REPEAT Blocks

MASM provides several looping directives for generating repeated blocks of statements: WHILE, REPEAT, FOR, and FORC.

These directives operate at assembly time and use constant values for loop conditions and counters.

WHILE Directive:

The WHILE directive repeats a block of statements as long as a specific constant expression remains true. It has the following syntax:

```
1987 WHILE constExpression
1988   statements
1989 ENDM
```

For example, you can use the WHILE directive to generate Fibonacci numbers within a specific range, like so:

```
1992 .data
1993     val1 = 1
1994     val2 = 1
1995     DWORD val1 ; First two values
1996     DWORD val2
1997     val3 = val1 + val2
1998
1999 WHILE val3 LT 0F0000000h
2000   DWORD val3
2001   val1 = val2
2002   val2 = val3
2003   val3 = val1 + val2
2004 ENDM
```

This code generates Fibonacci numbers and stores them as assembly-time constants until the value exceeds 0F0000000h.

REPEAT Directive:

The REPEAT directive repeats a statement block a fixed number of times at assembly time, based on an unsigned constant integer expression.

```
2008 REPEAT constExpression
2009   statements
2010 ENDM
```

It's used when you need to repeat a block of code a predetermined number of times, similar to the DUP directive.

REPEAT Directive Example: Creating an Array

In this example, we use the REPEAT directive to create an array of WeatherReadings. Each WeatherReadings struct contains a location string and arrays for rainfall and humidity readings. The loop repeats for a total of WEEKS_PER_YEAR times:

```
2050 WEEKS_PER_YEAR = 52
2051
2052 WeatherReadings STRUCT
2053     location BYTE 50 DUP(0)
2054     REPEAT WEEKS_PER_YEAR
2055         LOCAL rainfall, humidity
2056         rainfall DWORD ?
2057         humidity DWORD ?
2058     ENDM
2059 WeatherReadings ENDS
```

This code defines a structured array for recording weather readings over the course of a year.

FOR Directive:

The FOR directive repeats a statement block by iterating over a comma-delimited list of symbols. Each symbol in the list represents one iteration of the loop.

```
2015 FOR parameter, <arg1, arg2, arg3, ...>
2016   statements
2017 ENDM
```

It's useful when you want to perform a set of operations for each item in a list of symbols.

FOR Directive Example: Student Enrollment

In this example, the FOR directive is used to create multiple SEMESTER objects for student enrollment in different semesters. The loop iterates over a list of semester names and generates corresponding SEMESTER objects:

```
2065 .data
2066 FOR semName, <Fall2013, Spring2014, Summer2014, Fall2014>
2067     semName SEMESTER <>
2068 ENDM
```

This code generates SEMESTER objects with different names for each semester.

FORC Directive:

The FORC directive repeats a statement block by iterating over a string of characters. Each character in the string represents one iteration of the loop.

```
2020 FORC parameter, <string>
2021   statements
2022 ENDM
```

It's handy when you need to process a block of code for each character in a string.

Student Enrollment:

You can use the FOR directive to create multiple SEMESTER objects, each with a different name from a list of symbols. This can be useful for managing student enrollments over multiple semesters.

Character Lookup Table:

The FORC directive can be used to generate a character lookup table. In this example, a table of non-alphabetic characters is created by iterating through a string of special characters. These looping directives offer flexibility and structure for generating repetitive code in assembly language programs, making it easier to manage and control complex operations.

FORC Directive Example: Character Lookup Table

In this example, the FORC directive is used to create a character lookup table for non-alphabetic characters. Each character in the string is processed to generate a corresponding entry in the lookup table:

```
2070 Delimiters LABEL BYTE
2071 FORC code, <@#$%^&*!<!>>
2072     BYTE "&code"
2073 ENDM
```

This code generates a lookup table containing the ASCII values and corresponding characters for various special symbols.

In this example, we create a linked list data structure using the

ListNode structure, which contains a data area (NodeData) and a pointer to the next node (NextPtr).

The program defines and populates multiple instances of ListNode objects within a loop to create a linked list.

Here's the revised and expanded code with added explanations:

```
2077 INCLUDE Irvine32.inc
2078
2079 ; Define the ListNode structure
2080 ListNode STRUCT
2081     NodeData DWORD ?
2082     NextPtr DWORD ?
2083 ListNode ENDS
2084
2085 TotalNodeCount = 15
2086 NULL = 0
2087 Counter = 0
2088
2089 .data
2090 LinkedList LABEL PTR ListNode
2091
2092 ; Use the REPEAT directive to create a linked list
2093 REPEAT TotalNodeCount
2094     Counter = Counter + 1
2095     ; Create a new ListNode with data and link to the next node
2096     ListNode <Counter, ($ + Counter * SIZEOF ListNode)>
2097 ENDM
2098
2099 ; Create a tail node to mark the end of the list
2100 ListNode <0, 0>
2101
2102 .code
2103 main PROC
2104     mov esi, OFFSET LinkedList  ; Initialize the pointer to the start of the list
```

```
2105
2106 NextNode:
2107     ; Check for the tail node (end of the list).
2108     mov eax, (ListNode PTR [esi]).NextPtr
2109     cmp eax, NULL
2110     je quit  ; If NextPtr is NULL, exit the loop
2111
2112     ; Display the node data.
2113     mov eax, (ListNode PTR [esi]).NodeData
2114     call WriteDec  ; Display the integer value
2115     call Crlf      ; Move to the next line for the next value
2116
2117     ; Get the pointer to the next node.
2118     mov esi, (ListNode PTR [esi]).NextPtr
2119     jmp NextNode
2120
2121 quit:
2122     exit
2123
2124 main ENDP
2125
2126 END main
```

Program to illustrate what we have learnt above:

```
2132 INCLUDE Irvine32.inc
2133
2134 ; Define a macro that generates bytes based on a list of values
2135 mGenerateBytes MACRO values
2136     LOCAL L1
2137     FOR val, <values>
2138         BYTE val
2139     ENDM
2140 ENDM
2141
2142 .data
2143 byteArray BYTE 0, 0, 0, 0
2144
2145 .code
2146 main PROC
2147     ; Question 7 - Macro Expansion
2148     ; a
2149     mGenerateBytes <100, 20, 30>
2150     mov eax, OFFSET byteArray
2151     call DisplayByteArray
2152
2153     ; b
2154     mGenerateBytes <AL, 20>
2155     mov eax, OFFSET byteArray
2156     call DisplayByteArray
2157
```

```
2158     ; c
2159     byteVal = 42
2160     countVal = 5
2161     mGenerateBytes <byteVal, countVal>
2162     mov eax, OFFSET byteArray
2163     call DisplayByteArray
2164
2165     ; Question 8 - Linked List Scenario
2166     TotalNodeCount = 15
2167     NULL = 0
2168     Counter = 0
2169     LinkedList LABEL PTR ListNode
2170
2171     REPEAT TotalNodeCount
2172         Counter = Counter + 1
2173         ListNode <Counter, ($ + SIZEOF ListNode)>
2174     ENDM
2175
2176     ; Traverse and display the linked list
2177     mov esi, OFFSET LinkedList
2178
```

```
2179 NextNode:
2180     mov eax, (ListNode PTR [esi]).NextPtr
2181     cmp eax, NULL
2182     je quit
2183
2184     mov eax, (ListNode PTR [esi]).NodeData
2185     call WriteDec
2186     call Crlf
2187
2188     mov esi, (ListNode PTR [esi]).NextPtr
2189     jmp NextNode
2190
2191 quit:
2192     exit
2193
2194 main ENDP
2195
2196 ; Helper function to display the contents of the byteArray
2197 DisplayByteArray PROC
2198     mov ecx, LENGTHOF byteArray
2199     mov esi, 0
2200     mov edx, OFFSET byteArray
2201
2202 DisplayLoop:
2203     mov al, [edx + esi]
2204     call WriteHex
2205     call Crlf
2206     inc esi
2207     loop DisplayLoop
2208     ret
2209 DisplayByteArray ENDP
```

2210 It's gonna be **END main** you know...👊😄