Tutorial: x86_64 Assembly Language Programming

by Jerry McIntosh

INTRODUCTION

In this tutorial I hope to share some insight into Assembly language programming by working through developing a list library. That is, a glorified array with some structured bells and whistles. I also hope that this tutorial will be useful as a guide for how to build a shared library in Assembly and C. There will definitely be some insight given into how the two languages work together. And maybe some hint as to the nature of Assembly language. I learn best by doing. Therefore, my hope is that by working through developing a shared library you might gain some insight into x86_64 Assembly language, function calls, parameter passing, and return values. There's some basic math involved to compute offsets and addresses. You'll write some simple iterator functions and see them implemented in test programs written in C. All of the functions in the shared library are written in Assembly language, but have corresponding C declarations that make it possible for the C programs to use them. I provide the output generated by each test program. But, as you progress through this tutorial I suggest you look at the source code of the test programs, and modify them to learn more.

First, here is a list of the requirements for this project.

- Linux OS
- Programming languages: C and Assembly
- Netwide Assembler (NASM) and the GCC compiler
- your favorite text editor
- and working at the command line

We will be writing a shared library in Assembly and C. We will use C to demo our project as we add the bells and whistles that make the list library what it is. Using C will also provide the opportunity to see how the two languages interact.

Assembly language relies heavily on registers. For this project we need access to 64-bit and 32-bit registers. However, there are a number of registers at our disposal: general-purpose registers; the eflags register; and the instruction pointer register.

The 64-bit general-purpose registers are: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, and R8 thru R15.

The 32-bit general-purpose registers are: EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI, and R8D thru R15D. These registers are located in the lower half (32-bits) of the corresponding 64-bit register. For instance, EAX is the lower 32-bits of RAX, and R8D is the lower 32-bits of R8.

There are also 16-bit and 8-bit registers that form the lower half of the 32-bit and 16-bit registers accordingly. For more information on the general-purpose registers see section "BASIC PROGRAM EXECUTION REGISTERS" in chapter 3 of the "Intel 64 and IA-32 Architectures Software Developer's Manual."

On Linux systems the first 6 parameters (1-6) passed to a function are passed (as listed) in registers: RDI, RSI, RDX, RCX, R8, R9. The remaining parameters (7 and above) are passed on the stack in reverse order. That is the last parameter is pushed on the stack first and the 7th parameter is pushed last.

On Linux systems, functions have a responsibility to preserve the contents of certain registers referred to as callee-saved registers. Here are the callee-saved registers: RBX, RBP, RSP, R12, R13, R14, R15.

We will use the ELF (Executable and Linkable Format) format when generating the object files. The ELF format is used by Linux. You will create the makefile for the shared library, but all other makefiles will be provided. Other than providing the commads necessary to execute the makefiles, I won't be covering the make utility.

NOTE: The Assembly syntax we will use is the Intel syntax.

THE FILE: UTIL.ZIP

If you haven't already downloaded file `util.zip`, do so now. Once downloaded copy the file into your project folder. Next, at the command line change directory to your project folder and unzip the file. Once unzipped the following folder structure should appear.

```
util/
  + list_add/
  + list_begin/
  + list_delete/
  + list_find/
  + list_init/
  + list_sort/
  + util/
```

NOTE: In the `util/util/` folder you will see file `.gitignore`. As the name suggests, you can ignore the file. GitHub tracks files not folders, so, an empty folder is automatically scrubbed from a repository. You can either ignore the file or delete the file with the following command (while in the `util/util/` folder).

```
rm -f ./.gitignore
```

FUNCTION: MEMMOVE64

In the `util/util/` folder create the file `memmove64.asm` via your text editor, or at the command line with the following command.

```
touch memmove64.asm
```

Now fire up your text editor, open file `memmove64.asm`, and enter the code below.

```
; file:   memmove64.asm
; brief:  move quadword (8-byte) chucks from source to destination
```

```
%ifndef MEMMOVE64_ASM
%define MEMMOVE64_ASM  1
;
QW_SIZE     EQU     8
;
%endif
```

MEMMOVE64_ASM is a marco that is undefined when the NASM assembler first processes file memmove64.asm. The NASM directive %ifndef (which means: if not defined) controls the processing of the code between the directives %ifndef and %endif. These directives create what is know as a conditional-block of code. If the marco MEMMOVE64_ASM is undefined then the NASM assembler will process the code between %ifndef and the corresponding %endif. Within the conditional-block is a %define directive that defines macro MEMMOVE64_ASM. Any attempt by the assembler to process the conditional-block after MEMMOVE64_ASM is defined will fail. Thus, the contents of the conditional-block can only be processed by the assembler one time.

The only code in our conditional-block is what is called a pseudo-instruction—the EQU command. The EQU command defines a symbol as equivalent to a constant value. In this case symbol QW_SIZE (which stands for: quadword size) is equated to the constant value 8. A quadword is 8-bytes in size.

NASM uses semicolons to indicate the beginning of a comment. A semicolon can begin at any position on a line. The semicolon and everything to the right of the semicolon is considered a comment and ignored by the NASM assembler.

The comment-block below defines our memmove64 function. Enter the comment-block before the line with the %endif directive.

```
;-------------------------------------------------------------------------------
; C definition:
;
;   void * memmove64 (void *dst, void const *src, size_t size)
;
; param:
;
;   rdi = dst
;   rsi = src
;   rdx = size
;
; return:
;
;   rax = dst
;
; WARNING: this routine does not handle the overlapping source-destination
;          senario.
;-------------------------------------------------------------------------------
;
%endif
```

The memmove64 function is the workhorse of the list library. The comment-block provides: a C language declaration of function memmove64; a listing of the registers used to pass each parameter; the value memmove64 will return (the address of the destination), and a warning about the constraints of the function.

Below we define the section of the object file our function will reside in—the .text section. We declare the symbol memmove64 as global and of type function. The global directive works in conjunction with the extern directive that will be explained later. Should a shared library function be called during program execution, the dynamic linker will resolve the actual address of that function in a shared library and store that address in the Global Offset Table (GOT) for future reference. The Procedure Linkage Table (PLT) is used when a call is made from a shared library to a function outside that library. The PLT will access the address in the GOT once the dynamic linker as resolved the actual address. Next, the symbol memmove64 is defined as a label, which is equivalent to the address of the function used during program execution.

Enter the following code-block below before the line with the %endif directive.

```
section .text
      global memmove64:function
memmove64:
      push      rdi            ; push destination address on stack
; quadword count = size / QW_SIZE
      mov       rax, rdx       ; copy dividend (size) to rax
      xor       rdx, rdx       ; zero out rdx
      mov       r11, QW_SIZE   ; copy divisor to r11
      div       r11            ; rdx:rax == (byte count):(quadword count)
      mov       rcx, rax       ; copy quadword count to rcx
      cld                      ; increment index registers rsi and rdi
      rep movsq                ; repeat quadword move operation
      mov       rcx, rdx       ; copy byte count to rcx
      rep movsb                ; repeat byte move operation
      pop       rax            ; pop destination address off stack
      ret
%endif
```

(Some functions will have a prologue and epilogue section. Such functions either have parameters that are passed on the stack (parameters seven and above) or have local variables. To access the parameters and/or local variables the rbp register will be used).

First, the contents of register rdi (the destination address for the move operation) is stored on the stack with the push instruction. The destination address is the return value of the function, and will be popped off the stack into register rax just before the function returns.

Second, the number of quadwords (the number of 8-byte blocks) is computed by dividing the value represented by QW_SIZE into the value passed in parameter size. In this case, the div instruction performs an unsigned division operation on a dividend in registers rdx:rax by a 64-bit divisor which can reside in memory or another register. (More on the div instruction can be found in the Intel 64 and IA32 Architecture Software Developer Manual). I use register `r11` to hold the divisor. The `div` instruction will place the quotient (quadword count) in register `rax` and the remainder (byte count) in register `rdx`.

Quotient = size / 8 = number times to call instruction `movsq` (move quadwords)
Remainder = size mod 8 = number of times to call instruction `movsb` (move bytes)

Third, the number of quadwords is copied from `rax` to `rcx` in order to be used by the repeat instruction `rep`. The `movsq` instruction moves a quadword (8-byte) string from the source address held in `rsi` to the destination address held in `rdi`. The instruction combo `rep movsq` moves 8-bytes per iteration, decrementing register `rcx` by one each iteration until the value in `rcx` reaches zero. Each iteration increments by eight the source address in register `rsi` and the destination address in register `rdi`.

Forth, the `direction-flag` is cleared (`DF` bit is 0) by the `cld` instruction. Clearing the `direction-flag` means setting the corresponding bit in the `eflags` register to zero. When the `direction-flag` is cleared the index registers `rsi` and `rdi` are incremented by string operations. When the `direction-flag` is set (`DF` bit is 1) the index registers are decremented by string operations.

Now, when the number of quadwords was calculated the remainder was put in register `rdx` by the `div` instruction. The remainder represents the number of bytes (less than 8) that remain to be moved. So, we copy the remainder to register `rcx` to prepare for the `rep movsb` instruction combo which will move the remaining bytes (if any) from the source address in `rsi` to the destination address in `rdi`, decrementing register `rcx` and incrementing registers `rsi` and `rdi` accordingly. (Notice that the only register to be modified between the two different move cycles was `rcx`. The source and destination registers both hold the correct addresses for the next iteration (if any) of 1-byte move operations).

The last operation before function `memmove64` returns is to set the return value in register `rax`. The return value is the destination address (the parameter passed in register `rdi`) that was pushed onto the stack prior to any operations that would have modified `rdi`. That address is popped off the stack into `rax` as the return value of function `memmove64`.

See Appendix B for the full source code of function `memmove64`.

LIST STRUCTURE IN ASSEMBLY

Now that we have our workhorse function we can build the structure and functions of our list library. Next, create a file `list.inc` in the `util/util/` via the command line or with your text editor.

Enter the code below into file `list.inc`.

```
; file:   list.inc
; brief:  list library structure definition
%ifndef list_INC
%define list_INC  1
;------------------------------------------------------------------------------
```

```
;
%endif
```

We need a list structure definition in Assembly. Enter the following structure definition before the line with `%endif`.

```
struc list
    .o_size         resq      1     ; size (in bytes) of one object
    .total          resq      1     ; maximum # of objects this list can hold
    .count          resq      1     ; number of objects in list
    .index          resq      1     ; iterator index
    .buffer         resq      1     ; pointer to list buffer
endstruc
;
listSize      EQU     list_size
;
%endif
```

NASM provides the macros `struc` and `endstruc` to define a structure data type. The name of our structure is `list`. The structure is terminated by the `endstruc` macro. Each field in our list library uses the RESB family of psuedo-instructions in general and the RESQ (which means *reserve quadword* —equivalent to 64-bits or 8-bytes) in particular. As we work through coding the list library functions the purpose of each field will become clear. However, each field has a comment to the right that provides some insight into its purpose. All the fields are 64-bit unsigned integers with the exception of `.buffer` which is a 64-bit address (which is also a 64-bit unsigned integer).

To determine the size of a structure with NASM we append `_size` to our structure name `list` as in `list_size`. In this case, `struct list` is 40-bytes in size, so `list_size` equates to the integer value 40.

I define the symbol `listSize` as a constant equivalent to the value of `list_size` by using the `EQU` command.

LIST STRUCTURE IN C

We will need to define our list library in the C language as well. So, in the folder `util/util/` create the following files: `util.h` and `util.c`. Then open both files in your text editor. We will start with the C header file `util.h` by defining our list structure. Enter the following in file `util.h`.

```
#ifndef UTIL_H
#define UTIL_H

#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>

typedef struct list list_t;
```

```
struct list {
  size_t      o_size;
  size_t      total;
  size_t      count;
  void *      index;
  void *      buffer;
};
```

This certainly seems clunky. Why two definitions for the same structure? The answer is simply that neither language understands the definition of the other. And each language needs a structure definition to work with. Furthermore, we will be using C to demo our Assembly functions, so it will be our C code that will instantiate the list structure and free the same.

We need `<stdint.h>` in order to use the C library definition for data type: `size_t`.

We need `<stdlib.h>` in order to use the C library memory allocation and de-allocation functions: `calloc` and `free`.

`size_t` on my computer is a 64-bit unsigned integer.

We use the same structure name and field names in C as in the Assembly code.

The keyword `typedef` is used to create an alias for a data type. In this case, `list_t` is another name for `struct list`.

In file `util.h` enter the following after the `list` structure definition.

```
};

#define list_alloc() (calloc(1, sizeof(list_t)))
#define list_free(P) (free(P))
```

The two defines above provide straight forward functions to allocate and free a `list` structure. For example, `list_alloc` we be expanded to `calloc(1, sizeof(struct list)` by the C pre-processor.

We will also need some C function declarations for our list library functions. To begin with we need a function that will initialize a list structure, and a function that will de-initialize the same. The initialization function needs some parameters, one of which is the address of a list structure and the other is the size (in bytes) of the objects we want to store in that structure.

Next, in file `util.h`, enter the following C function declarations below the allocation/de-allocation defines. Then enter the `#endif` directive to terminate the conditional-block.

```
#define list_free(P) (free(P))
```

```
int list_init (list_t *, size_t const);

void list_term (list_t *);

#endif
```

Next, in file `util.c` enter the following.

```
#include "util.h"
```

(I recommend leaving a blank line at the bottom of your util.c file. I might be wrong, but I seem to remember that being a good thing. Though it might not matter anymore).

LIST IMPLEMENTATION

At this point we need some Assembly code to go with these function declarations. In the folder `util/util/` create the file `list.asm`. Then, enter the following Assembly code into file `list.asm`.

```
; file:   list.asm
; brief:  list implementation
%ifndef LIST_ASM
%define LIST_ASM
;-------------------------------------------------------------------------------
extern bsearch
extern bzero
extern calloc
extern free
extern qsort
extern memmove64
;
%endif
```

As before we have a conditional-block of Assembly code. What is new is the `EXTERN` directive which declares a symbol as undefined in the module being processed by the assembler. However, the symbol is assumed to be defined and accessible elsewhere. In this case, `bsearch`, `bzero`, `calloc`, `free`, and `qsort` are C library (LIBC) functions, and `memmove64` is the function we wrote earlier. As for the LIBC functions, I will not cover them here. I recommend using the Linux man pages to learn more about them.

We need some constants to make our code more readable, so, enter the following into `list.asm` before the line with `%endif`.

```
;
LIST_COUNT        EQU     32
ALIGN_SIZE        EQU     16
ALIGN_WITH        EQU     (ALIGN_SIZE - 1)
ALIGN_MASK        EQU     ~(ALIGN_WITH)
```

```
;
%endif
```

The constant `LIST_COUNT` defines the number of slots available in our list library once it has been initialized by the function `list_init`. `ALIGN_SIZE`, `ALIGN_WITH` and `ALIGN_MASK` will be used to adjust our buffer size to an 16-byte boundary, as well as, adjust the stack when calling C callback and LIBC functions.

The ~ operator generates the bitwise negation (ones compliment) of its operand.

`(ALIGN_WITH)` is `(16 - 1)` or `15`. The 64-bit bitwise negation of `15` is:

`11111111 11111111 11111111 11111111 11111111 11111111 11111111 11110000.`

This value will be used to adjust our buffer size (and the stack) to a 16-byte boundary.

We will need a consistent way to align the stack to a 16-byte boundary and then call C functions. Enter the following code into `list.asm` before the line `%endif`.

```
;
%macro ALIGN_STACK_AND_CALL 2-4
      mov     %1, rsp                 ; backup stack pointer (rsp)
      and     rsp, QWORD ALIGN_MASK ; align stack pointer (rsp) to
                                      ; 16-byte boundary
      call    %2 %3 %4                ; call C function
      mov     rsp, %1                 ; restore stack pointer (rsp)
%endmacro
;
%endif
```

The macro `ALIGN_STACK_AND_CALL` simplifies calls to some LIBC and C callback functions. The macro takes anywhere from two to four parameters as indicated by “`2-4`” after the macro name. We will pass two parameters for our C callback functions, since, the address of the callback is already known. But, for LIBC function calls we will pass four parameters. The macro stores the value of the Stack Point (register `rsp`) in a callee-saved register of our choice. Then, aligns the Stack Pointer down to a 16-byte boundary. Next the call is made to the function. Upon, return from the function the Stack Pointer is restored.

NOTE: We really could do without the `ALIGN_STACK_AND_CALL` macro for this tutorial. We won’t be dealing with single or double-precision values. However, if we were the macro would be necessary, for instance, the LIBC function `printf` requires the stack be aligned to a 16-byte boundary when passing single and double precision values. This is due to the use of Advanced Vector Extensions (AVX) instructions which we won’t be covering in this tutorial.

Our functions will need access to the Assembly definition of our list structure. Enter the following before the `%endif` directive.

```
;
%include “list.inc”
;
%endif
```

Now we declare a text section where the Assembly functions will reside. Enter the following before the `%endif` directive.

```
section .text
;
%endif
```

FUNCTION: LIST_INIT

The first Assembly function we will code is `list_init`. Enter the comment-block below into `list.asm` starting before the line with the `%endif`.

```
;
;-------------------------------------------------------------------------------
; C declaration:
;
;   int list_init (list_t *list, size_t const o_size);
;
; param:
;
;   rdi = list
;   rsi = o_size
;
; return:
;
;   eax = 0 (success) | -1 (failure)
;
; stack:
;
;   [rbp - 8]    = rdi (list)
;-------------------------------------------------------------------------------
;
%endif
```

The comments provide some insight into what we will be coding. For instance, the C declaration of function `list_init` is reproduced here. The parameters passed to `list_init` are passed via registers `rdi` and `rsi`. The `list_init` function returns a 32-bit integer result of either 0 for success or -1 for failure. And the value in register `rdi` (the address of the list structure) is stored on the stack. As you remember, parameters are passed to C functions on Linux in left-to-right order:

- parameters 1-6 are passed in registers rdi, rsi, rdx, rcx, r8, r9 respectively.
- parameters above 6 are passed on the stack in reverse order (and can be popped off the stack in order).

Enter the code below into `list.asm` starting before the line with `%endif`.

```
      global list_init:function
list_init:
; prologue
      push      rbp
      mov       rbp, rsp
      sub       rsp, 8
;
; more code goes here
;
.epilogue:
      mov       rsp, rbp
      pop       rbp
      ret
%endif
```

The `global` directive is the partner of the `extern` directive. Whereas, `extern` assumes the symbol is declared elsewhere (in a seperate module), the `global` directive provides that declaration. The symbol `list_init` is given type `function` by using an extension of the Executable and Linkable Format (ELF). Thus, the symbol `list_init` refers to a `function`. On the next line, the symbol is defined as a label which will become the offset to the beginning of function `list_init`. After that comes what I call the prologue of the function (some list library functions will have a prologue and an epilogue).

Register `rbp` is known as the Base (or Frame) Pointer Register and points to the base of the stack frame of the current function. However, as far as parameters and local variables are concerned the frame pointer `rbp` points to the center of the current frame. The `rbp` register is callee-saved meaning the called function must save the contents of the register before modifying it. Hence, the `push rbp` instruction in the prologue. Next, the instruction `mov rbp, rsp` copies the value of the stack pointer `rsp` into the base pointer `rbp`. Then register `rsp` is adjusted to make room for local variables (as indicated in the comments above). Hence, register rbp points to the end of the parameter portion of the frame and the beginning of the local variable portion of the frame.

The epilogue of the function reverses the affects of the prologue by restoring the stack pointer and base (or frame) pointer values of the calling function. The `ret` instruction updates the instruction pointer `rip`, thereby, continuing execution at the instruction following the call to `list_init` in the caller.

Enter the following code after the `sub` instruction.

```
      sub rsp, 8
; store rdi (list) on stack
      mov       QWORD [rbp - 8], rdi
; list->o_size = o_size
      mov       QWORD [rdi + list.o_size], rsi
```

The first thing after the prologue is to store the address of the list structure on the stack. `QWORD` is a size specifier though the assembler will know that `rdi` is a 64-bit register and generate the appropriate code. However, the keyword `STRICT` can be used to force the size. Next the object size is copied into the corresponding member (`o_size`) of the list structure. To be clear, the object size is the size of the objects that will be stored in the list. The object size is used to calculate the list buffer size and is used by almost all the list library functions to work with objects in the list. The list buffer size is then adjusted to an 16-byte boundary. The adjustment is more an exercise in itself than something we need to worry about.

Enter the following code after the `mov` instruction.

```
      mov       QWORD [rdi + list.o_size], rsi
; buffer_size = o_size * LIST_COUNT
      mov       rax, rsi
      mov       rcx, QWORD LIST_COUNT
      mul       rcx
; buffer_size = (buffer_size + ALIGN_WITH) & ALIGN_MASK
      add       rax, QWORD ALIGN_WITH
      and       rax, QWORD ALIGN_MASK
```

The size of objects to be stored in the list is passed to `list_init` via register `rsi`. We will use that value to calculate the initial size of the list buffer. The first `mov` instruction copies the object size from register `rsi` to register `rax`. Then, `LIST_COUNT` is copied to register `rcx`. Next, the object size in register `rax` is multiplied by the value `LIST_COUNT` in register `rcx` and the result stored in the register combo `rdx:rax` by the `mul` instruction. (NOTE: I assume register `rdx` to be zero after the `mul` instruction). Then, the buffer size is adjusted to an 16-byte boundary by adding the value of `ALIGN_WITH` to the buffer size in register `rax` (the `add` instruction stores the result in the destination register which is the first of the two operands). Finally, the value in `rax` is aligned to an 16-byte boundary via the bitwise `and` instruction against the alignment mask `ALIGN_MASK`.

Enter the following code after the `and` instruction.

```
      and       rax, QWORD ALIGN_MASK
; if ((list->buffer = calloc(1, buffer_size)) == NULL) return -1
      mov       rdi, 1
      mov       rsi, rax
      call      calloc wrt ..plt
      mov       rdi, QWORD [rbp - 8]
      mov       QWORD [rdi + list.buffer], rax
      test      rax, rax
      jnz       .continue
      mov       eax, -1   ; return -1 (failure)
      jmp       .epilogue
.continue:
```

Now we will allocate the list buffer on the heap. To allocate the buffer we use the C library function `calloc`. The `calloc` function takes two arguments, the first is the number of objects and the second is the size of each object. With these values `calloc` calculates the size of the buffer to allocate. In our case, the size has already been determined. So, we pass `1` as the first argument in register `rdi`, and the calculated buffer size as the second argument in register `rsi`. Then a call is made to `calloc` with the result being returned in register `rax`. Next, the value returned from `calloc` is stored in the `buffer` member of the list structure. In order to access the list structure, we must restore it's address from the stack to register `rdi`. We do this for two reasons: first, register `rdi` held the first parameter of function `calloc`; and second, register `rdi` is not a callee saved register. After storing the address of the buffer we check to see if the `calloc` function was successful. To do that we call the `test` instruction on register `rax` and check to determine if the `zero-flag` is *not* set with the `jnz` instruction. If the `zero-flag` was set by the `test` instruction then the `calloc` function returned a `NULL` pointer indicating failure, so, execution moves to the next instruction where register `eax` is set to `-1`, followed by a jump to the first instruction after label `.epilogue`, which leads to the termination of function `list_init`. Otherwise, function `calloc` returned a valid memory address, and we jump to the next instruction after the label `.continue`.

Enter the following code after label `.continue`.

```
.continue
; list->total = LIST_COUNT
      mov       rax, LIST_COUNT
      mov       QWORD [rdi + list.total], rax
; list->count = 0
      xor       rax, rax
      mov       QWORD [rdi + list.count], rax
; return 0
;     xor       eax, eax
```

At this point we have successfully allocated a buffer on the heap and can continue initializing the list structure. We set the `total` member of the list structure to the value of `LIST_COUNT` to indicate how many objects the list buffer can hold. To set the `total` member we use register `rdi` (which holds the address of the list structure) and add the offset to the member `total`. We use the structure we defined in file `list.inc` to provide the offset. The list structure amounts to a group of offsets we can use to access members in an actual list structure. Next we use the `xor` instruction to zero out the `rax` register. Then we use `rax` to set member `count` to zero. The last `xor` is commented out since `rax` has already been set to zero, so, we already have our return value.

See Appendix C for the full source code of function `list_init`.

FUNCTION: LIST_TERM

We need a function that will prepare a list structure to be deallocated. I call the function a list structure terminator. The list structure terminator is not that complicated, we just need to free the list buffer memory and zero out the list structure members. `list_term` will be that function.

Enter the comment-block below into `list.asm` before the `%endif`.

```
;
;-------------------------------------------------------------------------------
; C definition:
;
;   void list_term (list_t *list)
;
; param:
;
;   rdi = list
;
; stack:
;
;   [rbp - 8] = rdi (list}
;-------------------------------------------------------------------------------
;
%endif
```

Unlike function `list_init` the `list_term` function has no return value. Hence, the `void` keyword before the function name in the C definition.

Enter the code below into `list.asm` before `%endif`.

```
      global list_term:function
list_term:
; prologue
      push      rbp
      mov       rbp, rsp
      sub       rsp, 8
; store rdi (list) on stack
      mov       QWORD [rbp - 8], rdi
; free list buffer memory on heap
      mov       rdi, QWORD [rdi + list.buffer]
      call      free wrt ..plt
; zero out list structure
      mov       rdi, QWORD [rbp - 8]
      mov       rsi, QWORD listSize
      call      bzero wrt ..plt
; epilogue
      mov       rsp, rbp
      pop       rbp
      ret
%endif
```

`list_term` begins the same as the `list_init` function. The symbol `list_term` is declared as `global` and given the type `function`. The symbol is defined as a label, followed by the prologue of the function. The value in register `rdi` (the address of the list structure) is stored on the stack. Then the address of the list buffer is copied into register `rdi` as the first and only argument to the C library function `free`, which deallocates the corresponding memory on the heap. Next the address of the list structure is restored to register `rdi` as the first of two arguments to C library function `bzero`. The second argument, which is copied to register `rsi`, is the size of the list library (which is defined in file `list.inc`). `bzero` will zero out the entire list library and return. After that comes the epilogue of function `list_term`.

(NOTE: When making calls to functions in other shared libraries or the main program, NASM provides the keywords `wrt ..plt` which stands for "With Reference To" "Procedure Linkage Table." More information is available in the NASM documentation in sections "SEG and WRT" and "Calling Procedures Outside the Library").

See Appendix D for the full source code of function `list_term`.

SHARED LIBRARY MAKEFILE

Our goal here is to create a shared library. We need to create a `makefile` that will do the work of creating the shared object file `libutil.so`. So, in folder `util/util/` create the file `makefile`.

Now enter the following into file `makefile`. (NOTE: use tabs instead of spaces to indent lines).

```
# makefile for libutil.so
libutil.so: memmove64.o list.o util.o
	gcc -z noexecstack -shared memmove64.o list.o \
		util.o -o libutil.so
util.o: util.c
	gcc -fPIC -c util.c -o util.o
list.o: list.asm
	nasm -f elf64 list.asm -o list.o
memmove64.o: memmove64.asm
	nasm -f elf64 memmove64.asm -o memmove64.o
clean:
	rm -f libutil.so util.o list.o memmove64.o
```

Now that we have a makefile we can build our shared object file libutil.so. At the command line change directory to folder `util/util/`. Now run the following at the command line.

```
make clean; make
```

Assuming everything went as expected you should see the following output.

```
rm -f libutil.so util.o list.o memmove64.o
nasm -f elf64 memmove64.asm -o memmove64.o
nasm -f elf64 list.asm -o list.o
gcc -fPIC -c util.c -o util.o
gcc -z noexecstack -shared memmove64.o list.o util.o -o libutil.so
```

The `clean` argument tells `make` to execute the commands in the `clean:` section of the `makefile`. The next call to make with no argument tells make to execute the commands that comprise the recipe that produces the prerequisites (in this case files `memmove64.o`, `list.o`, `util.o`) necessary to create the target `libutil.so`; and then, make the shared library `libutil.so`.

At the command line change directory to folder `util/util/` and enter the following command:

```
ls -C1 *.o *.so
```

The `ls` command lists the object files created by the `make` command in folder `util/util/`. If all went well you should see the following files.

```
libutil.so
list.o
memmove64.o
util.o
```

If you do not see all these files or encountered an error message during compilation you will need to check your source files and/or `makefile` for accuracy.

TEST PROGRAM: LIST_INIT

At the command line change directory to folder `util/list_init/` and run the following command combo:

```
make clean; make
```

Now, enter the following command:

```
./list_init
```

You should see the following output from test program `list_init` indicating that all went well.

```
TEST LIST_INIT AND LIST_TERM

LIST_INIT SUCCESSFUL


list:     total:    24
          count:    0
```

```
            o_size:   48
            buffer:   0xf332e0

LIST_TERM SUCCESSFUL

list:       total:    0
            count:    0
            o_size:   0
            buffer:   (nil)
```

The members of the `list` structure are shown after the `list` structure is initialized successfully and after it is de-initialized successfully. Take time to look at the source code for test program `list_init` in files `main.h` and `main.c` in folder `util/list_init/`. The `list_alloc` and `list_free` defines (found in file `util.h` in the folder `util/util/`) are also used.

FUNCTION: LIST_ADD

Now we need a function that will add stuff to our list. We will name the function `list_add`. The `list_add` function will place the new object after the last object in the list. This function will increment the `count` member of the list structure after adding an object. If the list is full the `list_add` function will return `-1` indicating failure. Otherwise, zero will be returned.

In folder `util/util/`, in file `util.h`, enter the following function declaration before the `#endif` directive.

```
int list_add (list_t *, void const *);

%endif
```

Now open file `list.asm` in folder `util/util/` in your text editor. Next, enter the comment-block below into `list.asm` before the line with `%endif`.

```
;
;-------------------------------------------------------------------------------
; C definition:
;
;   void * list_add (list_t *list, void const *object);
;
; param:
;
; rdi = list
; rsi = object
;
; return:
;
; rax = address of object in list | NULL
;
; stack:
```

```
;
;   [rbp - 8]    = rdi (list)
;   [rbp - 16]   = (void *addr) = address of object in list buffer
;-------------------------------------------------------------------------------
;
%endif
```

Again we have the C declaration for function `list_add`; followed by the parameter list indicating which registers hold what data; then the possible return values are indicated; and finally the disposition of the stack.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
      global list_add:function
list_add:
; prologue
      push      rbp
      mov       rbp, rsp
      sub       rsp, 16
; store rdi (list) on stack
      mov       QWORD [rbp - 8], rdi
%endif
```

We have the function declaration as `global` with type `function` followed by the label definition and the prologue. The Stack Pointer (`rsp`) is adjusted by 16-bytes to provide storage for the address of the list structure passed in parameter `list`, and the address where, in the list, the new object will be stored. Then, the address of the list structure is stored on the stack.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
; if (list->count >= list->total) return NULL
      xor       rax, rax
      mov       rcx, QWORD [rdi + list.count]
      cmp       rcx, QWORD [rdi + list.total]
      jae       .epilogue
%endif
```

If the list buffer is full, then `NULL` is returned and the function terminates. Why zero out register `rax` with the `xor` instruction before the comparison is performed? By setting `rax` ahead of the `cmp` instruction only one jump instruction is necessary and the jump instruction mirrors the comparison in the comment above it. If the value in member `count` is above or equal-to the value in member `total` a jump will be made to the instruction following label `.epilogue`. The `jae` (jump, if above or equal-to) instruction is for unsigned comparisons only. For a signed comparison we would have used the `jge` (jump, if greater-than or equal-to) instruction.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
; void *slot = &list->buffer[(list->count * list->o_size)]
      mov       rax, QWORD [rdi + list.count]
      mul       QWORD [rdi + list.o_size]
      add       rax, QWORD [rdi + list.buffer]
      mov       QWORD [rbp - 16], rax
%endif
```

If there is room for another object in the list, then we need the address for that slot. That address is calculated by the first three lines of code above. The value in member `count` is copied to register `rax`. Then the count is multiplied by the object size in member `o_size` leaving the result in `rdx:rax`. (I ignore the value in register `rdx`, since my computer has nowhere near that much memory. Therefore, I expect the value in `rdx` to always be zero). The value in `rax` is an offset from the beginning of the list buffer. So, we will add the address in member `buffer` to the offset in `rax`. Now we have the address of the free slot in the list buffer. The last operation stores the address of the free slot on the stack.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
; (void) memmove64(slot, object, list->o_size)
      mov       rdx, QWORD [rdi + list.o_size]
      mov       rdi, rax
      call      memmove64 wrt ..plt
%endif
```

Now we move the new object into the list buffer. First, the size of an object in bytes is copied from member `o_size` into register `rdx`. Next, the address of the free slot is copied from `rax` to `rdi`. You may have noticed that we have not modified register `rsi` to this point. Therefore, `rsi` still contains the address of the `object` passed by the caller. So, we call function `memmove64` to copy the contents of the new object to the free slot in the list buffer.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
; list->count += 1
      mov       rdi, QWORD [rbp - 8]
      mov       rax, QWORD [rdi + list.count]
      inc       rax
      mov       QWORD [rdi + list.count], rax
; return addr
      mov       rax, QWORD [rbp - 16]
%endif
```

There are two operations to perform before the function epilogue. First, the value in member `count` must be incremented. To do that we restore the address of the list structure to register `rdi`; copy the value in member `count` to register `rax`; increment the value in `rax` by one with the `inc` instruction; and copy the incremented value into member `count`. (Incrementing the count could have been achieved by incrementing the memory location directly). Second, we need to set the function return value in register `rax`. So, the address of the free slot, where the `object` was copied to, is copied to register `rax`.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
.epilogue:
      mov       rsp, rbp
      pop       rbp
      ret
%endif
```

The epilogue restores the Stack Pointer Register (`rsp`) and the Base (or Frame) Pointer Register (`rbp`), erasing any modifications made during execution of the `list_add` function. The `ret` instruction returns execution to the caller.

See Appendix E for the full source code of function `list_add`.

TEST PROGRAM: LIST_ADD

Since, we modified file `list.asm`, we will need to rebuild the shared library file `libutil.so`. At the command line change directory to folder `util/util/`, and run the following command combo:

```
make clean; make
```

Now, change directory to folder `util/list_add/`, and run the same command combo:

```
make clean; make
```

Now, run the `list_add` test program with the following command:

```
./list_add
```

You should see the following output from test program `list_add` indicating that all went well. (NOTE: the buffer address will differ on your system).

```
TEST LIST_ADD

LIST_INIT SUCCESSFUL

list:     total:    36
          count:    0
          o_size:   36
          buffer:   0x19402e0

list_add:    address: 0x19402e0    offset:   0
```

```
list_add:    address: 0x1940304    offset:  36
list_add:    address: 0x1940328    offset:  72
list_add:    address: 0x194034c    offset: 108
list_add:    address: 0x1940370    offset: 144
list_add:    address: 0x1940394    offset: 180
list_add:    address: 0x19403b8    offset: 216
list_add:    address: 0x19403dc    offset: 252
list_add:    address: 0x1940400    offset: 288
list_add:    address: 0x1940424    offset: 324
list_add:    address: 0x1940448    offset: 360
list_add:    address: 0x194046c    offset: 396
list_add:    address: 0x1940490    offset: 432
list_add:    address: 0x19404b4    offset: 468
list_add:    address: 0x19404d8    offset: 504
list_add:    address: 0x19404fc    offset: 540
list_add:    address: 0x1940520    offset: 576
list_add:    address: 0x1940544    offset: 612
list_add:    address: 0x1940568    offset: 648
list_add:    address: 0x194058c    offset: 684
list_add:    address: 0x19405b0    offset: 720
list_add:    address: 0x19405d4    offset: 756
list_add:    address: 0x19405f8    offset: 792
list_add:    address: 0x194061c    offset: 828

LIST_ADD SUCCESSFUL

list:     total:    36
          count:    24
          o_size:   36
          buffer:   0x19402e0

LIST_ADD FAILED AS EXPECTED!

list:     total:    24
          count:    24
          o_size:   36
          buffer:   0x19402e0

LIST_TERM SUCCESSFUL

list:     total:    0
          count:    0
          o_size:   0
          buffer:   (nil)
```

After each function or loop completes the members of the list structure are printed out. After `list_init` completes successfully the list structure print out indicates that: the total number of objects the list can hold is 24; the number of objects in the list is 0; the size of each object is 36-bytes; and the address of the buffer where the objects will be stored—`0x19402e0`. (NOTE: the address you see on your system will be different). Next, the contents of 24 vehicle structures are added to the list via a for-loop. The function name is printed along with the address of the free slot and the offset of the free slot from the beginning of the list buffer. After the for-loop completes the members of the list structure are displayed. This time the count indicates that 24 objects are in the list. The list is now full. To test the integrity of the function, one more call is made to `list_add`, and as expected the `list_add` function returns `NULL` indicating failure. Another display of the list structure confirms that no additional object was added. Next the `list_term` and `list_free` functions are called and the test program terminates. Take time to look at the source code for test program `list_add` in files `main.h` and `main.c` in folder `util/list_add/`.

FUNCTION: LIST_COUNT

This function is very simple and demonstrates how easy it is to access a member of a structure via an Assembly function. (Gives the list structure an object-oriented touch.)

In folder `util/util/`, in file `util.h`, enter the following function declaration before the `#endif` directive.

```
size_t list_count (list_t *);

%endif
```

Now open file `list.asm` in folder `util/util/` in your text editor. Next, enter the comment-block below into `list.asm` before the line with `%endif`.

```
;
;-------------------------------------------------------------------
; C definition:
;
;   size_t list_count (list_t *list);
;
; param:
;
;   rdi = list
;
; return:
;
;   rax = value in member list->count
;-------------------------------------------------------------------
;
%endif
```

Again we have the C declaration of function list_count; followed by the parameter list indicating one parameter is being passed via register rdi; and, then the return value is indicated.

Enter the following code-block into file list.asm before the line with %endif.

```
      global list_count:function
list_count:
      mov       rax, QWORD [rdi + list.count]
      ret
```

We have the function declaration as global with type function followed by the label definition. One parameter is passed to the function which is the address of a list structure. The value in the count member of the list structure is copied to register rax. In this case, the rax register is where the caller will look for the return value of this function. The ret instruction returns execution to the caller.

See Appendix F for the full source code of function list_count.

FUNCTION: LIST_BEGIN

Eventually we will want to iterate through the stuff we add to the list structure. To do that we need a point of reference—somewhere to begin the iteration. That is the purpose of function list_begin. Assuming there is stuff in the list buffer to iterate through, the function will provide an address to the first object in the list buffer. However, we should expect a NULL pointer, if the list buffer is empty.

In folder util/util/, in file util.h, enter the following function declaration before the #endif directive.

```
void * list_begin (list_t *list);

%endif
```

Now in file list.asm in folder util/util/, enter the comment-block below before the line with %endif.

```
;
;---------------------------------------------------------------------------
; C definition:
;
;   void * list_begin (list_t *list);
;
; param:
;
;   rdi = list
;
; return:
```

```
;
;   rax = list->buffer | NULL
;-------------------------------------------------------------------------------
;
%endif
```

The comment block gives us the usual information. Enter the following code-block into file `list.asm` before the line with `%endif`.

```
      global list_begin:function
list_begin:
%endif
```

The symbol `list_begin` is declared `global` with type `function`, and then defined as a label. Next, enter the following code-block into file `list.asm` before the line with `%endif`.

```
; if (list->count == 0) return NULL
      mov       rax, QWORD [rdi + list.count]
      test      rax, rax
      jz        .epilogue
%endif
```

The comment line tells us what to expect from the Assembly code that follows. If the list structure is empty (indicated by the value of the `count` member being zero, then the function returns a `NULL` pointer. However, the code is a bit tricky. If the value of the `count` member is zero then zero is copied into register `rax`. The `test` instruction will set the `zero-flag` should the value in `rax` be zero. That is the requirement of the jump instruction `jz` which performs a jump to the `ret` instruction following the label `.epilogue`. The trick is killing to birds with one stone by using the zero in member `count` as the NULL pointer in register `rax`. As indicated in the comment block above, the function return value will be in register `rax`. The rax register is where the caller of this function expects to find the return value. Next, enter the following code-block into file `list.asm` before the line with `%endif`.

```
; list->index = 0L
      xor       rax, rax
      mov       QWORD [rdi + list.index], rax
%endif
```

If the value in the `count` member is greater than zero, then the `jz` instruction falls through and execution continues with setting the value of member `index` to zero. This is achieved by using the `xor` instruction to zero out the `rax` register. Then, the value in `rax` is copied into member `index`. Next, enter the following code-block into file `list.asm` before the line with `%endif`.

```
; return list->buffer
      mov       rax, QWORD [rdi + list.buffer]
```

```
.epilogue:
      ret
%endif
```

The last operation before the `ret` instruction is to copy the address from the `buffer` member to register `rax` as the return value of function `list_begin`.

See Appendix G for the full source code of function `list_begin`.

FUNCTION: LIST_NEXT

The `list_next` function is an incremental iterator, and works in conjunction with function `list_begin`.

In folder `util/util/`, in file `util.h`, enter the following function declaration before the `#endif` directive.

```
void * list_next (list_t *);

%endif
```

Next, enter the comment-block below into file `list.asm` before the line with `%endif`.

```
;
;-------------------------------------------------------------------------------
; C definition:
;
;   void * list_next (list_t *list);
;
; param:
;
;   rdi = list
;
; return:
;
;   rax = &list->buffer[list->index] | NULL
;-------------------------------------------------------------------------------
;
%endif
```

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
      global list_next:function
list_next:
; if (list->index >= (list->count - 1)) return NULL
      xor       rax, rax
```

```
      mov       rcx, QWORD [rdi + list.count]
      dec       rcx
      mov       rdx, QWORD [rdi + list.index]
      cmp       rdx, rcx
      jae       .return
%endif
```

We have the usual function declaration and definition. The if statement on the comment-line has a comparison to determine whether or not the iterator has reached the end of the objects in the list. If so, then a `NULL` pointer will be returned. The offset (in bytes) of the beginning of the first object in the list buffer is zero. That is, the memory address of the list buffer is also the address of the first object in the buffer. Since, the offset of the first object in the buffer is 0 as illustrated below.

A	B	C	D	E	F
0	1	2	3	4	5

The letters represent objects in the list buffer and the numbers represent the corresponding index of each object.

The index of an object multiplied by the size of the objects in the list, results in the offset of that object from the beginning of the list. And, by adding the offset of an object to the address of the beginning of the list buffer you get the address of the object itself.

Since the index of the first object in the list is zero the corresponding offset will be zero, as the integer value zero multiplied by another integer value is always zero. This leaves the address of the list buffer as the address of the first object in the list. (So simple, and yet, so many words).

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
; list->index += 1
      inc       rdx
      mov       QWORD [rdi + list.index], rdx
%endif
```

The purpose of function `list_next` is to return the address of the next object in the list buffer. Therefore, before that address can be calculated the value of the `index` member needs to be incremented. The value of member `index` is already in register `rdx`. So, the value in `rdx` is incremented; then copied into structure member `index`. (NOTE: we could have incremented the value in the `index` member directly with the following instruction:

```
inc       QWORD [rdi + list.index]
```

In this case, the size prefix QWORD is necessary, and indicates to the assembler that the memory operand is a 64-bit value).

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
; return (list->buffer + (list->index * list->o_size))
      mov       rax, rdx
      mul       QWORD [rdi + list.o_size]
      add       rax, QWORD [rdi + list.buffer]
.return:
      ret
%endif
```

The code-block above, calculates the address of the next object in the list, and places that address in register `rax` as the return value. The value of the incremented `index` member in `rdx` is copied to register `rax`. An offset from the beginning of the list buffer is calculated by multiplying the value in `rax` by the value in structure member `o_size`. Finally, the address (of the next object in the list buffer) is calculated by adding the address of the buffer held in structure member `buffer` to the calculated offset held in register `rax`. Once the calculation is complete the address of the next object in the list resides in register `rax`, and the function returns via the `ret` instruction.

See Appendix H for the full source code of function `list_next`.

TEST PROGRAM: LIST_BEGIN

Since, we modified file `list.asm`, we will need to rebuild the shared library file `libutil.so`. At the command line change directory to folder `util/util/`, and run the following command combo:

```
make clean; make
```

Now, change directory to folder `util/list_begin/`, and run the same command combo:

```
make clean; make
```

Now, run the `list_begin` test program with the following command:

```
./list_begin
```

You should see the following output from test program `list_begin` indicating that all went well. (NOTE: the buffer address will differ on your system).

```
TEST LIST_BEGIN AND LIST_NEXT

LIST_INIT SUCCESSFUL

list:     total:    24
          count:    0
          o_size:   36
          buffer:   0x11f72e0
```

```
LIST_ADD SUCCESSFUL

list:     total:    24
          count:    24
          o_size:   36
          buffer:   0x11f72e0

01:   Ford            Aspire               1994
02:   Chevrolet       Silverado 1500       2003
03:   Buick           Skylark              1997
04:   BMW             Z4                   2012
05:   BMW             Z4                   2008
06:   Oldsmobile      Bravada              2002
07:   Pontiac         Grand Prix           1968
08:   Subaru          Legacy               2007
09:   GMC             Yukon                2006
10:   Mitsubishi      Truck                1991
11:   Hyundai         Tiburon              2001
12:   Dodge           Ram 3500             2002
13:   Alfa Romeo      164                  1993
14:   GMC             Yukon XL 2500        2005
15:   Lexus           LS                   1994
16:   Audi            Q7                   2007
17:   Ford            Explorer             2007
18:   Pontiac         Grand Prix           1987
19:   Mercury         Capri                1993
20:   Hyundai         Equus                2012
21:   GMC             Savana 2500          2005
22:   Lexus           RX                   2010
23:   Lexus           SC                   2007
24:   Volkswagen      Cabriolet            1999

LIST_BEGIN AND LIST_NEXT SUCCESSFUL

LIST_TERM SUCCESSFUL

list:     total:    0
          count:    0
          o_size:   0
          buffer:   (nil)
```

After each function (or iteration of functions) completes the members of the list structure are printed out. After `list_init` completes successfully the list structure print out indicates that: the total number of objects the list can hold is 24; the number of objects in the list is 0; the size of each object is 36-bytes; and the address of the buffer where the objects will be stored—`0x11f72e0`. (NOTE: the address you see on your system will be different). Next, the contents of 24 vehicle structures are added to the list via a for-loop. After the for-loop completes the members of the `list` structure are displayed. This time the count indicates that 24 objects are in the list. The list is now full. The contents of 24 vehicle structures in the list buffer are displayed via a for-loop. Next, the `list_term` and `list_free` functions are called and the test program terminates. Take time to look at the source code for test program `list_begin` in files `main.h` and `main.c` in folder `util/list_begin/`. The iteration functions `list_begin` and `list_next` are used in a for-loop to iterate through the list of vehicle structures.

FUNCTION: LIST_SORT

We need a way to sort our list of vehicles by make, model, year or some combination of the three. In folder `util/util/`, in file `util.h`, enter the following function declaration before the `#endif` directive.

```
void list_sort (list_t *,
        int (*sort_cb) (void const *, void const *));

%endif
```

Next, enter the comment-block below into `list.asm` before the line with `%endif`.

```
;
;-------------------------------------------------------------------------------
; C definition:
;
;   void list_sort (list_t *list, int (*sort_cb) (void const *, void const *));
;
; param:
;
;   rdi = list
;   rsi = sort_cb
;-------------------------------------------------------------------------------
;
%endif
```

The sort function takes two parameters. The first is the address of a `list` structure, and the second is the address of a callback function (`sort_cb`). The callback function takes two parameters, each of which is an object in the list buffer. The callback function is a compare function provided by the caller that must return a 32-bit integer value that is less-than, equal-to, or greater-than zero. The callback functions will be C functions and part of the test program.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
      global list_sort:function
list_sort:
```

```
        push      r15
        mov       rcx, rsi
        mov       rdx, QWORD [rdi + list.o_size]
        mov       rsi, QWORD [rdi + list.count]
        mov       rdi, QWORD [rdi + list.buffer]
        ALIGN_STACK_AND_CALL r15, qsort, wrt, ..plt
        pop       r15
        ret
%endif
```

Yup, we be calling a C library function to do our sorting for us. The `qsort` function takes four parameters. We will copy the parameters in reverse order, though, all that matters is that each parameter is copied into the right register prior to the call to `qsort`. (If you need a refresher on what registers are involved in parameter passing, see the introduction). Register `rsi` holds the callback parameter (`sort_cb`) but will need to hold the `nmemb` parameter for the `qsort` function. For that reason I chose the order above. The `list_sort` function amounts to a wrapper function, but does simplify the call to `qsort` for the caller. We use the macro `ALIGN_STACK_AND_CALL` to make to the call to `qsort`. The macro sets the Stack Pointer to a 16-byte boundary. I use the macro for two reasons. First, it provides a consistent method of getting the stack pointer set correctly before the call. Second, the code is cleaner and the macro name informs us as to what we're doing. Register `r15` is a callee saved register (see introduction), and so we push the value in `r15` on the stack first thing. In the macro, the value in register `rsp` (the Stack Pointer) is stored in register `r15` before the call to `qsort`. After `qsort` returns the value in `r15` is copied back to register `rsp`, thus restoring the Stack Pointer. The macro will accept anywhere from 2 to 4 parameters, hence, the `2-4` in the macro definition. In this case, we use all four parameters, since, `qsort` is a LIBC function located somewhere outside this module.

See Appendix I for the full source code of function `list_sort`.

TEST PROGRAM: LIST_SORT

Since, we modified file `list.asm`, we will need to rebuild the shared library file `libutil.so`. So, at the command line change directory to folder `util/util/`, and run the following command combo:

```
make clean; make
```

Now, change directory to folder `util/list_sort/`, and run the same command combo:

```
make clean; make
```

Now, run the `list_sort` test program with the following command:

```
./list_sort
```

You should see the following output from test program `list_sort` indicating that all went well. (NOTE: the buffer address will differ on your system).

```
TEST LIST_SORT

LIST_INIT SUCCESSFUL

list: total:     24
      count:     0
      o_size:    36
      buffer:    0x14712e0

LIST_ADD SUCCESSFUL

list: total:     24
      count:     24
      o_size:    36
      buffer:    0x14712e0

01:   GMC             Yukon               2006
02:   Ford            Aspire              1994
03:   Chevrolet       Silverado 1500      2003
04:   Buick           Skylark             1997
05:   BMW             Z4                  2012
06:   BMW             Z4                  2008
07:   Oldsmobile      Bravada             2002
08:   Pontiac         Grand Prix          1968
09:   Subaru          Legacy              2007
10:   Mitsubishi      Truck               1991
11:   Hyundai         Tiburon             2001
12:   Dodge           Ram 3500            2002
13:   Alfa Romeo      164                 1993
14:   GMC             Yukon XL 2500       2005
15:   Lexus           LS                  1994
16:   Audi            Q7                  2007
17:   Ford            Explorer            2007
18:   Pontiac         Grand Prix          1987
19:   Mercury         Capri               1993
20:   Hyundai         Equus               2012
21:   Lexus           RX                  2010
22:   Lexus           SC                  2007
23:   Volkswagen      Cabriolet           1999
24:   GMC             Savana 2500         2005

RANDOM LIST SUCCESSFUL

01:   Alfa Romeo      164                 1993
02:   Audi            Q7                  2007
```

```
03:   BMW            Z4                  2012
04:   BMW            Z4                  2008
05:   Buick          Skylark             1997
06:   Chevrolet      Silverado 1500      2003
07:   Dodge          Ram 3500            2002
08:   Ford           Aspire              1994
09:   Ford           Explorer            2007
10:   GMC            Yukon               2006
11:   GMC            Yukon XL 2500       2005
12:   GMC            Savana 2500         2005
13:   Hyundai        Tiburon             2001
14:   Hyundai        Equus               2012
15:   Lexus          LS                  1994
16:   Lexus          RX                  2010
17:   Lexus          SC                  2007
18:   Mercury        Capri               1993
19:   Mitsubishi     Truck               1991
20:   Oldsmobile     Bravada             2002
21:   Pontiac        Grand Prix          1968
22:   Pontiac        Grand Prix          1987
23:   Subaru         Legacy              2007
24:   Volkswagen     Cabriolet           1999

LIST_SORT BY MAKE SUCCESSFUL

01:   Alfa Romeo     164                 1993
02:   Ford           Aspire              1994
03:   Oldsmobile     Bravada             2002
04:   Volkswagen     Cabriolet           1999
05:   Mercury        Capri               1993
06:   Hyundai        Equus               2012
07:   Ford           Explorer            2007
08:   Pontiac        Grand Prix          1968
09:   Pontiac        Grand Prix          1987
10:   Lexus          LS                  1994
11:   Subaru         Legacy              2007
12:   Audi           Q7                  2007
13:   Lexus          RX                  2010
14:   Dodge          Ram 3500            2002
15:   Lexus          SC                  2007
16:   GMC            Savana 2500         2005
17:   Chevrolet      Silverado 1500      2003
18:   Buick          Skylark             1997
19:   Hyundai        Tiburon             2001
20:   Mitsubishi     Truck               1991
21:   GMC            Yukon               2006
22:   GMC            Yukon XL 2500       2005
23:   BMW            Z4                  2012
```

```
24:   BMW          Z4                2008

LIST_SORT BY MODEL SUCCESSFUL

LIST_TERM SUCCESSFUL

list: total:     0
      count:     0
      o_size:    0
      buffer:    (nil)
```

The first listing of vehicles is random and in the order that the vehicle structures were added to the list. The second listing is sorted by vehicle make, which is the second column from the left. The third listing is sorted by vehicle model, which is the third column from the left. Take time to look at the source code for test program `list_sort` in files `main.h` and `main.c` in folder `util/list_sort/`.

FUNCTION: LIST_FIND

In order to find an object in our list structure we have two options: first, iterate through the list of objects until we find what we are looking for; second, sort the list and use a binary search to speed up the process. So, we need a binary search function. My solution is yet another wrapper function—`list_find`.

In folder `util/util/`, in file `util.h`, enter the following function declaration before the `#endif` directive.

```
void * list_find (list_t const *, void const *,
        int (*find_cb) (void const *, void const *));

%endif
```

Now open file `list.asm` in folder `util/util/` in your text editor, and enter the comment-block below before the line with `%endif`.

```
;
;-------------------------------------------------------------------
; C declaration:
;
;   void * list_find (list_t const *list, void const *key,
;       int (*find_cb) (void const *, void const *));
;
; param:
;
;   rdi = list
;   rsi = key
;   rdx = find_cb
```

```
;
; return:
;
;   eax = target (address of matching object) | NULL
;-------------------------------------------------------------------------------
;
```

The find function has three parameters: the address of a list structure; the address of a key to be searched for; and the address of a callback function. As before, the callback function is a compare function provided by the caller that must return a 32-bit integer value that is less-than, equal-to, or greater-than zero. The callback functions will be C functions found in the test program. Unlike function `list_sort`, the `list_find` function has a return value. The return value must be either an address of the found object or a `NULL` pointer.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
      global list_find:function
list_find:
      push      r12
      push      rsi
      mov       r8, rdx
      mov       rcx, QWORD [rdi + list.o_size]
      mov       rdx, QWORD [rdi + list.count]
      mov       rsi, QWORD [rdi + list.buffer]
      pop       rdi
      ALIGN_STACK_AND_CALL r12, bsearch, wrt, ..plt
      pop       r12
      ret
```

First, the contents of (callee-saved) register `r12` is pushed on the stack. The address of a `key` is passed in via register `rsi`. But, the `bsearch` function expects the address of the list buffer to be in register `rsi`. For that reason I chose to push the address in register `rsi` onto the stack. Next, the address of the callback function (parameter `find_cb`) is copied into register `r8`. (If you need a refresher on what registers are involved in parameter passing see the introduction). I chose to copy the parameters into the registers in reverse order (though all that matters is that the parameter be in the correct register for the call to `bsearch`). Next, the value in member `o_size` is copied to register `rcx`; then, the value in member `count` is copied to register `rdx`; next, the address in member `buffer` is copied to register `rsi`; and then, the address of a `key` to be searched for is popped off the stack into register `rdi`. Then, the macro `ALIGN_STACK_AND_CALL` does its thing and the call is made to `bsearch`. Upon completion, `bsearch` will return an address in register `rax` (or NULL). With the return value in register `rax`, nothing further need be done, but, to restore the contents of (callee-saved) register `r12` and return from the `list_find` function.

See Appendix J for the full source code of function `list_find`.

TEST PROGRAM: LIST_FIND

Since, we modified file `list.asm`, we will need to rebuild the shared library file `libutil.so`. So, at the command line change directory to folder `util/util/`, and run the following command combo:

```
make clean; make
```

Now, change directory to folder `util/list_find/`, and run the same command combo:

```
make clean; make
```

Now, run the `list_find` test program with the following command:

```
./list_find
```

You should see the following output from test program `list_find` indicating that all went well. (NOTE: the buffer address will differ on your system).

```
TEST LIST_SORT

LIST_INIT SUCCESSFUL

list: total:     24
      count:     0
      o_size:    36
      buffer:    0xedb2e0

LIST_ADD SUCCESSFUL

list: total:     24
      count:     24
      o_size:    36
      buffer:    0xedb2e0

01:   Alfa Romeo      164                 1993
02:   Audi            Q7                  2007
03:   BMW             Z4                  2012
04:   BMW             Z4                  2008
05:   Buick           Skylark             1997
06:   Chevrolet       Silverado 1500      2003
07:   Dodge           Ram 3500            2002
08:   Ford            Aspire              1994
09:   Ford            Explorer            2007
10:   GMC             Yukon               2006
11:   GMC             Yukon XL 2500       2005
12:   GMC             Savana 2500         2005
13:   Hyundai         Tiburon             2001
14:   Hyundai         Equus               2012
```

```
15:   Lexus          LS                  1994
16:   Lexus          RX                  2010
17:   Lexus          SC                  2007
18:   Mercury        Capri               1993
19:   Mitsubishi     Truck               1991
20:   Oldsmobile     Bravada             2002
21:   Pontiac        Grand Prix          1968
22:   Pontiac        Grand Prix          1987
23:   Subaru         Legacy              2007
24:   Volkswagen     Cabriolet           1999

LIST_SORT BY VEHICLE MAKE SUCCESSFUL

LIST_FIND FOUND ALFA ROMEO

LIST_FIND FOUND LEXUS LS

LIST_FIND FOUND VOLKSWAGEN CABRIOLET

LIST_FIND FAILED TO FIND BUICK LUCERNE AS EXPECTED

LIST_TERM SUCCESSFUL

list: total:    0
      count:    0
      o_size:   0
      buffer:   (nil)
```

The listing of vehicles is sorted by vehicle make, which is the second column from the left. A sorted list, sorted on the search key, is necessary for a binary search. Then a search is made (with function `list_find`) for an Alfa Romeo. That search is successful. Next, a search is made for a Lexus LS, and that search is successful. Then, a search is made for a Volkswagen Cabriolet, which is also successful. Finally, a search is made for a Buick Lucerne, and that search fails as expected. Two of the searches were to test corner cases. For instance, the Alfa Romeo is the first vehicle in the list; and the Volkswagen Cabriolet is the last vehicle in the list. The Lexus LS is somewhere in the middle of the list. Finally, a search is made for a vehicle that is not in the list (the Buick Lucerne). Of course, no one expected the C library function `bsearch` to fail. Our tests have proven only that the `list_find` function and the supporting callback function `find_cb` were working correctly. Take time to look at the source code for test program `list_find` in files `main.h` and `main.c` in folder `util/list_find/`.

FUNCTION: LIST_DELETE

We have a way to add stuff to a list, so, how about a way to delete stuff. The `list_delete` function will remove an object from any slot in the list. The delete function will then shift objects left (if necessary) to fill the empty slot and keep the list contiguous. The function will decrement the `count` member of the list structure after removing an object from the list. And, return a value indicating the success or failure of the delete operation. For instance, if the list is empty the delete operation will fail and the function will return `-1`. Or, if the address passed to the function is outside the address space of the list of objects in the list buffer, the function will fail and return `-1`.

In folder `util/util/`, in file `util.h`, enter the following function declaration before the `#endif` directive.

```
int list_delete (list_t *, void const *);

%endif
```

Now open file `list.asm` in folder `util/util/` in your text editor. Next, enter the comment-block below into `list.asm` before the line with `%endif`.

```
;
;------------------------------------------------------------------------------
; C definition:
;
;   int list_delete (list_t *list, void const *key,
;       int (*find_cb) (void const *, void const *),
;       void (*delete_cb) (void const *));
;
; param:
;
;   rdi = list
;   rsi = key
;   rdx = find_cb
;   rcx = delete_cb
;
; return:
;
;   0 (success) | -1 (failure)
;
; stack:
;
;   QWORD [rbp - 8]    = rdi (list)
;   QWORD [rbp - 16]   = rcx (delete_cb)
;   QWORD [rbp - 24]   = (void *target)
;   QWROD [rbp - 32]   = (void *blk_tail)
;------------------------------------------------------------------------------
;
%endif
```

There are four parameters and four local variables. The parameters include the address of a list structure; the address of a `key` (which could be any basic data type or structure); the address of a callback function for function `list_find`; and the address of a callback function for `list_delete`. The `delete_cb` function provides a way for the user to de-initialize the `target` object prior to deletion. For instance, the `target` object may contain pointers to memory on the heap that need to be freed; or file descriptors that need closing; or any number of other reasons the user might want access to the `target` object prior to deletion.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
      global list_delete:function
list_delete:
; prologue
      push      rbp
      mov       rbp, rsp
      sub       rsp, 32
; store rdi (list) and rcx (delete_cb) on stack
      mov       QWORD [rbp - 8], rdi
      mov       QWORD [rbp - 16], rcx
%endif
```

The code-block includes the function declaration, label definition, and prologue of the `list_delete` function. The address of a list structure and the address of the `delete_cb` function are stored on the stack.

Now, enter the following code-block into file `list.asm` before the line with `%endif`.

```
; if ((target = list_find(list, key, find_cb)) == NULL) return -1
      call      list_find
      mov       QWORD [rbp - 24], rax
      test      rax, rax
      jnz       .target_found
      mov       eax, -1
      jmp       .epilogue
.target_found:
%endif
```

As the C code in the comment-line states, the `list_find` function is called with the key and `find_cb` function. This looks a bit strange, since, we did not provide the parameters to the `list_find` function. As it turns out, the first three parameters to `list_delete` are also the parameters to `list_find`. Those parameters are still in the appropriate registers, so, the call to `list_find` is made. If the `target` object is found, then the address of the `target` object will be in register `rax` upon return from function `list_find`. Otherwise, a `NULL` pointer is returned in `rax`. The `test` instruction will set the `zero-flag` should register `rax` hold a `NULL` value. If the `test` instruction does not set the `zero-flag`, then the `jnz` instruction (jump, if `zero-flag` not set) will perform a jump to label `.target_found`. However, should the `zero-flag` be set, then instuction `jnz` will fall through (not perform a jump) and execution will continue with copying `-1` to register `rax` to indicate that the `list_delete` function has failed. Next, the jmp instruction will update the instruction pointer (register `rip`) with the address of the instruction following the label `.epilogue` where execution will continue.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
; delete_cb(target)
      mov       rcx, QWORD [rbp - 16]
      test      rcx, rcx
      jz        .no_delete_cb
      mov       rdi, rax
      call      rcx
.no_delete_cb:
%endif
```

If execution reaches the code-block above, then the `target` object has been found in the list buffer. In that case, the address of the `delete_cb` function is copied from the stack into register `rcx`. Followed by a test to determine if the address of the `delete_cb` function is `NULL` (the user can pass a `NULL` pointer, if the `delete_cb` function is not needed). As before, if the `delete_cb` address is a `NULL` pointer, then the `zero-flag` is set and the `jz` instruction (jump, if `zero-flag` set) will update the instruction pointer (`rip`) with the address of the instruction following label `.no_delete_cb`. Otherwise, the `jz` instruction will fall through and execution will continue with moving the address of the `target` object from register `rax` to register `rdi` (the first and only parameter to the `delete_cb` function). Until now, we have passed names to the `call` instruction. Now, we pass a register which holds the address of the function we want to call. So, lets take a moment and think this through. When we call a function by name what really takes place? Well, the assembler generates a call to the Procedure Linkage Table if the function is outside the module where the call took place. Otherwise, the address of the called function (a function within the module) is used. However, in this case, we provided the address of the function via a parameter to the `list_delete` function. And that address is in register `rcx`, which is a valid operand to the `call` instruction. (For more on the `call` instruction, see chapter 3 subsection 3 "INSTRUCTIONS (A-L)" of the "Intel 64 and IA-32 Architectures Software Developer's Manual.")

Now, enter the following code-block into file `list.asm` before the line with `%endif`.

```
; void *blk_tail = list->buffer + (list->count * list->o_size)
      mov       rdi, QWORD [rbp - 8]
      mov       rax, [rdi + list.count]
      mul       QWORD [rdi + list.o_size]
      add       rax, QWORD [rdi + list.buffer]
      mov       QWORD [rbp - 32], rax
%endif
```

The code-block above will calculate the address pointing to the end of the last object in the list buffer (`blk_tail`). First, we restore the address of the list structure (the first parameter to function `list_delete`) to register `rdi` from the stack. Then, we perform the equation in the comment line to calculate the address pointing to the end of the last object in the list buffer. We copy the value in member `count` to register `rax`. Next, we multiply the value in register `rax` by the value in member `o_size`. The result of the multiplication is the size of the block of objects in the list buffer (think, adding the size of all the objects in the list together). Then, we add the address in member `buffer` to the calculated offset in register `rax`. The resulting address points to the end of the last object in the list buffer. We store on the address on the stack. (NOTE: The reason register `rax` is used for most math operations is due to the `mul` instruction. The implied first operand of the `mul` instruction is one of the following registers: `RAX`, `EAX`, `AX`, or `AL`. And the result of the `mul` instruction is stored in `AX` (`AH:AL`), `DX:AX`, `EDX:EAX`, or `RDX:RAX`).

Now, enter the following code-block into file `list.asm` before the line with `%endif`.

```
; if (target == (blk_tail - list->o_size)) goto .dec_count
      sub       rax, QWORD [rdi + list.o_size]
      cmp       QWORD [rbp - 24], rax
      je       .dec_count
%endif
```

Next, we need to determine whether or not the `target` object is the last object in the list buffer. The reason for this is that deleting the last object in the list amounts to decrementing the value in member `count` (no move operation is necessary). So, to start with, register `rax` holds the address that points to the end of the last object in the list. We will subtract the size of an object from that address to get the address of the last object in the list. With that done, we compare the address of the `target` object with the address of the last object in the list. Should the addresses be equivalent, execution jumps to the first instruction following label `.dec_count`.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
; void *blk_head = target + list->o_size
      mov       rcx, QWORD [rbp - 24]
      add       rcx, QWORD [rdi + list.o_size]
%endif
```

Now, having determined that the `target` address is not the address of the last object in the list buffer, execution proceeds with calculating the address of the object immediately following the `target` object in the `list` buffer. This address will be the `blk_head`, the address pointing to the beginning of the block of objects that we need to shift left one slot to fill the gap left by the deleted object (`target`). The process is simple enough. We add the value of member `o_size` to the address of the `target` object. The `target` address is copied to register `rcx`. Next, the value of member `o_size` (object size) is added to the value in register `rcx`. With the `blk_head` address calculated execution will proceed to the next block of code.

Enter the following code-block into file `list.asm` before the line with `%endif`.

```
; size_t blk_size = blk_tail - blk_head
      mov       rax, QWORD [rbp - 32]
      sub       rax, rcx
```

```
%endif
```

The memmove64 function has three parameters: the destination address; the source address; and the size of the block of bytes to be copied. In this case, the destination address is the address of the target object; the source address is the blk_head address. So, all that remains to be calculated is the size (in bytes) of the block to be copied. The block size (blk_size) is calculated by subtracting the blk_head address from the blk_tail address. The blk_tail address was calculated earlier and stored on the stack. So, the blk_tail address is copied from the stack into register rax by the mov instruction. The the blk_head address was just calculated and resides in register rcx. The subtraction is performed and the sub instruction puts the resulting value (blk_size) into the destination register rax.

Enter the following code-block into file list.asm before the line with %endif.

```
; memmove64(target, blk_head, blk_size)
      mov       rdi, QWORD [rbp - 24]
      mov       rsi, rcx
      mov       rdx, rax
      call      memmove64 wrt ..plt
%endif
```

With the three parameters calculated and placed in the correct registers, a call is made to the memmove64 function to copy the block of objects one slot (object size) to the left.

Enter the following code-block into file list.asm before the line with %endif.

```
.dec_count:   ; list->count -= 1
      mov       rdi, QWORD [rbp - 8]
      mov       rax, QWORD [rdi + list.count]
      dec       rax
      mov       QWORD [rdi + list.count], rax
%endif
```

Another operation to be performed is adjusting the count member of the list structure to indicate the deletion of an object. Before this can be done the address of the list structure needs to be restored to register rdi. Next, the value in the count member of the list structure is copied to register rax. Then, the value in register rax is decremented by one. Next, the decremented count value is copied back to the count member in the list structure. (NOTE: The decrement operation could have been performed directly on the value in the list structure member count, as follows:

```
      dec       QWORD [rdi + list.count]
```

The size prefix QWORD is necessary, and indicates to the assembler that the memory operand is a 64-bit value.)

Enter the following code-block into file list.asm before the line with %endif.

```
; return 0
      xor       eax, eax
.epilogue:
      mov       rsp, rbp
      pop       rbp
      ret
%endif
```

With the count value updated we move on to setting the return value of the function (in this case zero), and performing the epilogue operations that essentially reverse what was done in the prologue. Then the `ret` instruction returns execution to the caller.

See Appendix I for the full source code of function `list_delete`.

TEST PROGRAM: LIST_DELETE

Since, we modified file `list.asm`, we will need to rebuild the shared library file `libutil.so`. At the command line change directory to folder `util/util/`, and run the following command combo:

```
make clean; make
```

Now, change directory to folder `util/list_delete/`, and run the same command combo:

```
make clean; make
```

Now, run the `list_delete` test program with the following command:

```
./list_delete
```

You should see the following output from test program `list_delete` indicating that all went well. (NOTE: the buffer address will differ on your system).

```
TEST LIST_DELETE

LIST_INIT SUCCESSFUL

list:     total:    24
          count:    0
          o_size:   36
          buffer:   0x1cf42e0

LIST_ADD SUCCESSFUL

list:     total:    24
          count:    24
```

```
          o_size:   36
          buffer:   0x1cf42e0

01:   GMC              Yukon                 2006
02:   Ford             Aspire                1994
03:   Chevrolet        Silverado 1500        2003
04:   Buick            Skylark               1997
05:   BMW              Z4                    2012
06:   BMW              Z4                    2008
07:   Oldsmobile       Bravada               2002
08:   Pontiac          Grand Prix            1968
09:   Subaru           Legacy                2007
10:   Mitsubishi       Truck                 1991
11:   Hyundai          Tiburon               2001
12:   Dodge            Ram 3500              2002
13:   Alfa Romeo       164                   1993
14:   GMC              Yukon XL 2500         2005
15:   Lexus            LS                    1994
16:   Audi             Q7                    2007
17:   Ford             Explorer              2007
18:   Pontiac          Grand Prix            1987
19:   Mercury          Capri                 1993
20:   Hyundai          Equus                 2012
21:   Lexus            RX                    2010
22:   Lexus            SC                    2007
23:   Volkswagen       Cabriolet             1999
24:   GMC              Savana 2500           2005

LIST_BEGIN AND LIST_NEXT SUCCESSFUL

LIST_SORT SUCCESSFUL

01:   Alfa Romeo       164                   1993
02:   Audi             Q7                    2007
03:   BMW              Z4                    2012
04:   BMW              Z4                    2008
05:   Buick            Skylark               1997
06:   Chevrolet        Silverado 1500        2003
07:   Dodge            Ram 3500              2002
08:   Ford             Aspire                1994
09:   Ford             Explorer              2007
10:   GMC              Savana 2500           2005
11:   GMC              Yukon                 2006
12:   GMC              Yukon XL 2500         2005
```

```
13:   Hyundai         Equus               2012
14:   Hyundai         Tiburon             2001
15:   Lexus           LS                  1994
16:   Lexus           RX                  2010
17:   Lexus           SC                  2007
18:   Mercury         Capri               1993
19:   Mitsubishi      Truck               1991
20:   Oldsmobile      Bravada             2002
21:   Pontiac         Grand Prix          1968
22:   Pontiac         Grand Prix          1987
23:   Subaru          Legacy              2007
24:   Volkswagen      Cabriolet           1999

LIST_BEGIN AND LIST_NEXT SUCCESSFUL

DELETING GMC VEHICLES FROM LIST

delete_cb: GMC             Savana 2500         2005
delete_cb: GMC             Yukon XL 2500       2005
delete_cb: GMC             Yukon               2006

LIST_DELETE SUCCESSFUL

list:     total:    24
          count:    21
          o_size:   36
          buffer:   0x1cf42e0

01:   Alfa Romeo      164                 1993
02:   Audi            Q7                  2007
03:   BMW             Z4                  2012
04:   BMW             Z4                  2008
05:   Buick           Skylark             1997
06:   Chevrolet       Silverado 1500      2003
07:   Dodge           Ram 3500            2002
08:   Ford            Aspire              1994
09:   Ford            Explorer            2007
10:   Hyundai         Equus               2012
11:   Hyundai         Tiburon             2001
12:   Lexus           LS                  1994
13:   Lexus           RX                  2010
14:   Lexus           SC                  2007
15:   Mercury         Capri               1993
16:   Mitsubishi      Truck               1991
```

```
17:   Oldsmobile      Bravada             2002
18:   Pontiac         Grand Prix          1968
19:   Pontiac         Grand Prix          1987
20:   Subaru          Legacy              2007
21:   Volkswagen      Cabriolet           1999

LIST_BEGIN AND LIST_NEXT SUCCESSFUL

LIST_DELETE FAILED AS EXPECTED

list:     total:    24
          count:    21
          o_size:   36
          buffer:   0x1cf42e0

LIST_DELETE FAILED AS EXPECTED

list:     total:    24
          count:    21
          o_size:   36
          buffer:   0x1cf42e0

delete_cb: Alfa Romeo      164                 1993

LIST_DELETE SUCCESSFUL

list:     total:    24
          count:    20
          o_size:   36
          buffer:   0x1cf42e0

01:   Audi            Q7                  2007
02:   BMW             Z4                  2012
03:   BMW             Z4                  2008
04:   Buick           Skylark             1997
05:   Chevrolet       Silverado 1500      2003
06:   Dodge           Ram 3500            2002
07:   Ford            Aspire              1994
08:   Ford            Explorer            2007
09:   Hyundai         Equus               2012
10:   Hyundai         Tiburon             2001
11:   Lexus           LS                  1994
12:   Lexus           RX                  2010
13:   Lexus           SC                  2007
```

```
14:    Mercury          Capri               1993
15:    Mitsubishi       Truck               1991
16:    Oldsmobile       Bravada             2002
17:    Pontiac          Grand Prix          1968
18:    Pontiac          Grand Prix          1987
19:    Subaru           Legacy              2007
20:    Volkswagen       Cabriolet           1999

delete_cb: Volkswagen      Cabriolet           1999

LIST_DELETE SUCCESSFUL

list:     total:    24
          count:    19
          o_size:   36
          buffer:   0x1cf42e0

01:    Audi             Q7                  2007
02:    BMW              Z4                  2012
03:    BMW              Z4                  2008
04:    Buick            Skylark             1997
05:    Chevrolet        Silverado 1500      2003
06:    Dodge            Ram 3500            2002
07:    Ford             Aspire              1994
08:    Ford             Explorer            2007
09:    Hyundai          Equus               2012
10:    Hyundai          Tiburon             2001
11:    Lexus            LS                  1994
12:    Lexus            RX                  2010
13:    Lexus            SC                  2007
14:    Mercury          Capri               1993
15:    Mitsubishi       Truck               1991
16:    Oldsmobile       Bravada             2002
17:    Pontiac          Grand Prix          1968
18:    Pontiac          Grand Prix          1987
19:    Subaru           Legacy              2007


LIST_DELETE SUCCESSFUL (NO CALLBACK)

list:     total:    24
     count:    18
     o_size:   36
     buffer:   0x1cf42e0
```

```
01:   Audi          Q7                  2007
02:   BMW           Z4                  2012
03:   BMW           Z4                  2008
04:   Buick         Skylark             1997
05:   Chevrolet     Silverado 1500      2003
06:   Dodge         Ram 3500            2002
07:   Ford          Aspire              1994
08:   Ford          Explorer            2007
09:   Hyundai       Equus               2012
10:   Hyundai       Tiburon             2001
11:   Lexus         LS                  1994
12:   Lexus         RX                  2010
13:   Lexus         SC                  2007
14:   Mercury       Capri               1993
15:   Mitsubishi    Truck               1991
16:   Pontiac       Grand Prix          1968
17:   Pontiac       Grand Prix          1987
18:   Subaru        Legacy              2007

LIST_TERM SUCCESSFUL

list:     total:    0
          count:    0
          o_size:   0
          buffer:   (nil)
```

The members of the `list` structure are printed out after every operation to indicate success. After `list_init` completes successfully the `list` structure print out indicates that: the total number of objects the list can hold is 24; the number of objects in the list is 0; the size of each object is 36-bytes; and the address of the buffer where the objects will be stored—`0x233c2e0`. (NOTE: the address you see on your system will be different). Next, the contents of 24 vehicle structures are added to the list via a for-loop. Then the contents of the `list` structure is displayed with the count indicating that all 24 vehicle structures were indeed added to the list buffer. The list is now full. Next, the contents of the vehicle structures in the list buffer are displayed with the help the iteration functions: `list_begin` and `list_next`. After that, all the GMC vehicles are targeted for deletion from the list. (Nothing against GMC vehicles). The contents of the list structure are displayed with the count now 21 instead of 24. All three GMC vehicles were deleted successfully from the list as the following listing indicates. Next, two attempts are made to delete vehicles that not in the list. Both, attempts fail as expected. Next we successfully delete the first vehicle (`Alfa Romeo`) and the last vehicle (`Volkswagen`) from the list. The importance of these two deletions is to test corner cases: one at the beginning of the list; and the other at the end. The last test involves deleting a vehicle (`Oldsmobile`) without providing a callback function. The deletion is successful as the last listing of vehicles indicates. However, there is no printout of the vehicle being deleted by the callback function. Finally, the `list_term` and `list_free` functions are called and the test program terminates. Take time to look at the source code for test program `list_delete` in files `main.h` and `main.c` in folder `util/list_delete/`.

I hope you found this tutorial helpful. If you want a challenge see Appendix A for an exercise.

Appendix A

Exercises:

1. We have added a most of the bells and whistles you would expect for a glorified array. However, there is one I can think of that we left undone. The iterator functions `list_end` and `list_prev` to iterate from the end of the list to the beginning. And a test program for these functions.

Appendix B

```
;-------------------------------------------------------------------------------
; C/C++ definition:
;
;   void * memmove64 (void *dst, void const *src, size_t size)
;
; passed in:
;
;   rdi = dst
;   rsi = src
;   rdx = size
;
; returned:
;
;   rax = dst
;
; WARNING: this routine does not handle the overlapping source-
;          destination senario.
;-------------------------------------------------------------------------------
;
      global memmove64:function
memmove64:
      push      rdi            ; push destination address on stack
; quadword count = size / QW_SIZE
      mov       rax, rdx       ; copy value of size parameter to rax
      xor       rdx, rdx       ; zero out rdx
      mov       r11, QW_SIZE
      div       r11            ; rdx:rax = (byte count):(quadword count)
      mov       rcx, rax       ; copy quadword count to rcx
      cld                      ; increment index registers rsi and rdi
      rep movsq                ; repeat quadword move operation
      mov       rcx, rdx       ; copy byte count to rcx
      rep movsb                ; repeat byte move operation
      pop       rax            ; pop destination address off stack
      ret
```

Appendix C

```
;-------------------------------------------------------------------------------
; C declaration:
;
;   int list_init (list_t *list, size_t const o_size);
;
; param:
;
;   rdi = list
;   rsi = o_size
;
; return:
;
;   eax = 0 (success) | -1 (failure)
;
; stack:
;
;   [rbp - 8]   = rdi (list)
;-------------------------------------------------------------------------------
;
      global list_init:function
list_init:
; prologue
      push      rbp
      mov       rbp, rsp
      sub       rsp, 8
; store rdi (list) on stack
      mov       QWORD [rbp - 8], rdi
; list->o_size = o_size
      mov       QWORD [rdi + list.o_size], rsi
; buffer_size = o_size * LIST_COUNT
      mov       rax, rsi
      mov       rcx, QWORD LIST_COUNT
      mul       rcx
; buffer_size = (buffer_size + ALIGN_WITH) & ALIGN_MASK
      add       rax, QWORD ALIGN_WITH
      and       rax, QWORD ALIGN_MASK
; if ((list->buffer = calloc(1, buffer_size)) == NULL) return -1
      mov       rdi, 1
      mov       rsi, rax
      call      calloc wrt ..plt
      mov       rdi, QWORD [rbp - 8]
      mov       QWORD [rdi + list.buffer], rax
      test      rax, rax
      jnz       .continue
      mov       eax, -1
      jmp       .epilogue
.continue:
; list->total = LIST_COUNT
      mov       rax, LIST_COUNT
```

Appendix C (cont.)

```
      mov       QWORD [rdi + list.total], rax
; list->count = 0
      xor       rax, rax
      mov       QWORD [rdi + list.count], rax
; return 0
;     xor       eax, eax
.epilogue:
      mov       rsp, rbp
      pop       rbp
      ret
```

Appendix D

```
;----------------------------------------------------------------------------
; C definition:
;
;   void list_term (list_t *list)
;
; param:
;
;   rdi = list
;
; stack:
;
;   [rbp - 8] = rdi (list}
;----------------------------------------------------------------------------
;
      global list_term:function
list_term:
; prologue
      push      rbp
      mov       rbp, rsp
      sub       rsp, 8
; store rdi (list) on stack
      mov       QWORD [rbp - 8], rdi
; free list buffer memory on heap
      mov       rdi, QWORD [rdi + list.buffer]
      call      free wrt ..plt
; zero out list structure
      mov       rdi, QWORD [rbp - 8]
      mov       rsi, QWORD listSize
      call      bzero wrt ..plt
; epilogue
      mov       rsp, rbp
      pop       rbp
      ret
```

Appendix E

```
;---------------------------------------------------------------------------------
; C definition:
;
;   void * list_add (list_t *list, void const *object);
;
; param:
;
;   rdi = list
;   rsi = object
;
; return:
;
;   rax = address of object in list | NULL
;
; stack:
;
;   [rbp - 8]    = rdi (list)
;   [rbp - 16]   = (void *addr) = address of object in list buffer
;---------------------------------------------------------------------------------
;
      global list_add:function
list_add:
; prologue
      push      rbp
      mov       rbp, rsp
      sub       rsp, 16
; store rdi (list) on stack
      mov       QWORD [rbp - 8], rdi
; if (list->count >= list->total) return NULL
      xor       rax, rax
      mov       rcx, QWORD [rdi + list.count]
      cmp       rcx, QWORD [rdi + list.total]
      jae       .epilogue
; void *slot = &list->buffer[(list->count * list->o_size)]
      mov       rax, QWORD [rdi + list.count]
      mul       QWORD [rdi + list.o_size]
      add       rax, QWORD [rdi + list.buffer]
      mov       QWORD [rbp - 16], rax
; (void) memmove64(slot, object, list->o_size)
      mov       rdx, QWORD [rdi + list.o_size]
      mov       rdi, rax
      call      memmove64 wrt ..plt
; list->count += 1
      mov       rdi, QWORD [rbp - 8]
      mov       rax, QWORD [rdi + list.count]
      inc       rax
      mov       QWORD [rdi + list.count], rax
; return addr
      mov       rax, QWORD [rbp - 16]
.epilogue:
      mov       rsp, rbp
```

Appendix E (cont.)

```
    pop     rbp
    ret
```

Appendix F

```
;-------------------------------------------------------------------------------
; C definition:
;
;   size_t list_count (list_t *list);
;
; param:
;
;   rdi = list
;
; return:
;
;   rax = list->count
;-------------------------------------------------------------------------------
;
      global list_count:function
list_count:
      mov       rax, QWORD [rdi + list.count]
      ret
```

Appendix G

```
;-------------------------------------------------------------------------------
; C definition:
;
;   void * list_begin (list_t *list);
;
; param:
;
;   rdi = list
;
; return:
;
;   rax = list->buffer | NULL
;-------------------------------------------------------------------------------
;
      global list_begin:function
list_begin:
; if (list->count == 0) return NULL
      mov       rax, QWORD [rdi + list.count]
      test      rax, rax
      jz        .epilogue
; list->index = 0L
      xor       rax, rax
      mov       QWORD [rdi + list.index], rax
; return list->buffer
      mov       rax, QWORD [rdi + list.buffer]
.epilogue:
      ret
```

Appendix H

```
;------------------------------------------------------------------------------
; C definition:
;
;   void * list_next (list_t *list);
;
; param:
;
;   rdi = list
;
; return:
;
;   rax = (list->buffer + ((list->index + 1) * list->o_size)) | NULL
;------------------------------------------------------------------------------
;
      global list_next:function
list_next:
; if (list->index >= (list->count - 1)) return NULL
      xor       rax, rax
      mov       rcx, QWORD [rdi + list.count]
      dec       rcx
      mov       rdx, QWORD [rdi + list.index]
      cmp       rdx, rcx
      jae       .return
; list->index += 1
      inc       rdx
      mov       QWORD [rdi + list.index], rdx
; return (list->buffer + (list->index * list->o_size))
      mov       rax, rdx
      mul       QWORD [rdi + list.o_size]
      add       rax, QWORD [rdi + list.buffer]
.return:
      ret
```

Appendix I

```
;-------------------------------------------------------------------------------
; C definition:
;
;   void list_sort (list_t *list, int (*sort_cb) (void const *, void const *));
;
; param:
;
;   rdi = list
;   rsi = sort_cb
;-------------------------------------------------------------------------------
;
      global list_sort:function
list_sort:
      push      r15
      mov       rcx, rsi
      mov       rdx, QWORD [rdi + list.o_size]
      mov       rsi, QWORD [rdi + list.count]
      mov       rdi, QWORD [rdi + list.buffer]
      ALIGN_STACK_AND_CALL r15, qsort, wrt, ..plt
      pop       r15
      ret
```

Appendix J

```
;-------------------------------------------------------------------------------
; C declaration:
;
;   void * list_find (list_t const *list, void const *key,
;       int (*find_cb) (void const *, void const *));
;
; param:
;
;   rdi = list
;   rsi = key
;   rdx = find_cb
;
; return:
;
;   eax = iter (address of matching object) | NULL
;-------------------------------------------------------------------------------
;
      global list_find:function
list_find:
      push      r12
      push      rsi
      mov       r8, rdx
      mov       rcx, QWORD [rdi + list.o_size]
      mov       rdx, QWORD [rdi + list.count]
      mov       rsi, QWORD [rdi + list.buffer]
      pop       rdi
      ALIGN_STACK_AND_CALL r12, bsearch, wrt, ..plt
      pop       r12
      ret
```

Appendix K

```
;--------------------------------------------------------------------------
; C definition:
;
;   int list_delete (list_t *list, void const *key,
;       int (*find_cb) (void const *, void const *),
;       void (*delete_cb) (void const *));
;
; param:
;
;   rdi = list
;   rsi = key
;   rdx = find_cb
;   rcx = delete_cb
;
; return:
;
;   0 (success) | -1 (failure)
;
; stack:
;
;   QWORD [rbp - 8]    = rdi (list)
;   QWORD [rbp - 16]   = rcx (delete_cb)
;   QWORD [rbp - 24]   = (void *target)
;   QWROD [rbp - 32]   = (void *blk_tail)
;--------------------------------------------------------------------------
;
      global list_delete:function
list_delete:
; prologue
      push      rbp
      mov       rbp, rsp
      sub       rsp, 32
; store rdi (list) and rcx (delete_cb) on stack
      mov       QWORD [rbp - 8], rdi
      mov       QWORD [rbp - 16], rcx
; if ((target = list_find(list, key, find_cb)) == NULL) return -1
      call      list_find
      mov       QWORD [rbp - 24], rax
      test      rax, rax
      jnz       .target_found
      mov       eax, -1
      jmp       .epilogue
.target_found:
; delete_cb(target)
      mov       rcx, QWORD [rbp - 16]
      test      rcx, rcx
```

Appendix K (cont.)

```
      jz        .no_delete_cb
      mov       rdi, rax
      call      rcx
.no_delete_cb:
; void *blk_tail = list->buffer + (list->count * list->o_size)
      mov       rdi, QWORD [rbp - 8]
      mov       rax, [rdi + list.count]
      mul       QWORD [rdi + list.o_size]
      add       rax, QWORD [rdi + list.buffer]
      mov       QWORD [rbp - 32], rax
; if (target == (blk_tail - list->o_size)) goto .dec_count
      sub       rax, QWORD [rdi + list.o_size]
      cmp       QWORD [rbp - 24], rax
      je       .dec_count
; void *blk_head = target + list->o_size
      mov       rcx, QWORD [rbp - 24]
      add       rcx, QWORD [rdi + list.o_size]
; size_t blk_size = blk_tail - blk_head
      mov       rax, QWORD [rbp - 32]
      sub       rax, rcx
; memmove64(target, blk_head, blk_size)
      mov       rdi, QWORD [rbp - 24]
      mov       rsi, rcx
      mov       rdx, rax
      call      memmove64 wrt ..plt
.dec_count:   ; list->count -= 1
      mov       rdi, QWORD [rbp - 8]
      mov       rax, QWORD [rdi + list.count]
      dec       rax
      mov       QWORD [rdi + list.count], rax
; return 0
      xor       eax, eax
.epilogue:
      mov       rsp, rbp
      pop       rbp
      ret
```