер присваивает числовое значение, равное количеству сдвигов вправо, которые нужно произвести для того, чтобы этот элемент оказался “прижатым” к началу ячейки. Это дает нам возможность локализовать его и работать с ним. Но для этого нужно знать длину элемента в битах.
* Сдвиг вправо производится с помощью команды сдвига **[shr](https://i-assembler.ru/25/Text/Command.htm" \l "shr)**.
* Ассемблер содержит оператор **width**, который позволяет узнать размер элемента записи в битах или полностью размер записи. Варианты применения оператора **width**:

|  |
| --- |
| width имя\_элемента\_записи ;значением оператора будет размер элемента в битах. |

|  |
| --- |
| width имя\_экземпляра\_записи  или  width имя\_типа\_записи ;значением оператора будет размер всей записи в битах. |

|  |
| --- |
| mov al,width i2  ...  mov ax,width iotest |

* Ассемблер содержит оператор **mask**, который позволяет *локализовать* биты нужного элемента записи. Эта локализация производится путем создания маски, размер которой совпадает с размером записи. В этой маске обнулены биты на всех позициях, за исключением тех, которые занимает элемент в записи.
* Сами действия по преобразованию элементов записи производятся с помощью логических команд.

Теперь у вас есть вся информация о средствах ассемблера для работы с записями.   
Вы также поняли, что непосредственно обратиться к элементу записи невозможно. Чтобы произвести обработку интересующего нас элемента, нужно сначала выделить, сдвинуть его, при необходимости, к младшим разрядам, выполнить необходимые действия и поместить его обратно на свое место в записи. Поэтому, чтобы вам не изобретать каждый раз велосипед, далее мы опишем типовые алгоритмы осуществления этих операций над элементами записи.   
Ваша задача — закодировать эти алгоритмы тем или иным способом в соответствии с требованиями задачи.

**Выделение элемента записи:**

* Поместить запись во временную память — регистр (8, 16 или 32-битный в зависимости от размера записи).
* Получить битовую маску, соответствующую элементу записи, с помощью оператора **mask**.
* Локализовать биты в регистре с помощью маски и команды [**and**](https://i-assembler.ru/25/Text/Command.htm#and).
* Сдвинуть биты элемента к младшим разрядам регистра командой [**shr**](https://i-assembler.ru/25/Text/Command.htm#shr). Число разрядов для сдвига получить с использованием имени элемента записи.

В результате этих действий элемент записи будет локализован в начале рабочего регистра и далее с ним можно производить любые действия.

**Работа с элементом записи:**

Как мы уже выяснили, с элементами записи производятся любые действия, как над обычной двоичной информацией.   
Единственное, что нужно отслеживать, — это размер битового поля. Если, к примеру, размер поля увеличится, то впоследствии может произойти случайное изменение соседних полей битов. Поэтому желательно исключить изменение размера поля.

**Помещение измененного элемента на его место в запись:**

* Используя имя элемента записи в качестве счетчика сдвигов, сдвинуть влево биты элемента записи.
* Если вы не уверены в том, что разрядность результата преобразований не превысила исходную, можно выполнить “обрезание” лишних битов, используя команду [**and**](https://i-assembler.ru/25/Text/Command.htm#and) и маску элемента.
* Подготовить исходную запись к вставке измененного элемента путем обнуления битов в записи на месте этого элемента. Это можно сделать путем наложения командой [**and**](https://i-assembler.ru/25/Text/Command.htm#and) инвертированной маски элемента записи на исходную запись.
* С помощью команды [**or**](https://i-assembler.ru/25/Text/Command.htm#or) наложить значение в регистре на исходную запись.

В качестве примера рассмотрим листинг 8, который обнуляет поле *i2* в записи *iotest*.

|  |
| --- |
| Листинг 8. Работа с полем записи  ;prg\_12\_7.asm  masm  model small  stack 256  iotest record i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00  .data  flag iotest <>  .code  main:  mov ax,@data  mov ds,ax  mov al,mask i2  shr al,i2 ;биты i2 в начале ax  and al,0fch ;обнулили i2  ;помещаем i2 на место  shl al,i2  mov bl,[flag]  xor bl,mask i2 ;сбросили i2  or bl,al ;наложили  exit:  mov ax,4c00h ;стандартный выход  int 21h  end main ;конец программы |

В заключение еще раз проанализируйте тип записи и особенности работы с ним. При этом обратите внимание на то обстоятельство, что мы нигде явно не просчитываем расположение битов. Поэтому если понадобится изменить размер элемента или его начальное значение, достаточно внести изменения в экземпляр записи или в описание ее типа; функциональную часть программы, работающую с этой записью, трогать не нужно.

**Записи: дополнительные возможности обработки**

Понимая важность для эффективного программирования такого типа данных, как запись, разработчики транслятора TASM, начиная с версии 3.0, включили в систему его команд две дополнительные команды на правах директив.   
Последнее означает, что эти команды внешне имеют формат обычных команд ассемблера, но после трансляции они приводятся к одной или нескольким машинным командам.   
Введение этих команд в язык TASM повышает наглядность работы с записями, оптимизирует код и уменьшает размер программы.   
Эти команды позволяют скрыть от программиста действия по выделению и установке отдельных полей записи (мы их обсуждали выше).

Для *установки значения некоторого поля* записи используется команда **setfield** с синтаксисом:

|  |
| --- |
| setfield имя\_элемента\_записи назначение,регистр\_источник |

Для *выборки значения некоторого поля* записи используется команда **getfield** с синтаксисом:

|  |
| --- |
| getfield имя\_элемента\_записи регистр\_назначение, источник |

Работа команды **setfield** заключается в следующем. Местоположение записи определяется операндом ***назначение***, который может представлять собой *имя регистра* или *адрес памяти*.   
Операнд ***имя\_элемента\_записи*** определяет элемент записи, с которым ведется работа (по сути, если вы были внимательны, он определяет смещение элемента в записи относительно младшего разряда). Новое значение, в которое необходимо установить указанный элемент записи, должно содержаться в операнде ***регистр\_источник***. Обрабатывая данную команду, транслятор генерирует последовательность команд, которые выполняют следующие действия:

* *сдвиг* содержимого ***регистр\_источник*** *влево* на количество разрядов, соответствующее расположению элемента в записи;
* логическую операцию **[or](https://i-assembler.ru/25/Text/Command.htm" \l "or)** над операндами ***назначение*** и ***регистр\_источник***. Результат операции помещается в операнд ***назначение***.

Важно отметить, что **setfield** не производит предварительной очистки элемента, в результате после логического сложения командой [**or**](https://i-assembler.ru/25/Text/Command.htm#or) возможно наложение старого содержимого элемента и нового устанавливаемого значения. Поэтому требуется предварительно подготовить поле в записи путем его обнуления.

Действие команды **getfield** обратно **setfield**. В качестве операнда ***источник*** может быть указан либо регистр либо адрес памяти.   
В регистр, указанный операндом ***регистр\_назначение***, помещается результат работы команды — значение элемента записи.   
Интересная особенность связана с ***регистр\_назначение***. Команда **getfield** всегда использует 16-битный регистр, даже если вы укажете в этой команде имя 8-битного регистра.

В качестве примера применения команд **setfield** и **getfield** рассмотрим листинг 9.

|  |
| --- |
| Листинг 9. Работа с полями записи  ;prg\_12\_8.asm  masm  model small  stack 256  iotest record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00  .data  flag iotest <>  .code  main:  mov ax,@data  mov ds,ax  mov al,flag  mov bl,3  setfield i5 al,bl  xor bl,bl  getfield i5 bl,al  mov bl,1  setfield i4 al,bl  setfield i5 al,bl  exit:  mov ax,4c00h ;стандартный выход  int 21h  end main ;конец программы |

Результат работы команд **setfield** и **getfield** удобнее всего изучать в отладчике.   
При установке значений полей не производится их предварительная очистка. Это сделано специально. Для такого рода операций лучше использовать некоторые универсальные механизмы, иначе велик риск внесения ошибок, которые трудно обнаружить и исправить. В качестве такого механизма можно предложить [*механизм макросредств*](https://i-assembler.ru/25/Text/Macro.htm).

В заключение хотелось бы привести еще один пример использования записей.   
Это описание регистра **eflags**. Для удобства мы разбили это описание на три части:

* *eflags\_1\_7* — младший байт **eflags/flags**;
* *eflags\_8\_15* — второй байт **eflags/flags**;
* *eflags\_h* — старшая половина **eflags**.

|  |
| --- |
| eflags\_l\_7 record  sf7:1=0,zf6:1=0,c5:1=0,af4:1=0,c3:1=0,pf2:1=0,c1:=1,cf0:1=0  eflags\_l\_15 record  c15:1=0,nt14:1=0,iopl:2=0,of11:1=0,df10:1=0,if9:1=1,tf8:1=0  eflags\_h record  c:13=0,ac18:1=0,vm17:1=0,rf16:1=0 |

Запомните это описание. Когда вы освоите работу с макрокомандами и в дальнейшей своей работе столкнетесь с необходимостью работать с регистром флагов, то у вас буквально “зачешутся” руки, чтобы написать соответствующую макрокоманду. Эта макрокоманда, если вы не забудете хорошо ее оттестировать, избавит вас от многих трудно обнаруживаемых ошибок.