Structures

A structure in assembly language is defined using the STRUCT and ENDS directives.

Inside the structure, fields are defined using the same syntax as for ordinary variables. Structures can contain virtually any number of fields:

```
0890 name STRUCT
0891 field-declarations
0892 name ENDS
```

For example, the following structure defines an employee structure with fields for ID number, last name, years of service, and an array of salary history values:

```
0900 Employee STRUCT
0901     IdNum BYTE "000000000"
0902     LastName BYTE 30 DUP(0)
0903     Years WORD 0
0904     SalaryHistory DWORD 0,0,0,0
0905 Employee ENDS
```

The first field, IdNum, is a byte-sized field initialized to the string "000000000".

The second field, LastName, is a 30-byte array of bytes initialized to all zeros.

The third field, Years, is a 2-byte word initialized to zero.

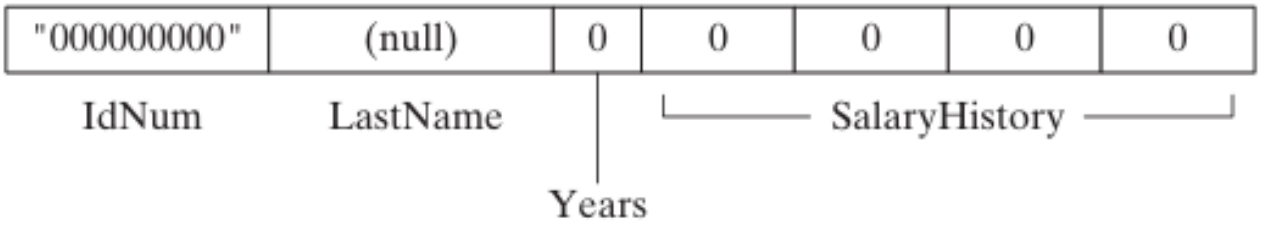
The fourth field, SalaryHistory, is a 4-byte double word initialized to all zeros.

Note: The DUP operator is used to initialize the SalaryHistory array to all zeros. The syntax 30 DUP(0) creates an array of 30 bytes, each of which is initialized to zero.

Structure Memory Layout

In memory, structures are laid out contiguously, with each field starting at the next available address.

The following diagram shows the memory layout of the Employee structure:



Memory Layout:

```
+-----------------------------------------+
| IdNum                                   | (1 byte)
+-----------------------------------------+
| LastName                                | (30 bytes)
+-----------------------------------------+
| Years                                   | (2 bytes)
+-----------------------------------------+
| SalaryHistory                           | (4 bytes)
+-----------------------------------------+
```

Using Structures

Declaring structures:

```
identifier structureType < initializer-list >
```

The **identifier** is the name of the structure variable.

The **structureType** is the name of the structure type that the variable

will be.

The **initializer-list** is a list of values that will be used to initialize the structure fields. The values in the initializer-list must be in the same order as the fields in the structure definition.

If you do not specify an initializer-list, the structure fields will be initialized to their default values.

Example:

```
Employee worker <>
```

This code declares a structure variable named worker of type Employee. The <> (angle brackets) indicate that the structure fields should be initialized to their default values.

Referencing Structure Variables:

To reference a structure variable, you simply use its name. For example, the following code references the Years field of the worker structure variable:

Once a structure is defined, you can declare variables of that type using the same syntax as for ordinary variables. For example, the following code declares two employee variables:

```
mov dx, worker.Years
```

Referencing Structure Members:

To reference a structure member, you use the following syntax:

```
structureVariable.memberName
```

The structureVariable is the name of the structure variable. The memberName is the name of the structure member.

Example:

```
mov dx, worker.SalaryHistory
```

This code references the SalaryHistory field of the worker structure variable.

Using the OFFSET Operator

The OFFSET operator can be used to obtain the address of a structure member. The syntax for the OFFSET operator is as follows:

```
OFFSET structureVariable.memberName
```

The structureVariable is the name of the structure variable. The memberName is the name of the structure member.

Example:

```
mov edx, OFFSET worker.LastName
```

This code obtains the address of the LastName field of the worker structure variable.

==========

Declaring another structure:

```
0926 Employee1 Employee
0927 Employee2 Employee
```

You can then access the fields of these variables using the dot operator (.). For example, the following code sets the IdNum field of Employee1 to 123456789:

```
mov Employee1.IdNum, 123456789
```

You can also access the fields of a structure using the displacement operator ([]). For example, the following code sets the Years field of Employee2 to 5:

```
mov Employee2[2], 5
```

Note: The displacement operator is calculated relative to the beginning of the structure. In the previous example, the displacement of the Years field is 2, because it is the second field in the structure.

Aligning Structure Fields

To achieve the best memory I/O performance, structure members should be aligned to addresses matching their data types.

This means that a byte member should be aligned on a byte boundary, a word member should be aligned on a word boundary, and so on.

If structure members are not aligned correctly, the CPU will require more time to access them.

The following table lists the alignments used by the Microsoft C and C++ compilers and by Win32 API functions:

Member Type	Alignment
BYTE, SBYTE	8-bit (byte) boundary
WORD, SWORD	16-bit (word) boundary
DWORD, SDWORD	32-bit (doubleword) boundary
QWORD	64-bit (quadword) boundary
REAL4	32-bit (doubleword) boundary
REAL8	64-bit (quadword) boundary
Structure	Largest alignment requirement of any member
Union	Alignment requirement of the first member

The key points are:

• Basic data types like BYTE, WORD, DWORD etc have natural alignment to their size boundary.

• 32-bit and 64-bit floating point values align to doubleword and quadword respectively.

• Structures align to the largest alignment needed by any member.

• Unions align to the first member.

Properly aligning data prevents performance issues from unaligned memory accesses. The assembler takes care of alignment automatically based on the data type.

To align structure fields in assembly language, you can use the ALIGN directive.

The ALIGN directive sets the address alignment of the next field or variable.

For example, the following code aligns the myVar variable to a doubleword boundary:

```
.data
    ALIGN DWORD
    myVar DWORD ?
```

You can also use the ALIGN directive to align structure fields.

For example, the following code correctly defines the Employee structure, using ALIGN to put Years on a word boundary and SalaryHistory on a doubleword boundary:

```
Employee STRUCT
    IdNum BYTE "000000000" ; 9
    LastName BYTE 30 DUP(0) ; 30
    ALIGN WORD ; 1 byte added
    Years WORD 0 ; 2
    ALIGN DWORD ; 2 bytes added
    SalaryHistory DWORD 0,0,0,0 ; 16
Employee ENDS ; 60 total
```

Why is Aligning Structure Fields Important?

Aligning structure fields is important for performance because the CPU can access aligned data more efficiently.

When structure fields are not aligned, the CPU may have to perform additional instructions to access the data.

This can lead to a significant performance decrease, especially in applications that manipulate large amounts of data.