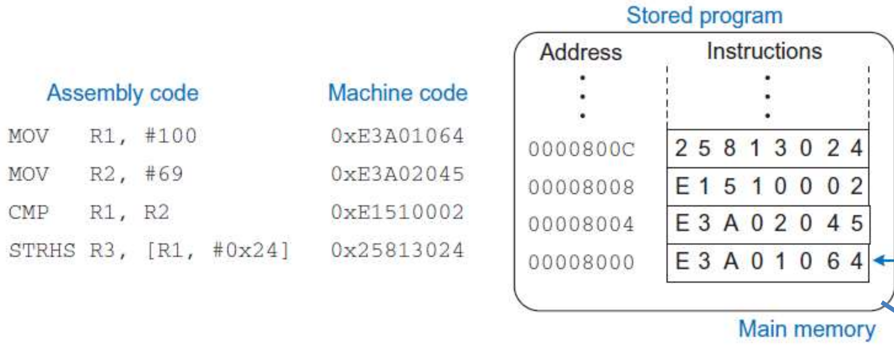
ARM Assembly Programming

Dynamic Data Structure & OOP

A Programmer's Perspective



```
MOV   R1, #100           0xE3A01064
MOV   R2, #69            0xE3A02045
CMP   R1, R2             0xE1510002
STRHS R3, [R1, #0x24]    0x25813024
```

Figure 6.28 **Stored program**

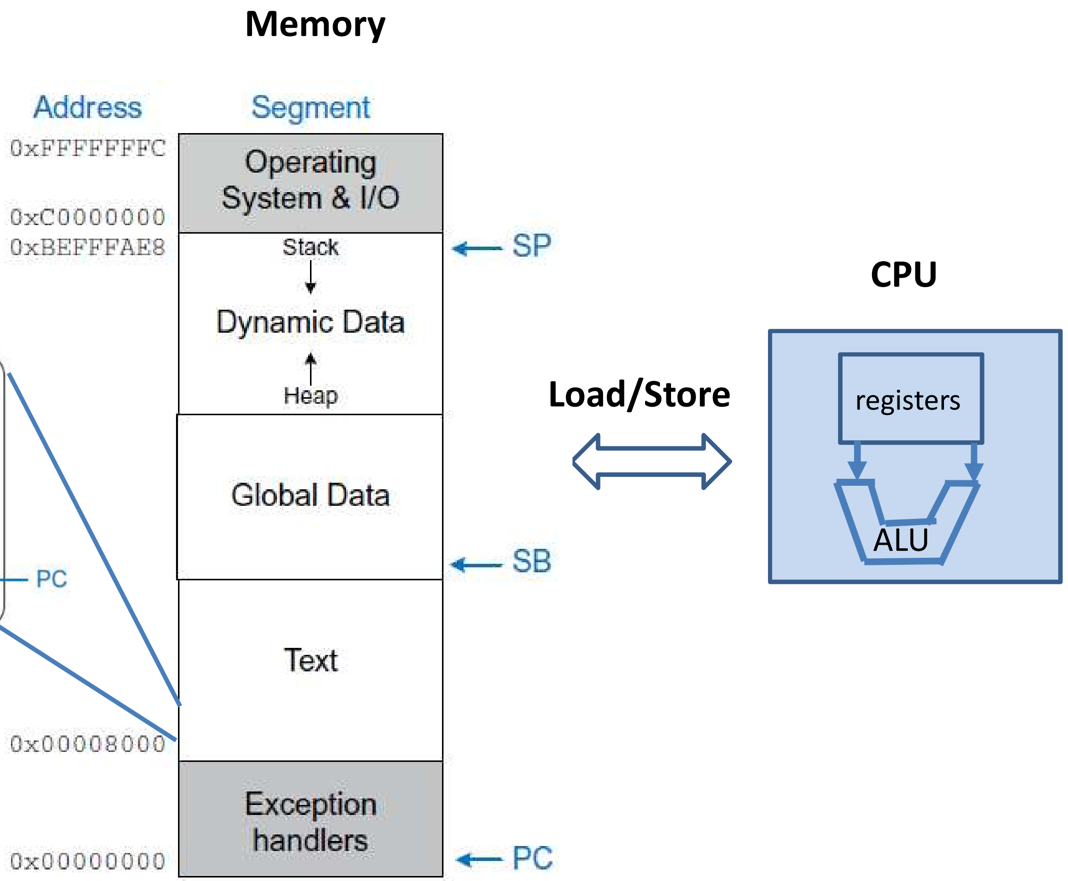


Figure 6.30 **Example ARM memory map**

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Comparison of Array vs. Pointer

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

Array initialization

In C language,

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
char pattern = “ould”;
char pattern1 = {‘o’, ‘u’, ‘l’, ‘d’, ‘\0’};
```

null character

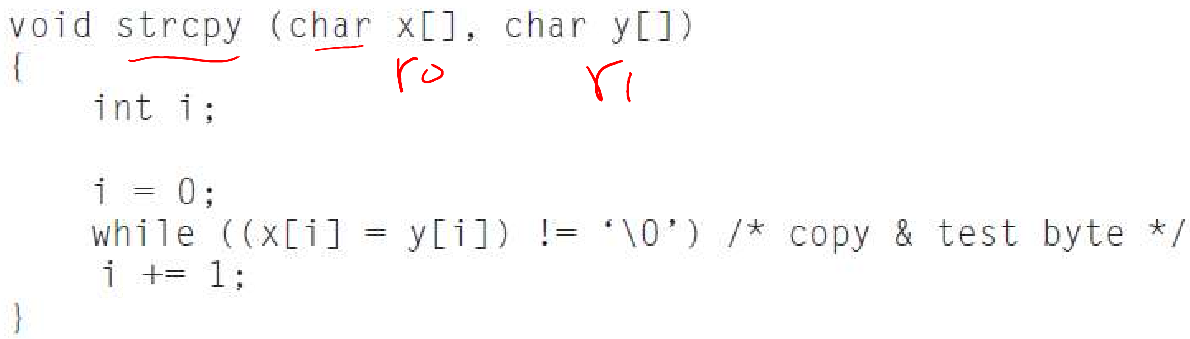
In assembly,

```
.data
 days:  .word  31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
 pattern1: .byte 111, 117, 108, 100, 0
 pattern: .asciz  “ould”
```

Handling large immediate values, label addresses, words, and bytes, …

```
.text
 @mov r0, #345          @ see this number cannot be used as immediate value

 ldr r0, =0x12345678    @ (pseudo instruction) the way to load a large number to register
                        @ see where the number is and pc-relative addressing

 ldr r1, =myByte        @ the way to load address of a label to register

 ldr r2, [r1]           @ see the order of these 4 bytes in memory and in register

 str r0, [r1]           @ see the 4 bytes in a word are stored in memory (little endian)

 ldrb r4, [r1]          @ see which byte in 0x12345678 is loaded back

.data
myByte: .byte 1, 2, 3, 4
```

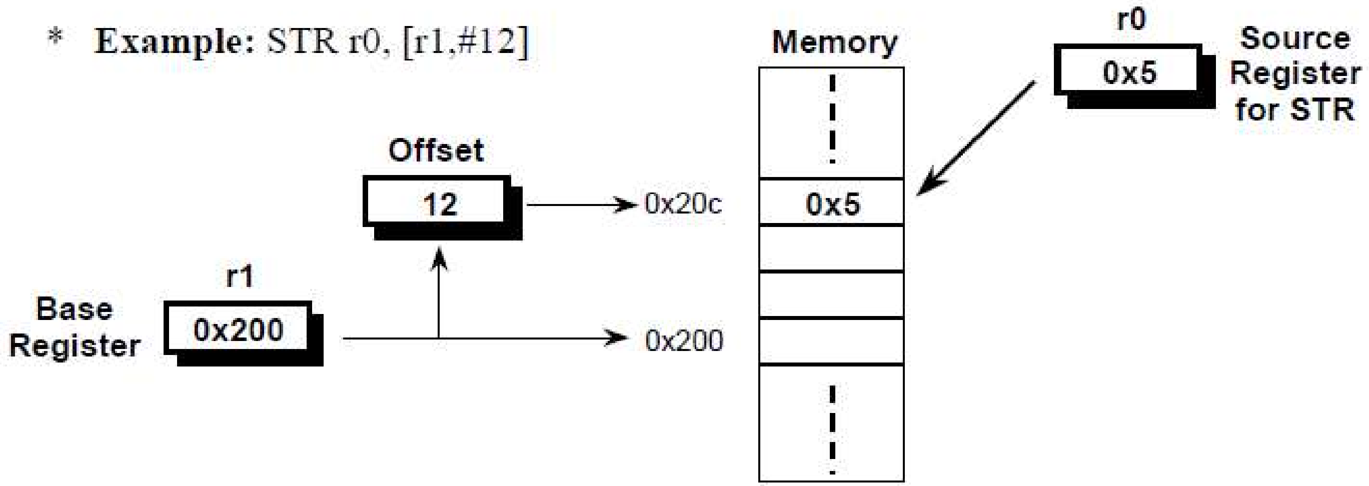
literal pool



```
void strcpy (char x[], char y[])
{                 r0        r1
    int i;

    i = 0;
    while ((x[i] = y[i]) != ‘\0’) /* copy & test byte */
     i += 1;
}
```

```
strcpy:        sp
        sub    sp, #4
        str    r4, [sp, #0]
        mov    r4, #0
L1:     add    r2, r4, r1
        ldrsb  r3, [r2, #0]
        add    r12, r4, r0
        strb   r3, [r12, #0]
        cmp    r3, #0
        beq    L2
        add    r4, r4, #1
        b      L1
L2:     ldr    r4, [sp, #0]
        add    sp, sp, #4
        mov    pc, lr
```

Note: r4 is for saved value; r12 for temp value



* Example: STR r0, [r1,#12]

STR r0, [r1, r2, LSL #2] @ address = r1 + 4 x r2
@ if r2 has value 3, this has the same effect of STR r0, [r1,#12].

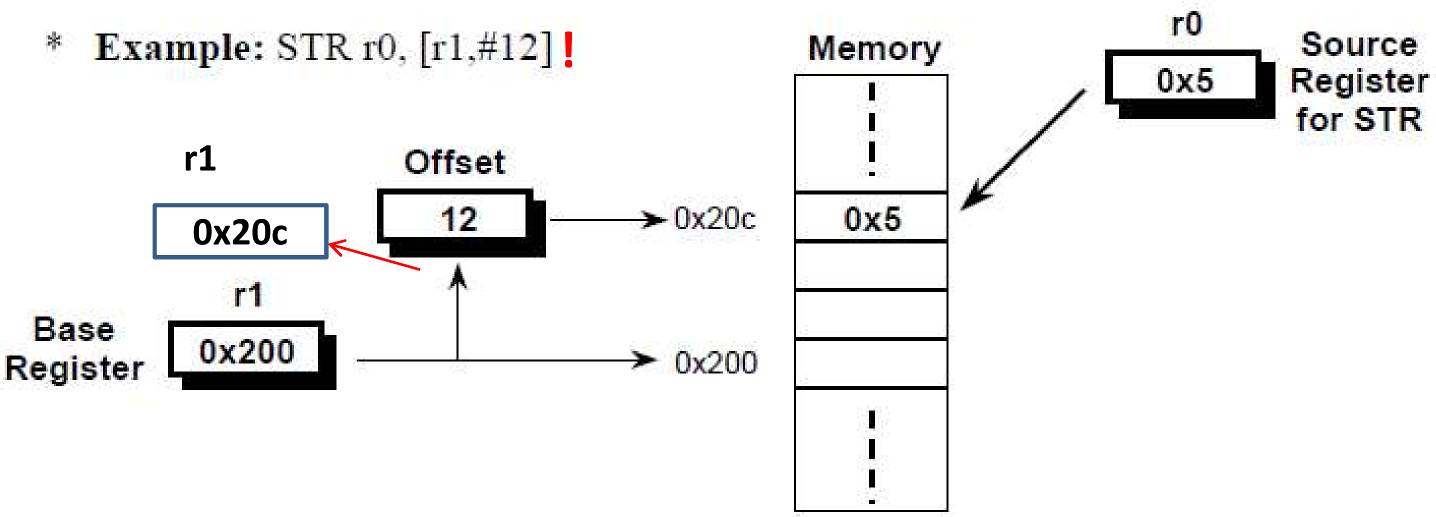
STR r0, [r1, #12]! @ pre-indexing, r0 =M[r1+12], r1 = r1 + 12

STR r0 [r1], #12 @ post-indexing, r0 = M[r1], r1 = r1 +12

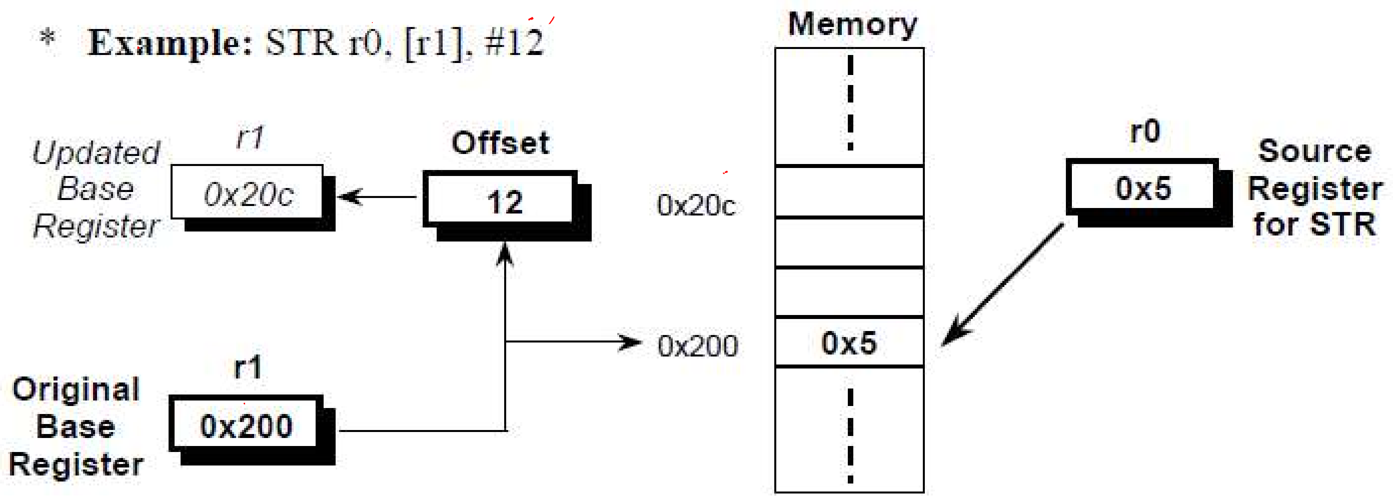
Pre-indexing

* **Example:** STR r0, [r1,#12] !

Updated base register



Post-indexing



* **Example:** STR r0, [r1], #12

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
@ r0 = pointer to array;
@ r1 = size
clear1:
  mov   r2  #0        @ index i
  mov   r3, #0        @ constant zero
Loop:
  add   r4, r0, r2 LSL #2
  str   r3, [r4]
  add   r2, r2, #1
  cmp   r2, r1
  blt   loop
```

```
@ r0 = pointer to array;
@ r1 = size
clear1:
  mov   r2  #0        @ index i
  mov   r3, #0        @ constant zero
Loop:
  str   r3, [r0, r2, LSL #2]
  add   r2, r2, #1
  cmp   r2, r1
  blt   loop
```

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size]; p = p + 1)
    *p = 0;
}
```

```
@ r0 = pointer to array;
@ r1 = size
clear1:
   mov  r2  #0             @ index i
   mov  r3, #0             @ constant zero
loop1:
   str   r3, [r0, r2, LSL #2]
   add   r2, r2, #1
   cmp   r2, r1
   blt   loop 1
```

```
@ r0 = pointer to array;
@ r1 = size
clear2:
   mov  r2  r0
   mov  r3, #0          @ constant zero
loop2:
   str   r3, [r2], #4    @post-indexing
   cmp   r2, r1
   blt   loop2
```

Dynamic Data Structures
linked-list, tree, …

Dynamic memory allocation on the heap

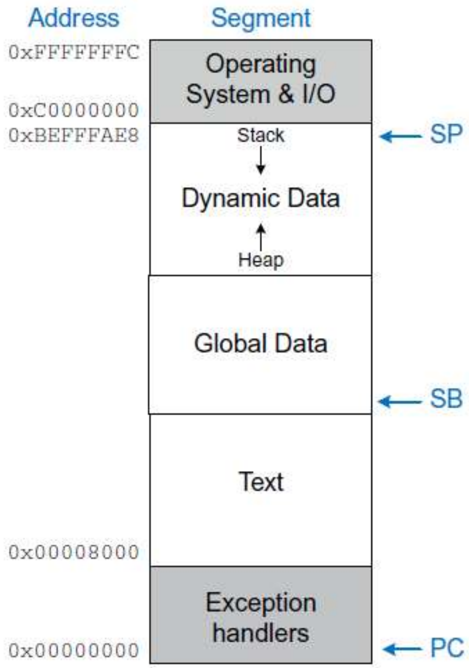


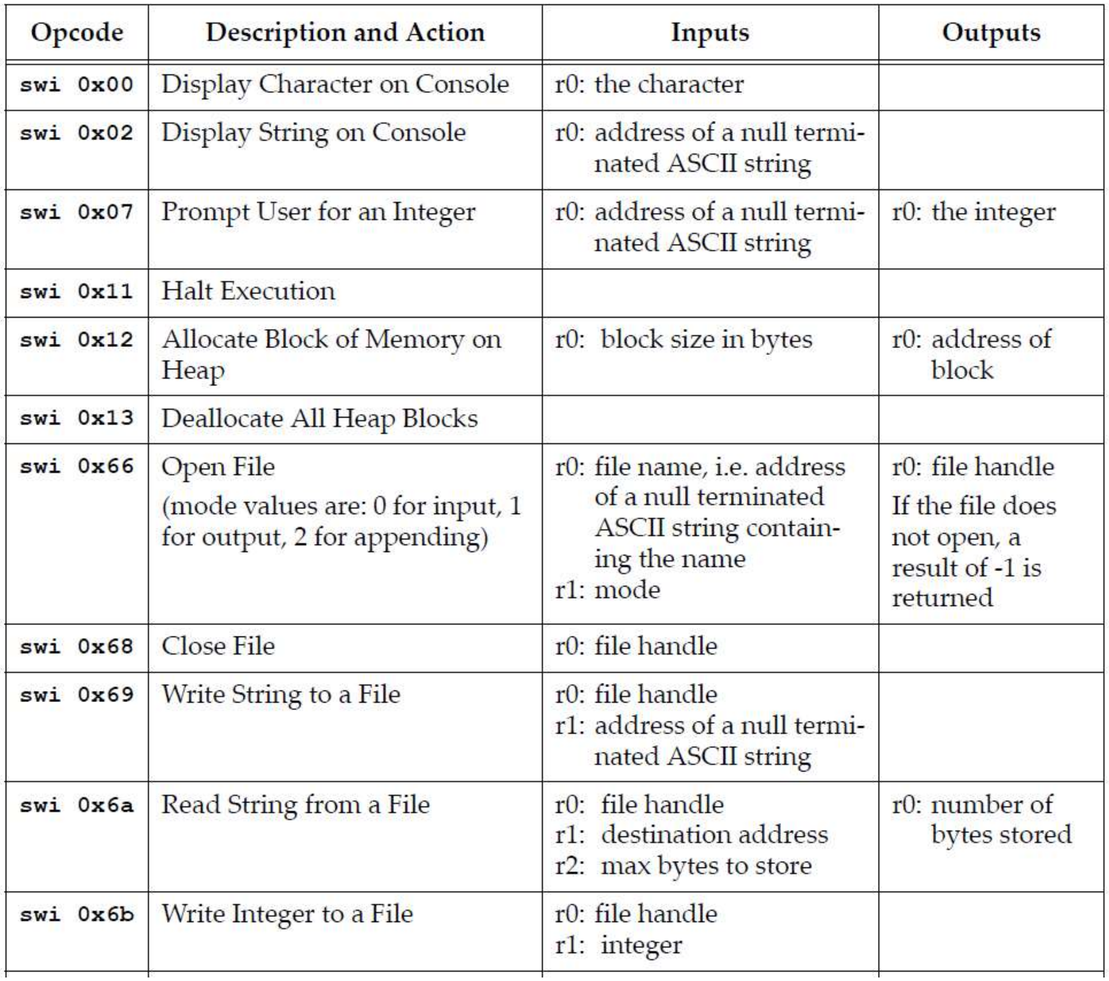
Figure 6.30 Example ARM memory map

In C language, we use
*malloc(unsigned, nbytes)

In ARM assembly, swi instruction is used to request a block of memory from the heap

```
MOV    r0, #12  @ r0 = 12 bytes, the requested size
SWI    0x12     @ SWI instruction to request memory space from the heap
                @ r0 contains the address of the allocated space.
```

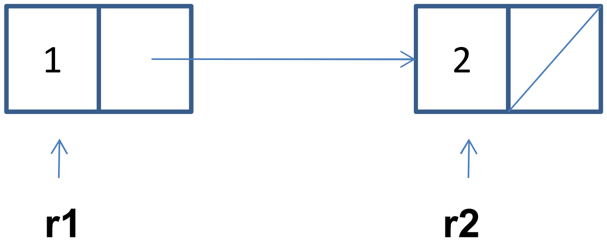
Table 1: SWI operations (0x00 - 0xFF)



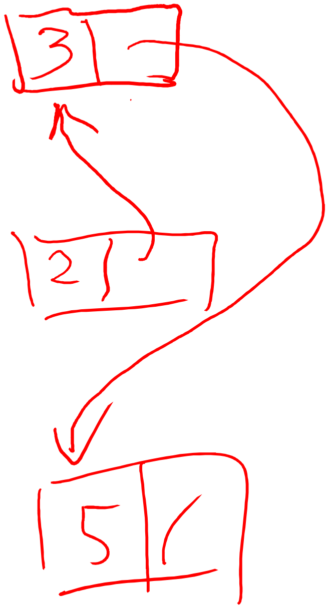
Opcode	Description and Action	Inputs	Outputs
`swi 0x00`	Display Character on Console	r0: the character	
`swi 0x02`	Display String on Console	r0: address of a null terminated ASCII string	
`swi 0x07`	Prompt User for an Integer	r0: address of a null terminated ASCII string	r0: the integer
`swi 0x11`	Halt Execution		
`swi 0x12`	Allocate Block of Memory on Heap	r0: block size in bytes	r0: address of block
`swi 0x13`	Deallocate All Heap Blocks		
`swi 0x66`	Open File (mode values are: 0 for input, 1 for output, 2 for appending)	r0: file name, i.e. address of a null terminated ASCII string containing the name r1: mode	r0: file handle If the file does not open, a result of -1 is returned
`swi 0x68`	Close File	r0: file handle	
`swi 0x69`	Write String to a File	r0: file handle r1: address of a null terminated ASCII string	
`swi 0x6a`	Read String from a File	r0: file handle r1: destination address r2: max bytes to store	r0: number of bytes stored
`swi 0x6b`	Write Integer to a File	r0: file handle r1: integer	

Linked List

```
MOV     r0, #8
SWI     0x12

MOV     r1, r0
MOV     r3, #1
STR     r3, [r1, #0]

MOV     r0, #8
SWI     0x12

MOV     r2, r0

STR     r2, [r1, #4]

MOV     r3, #2
STR     r3, [r2, #0]
MOV     r3, #0
STR     r3, [r2, #4]
```



3 2 5

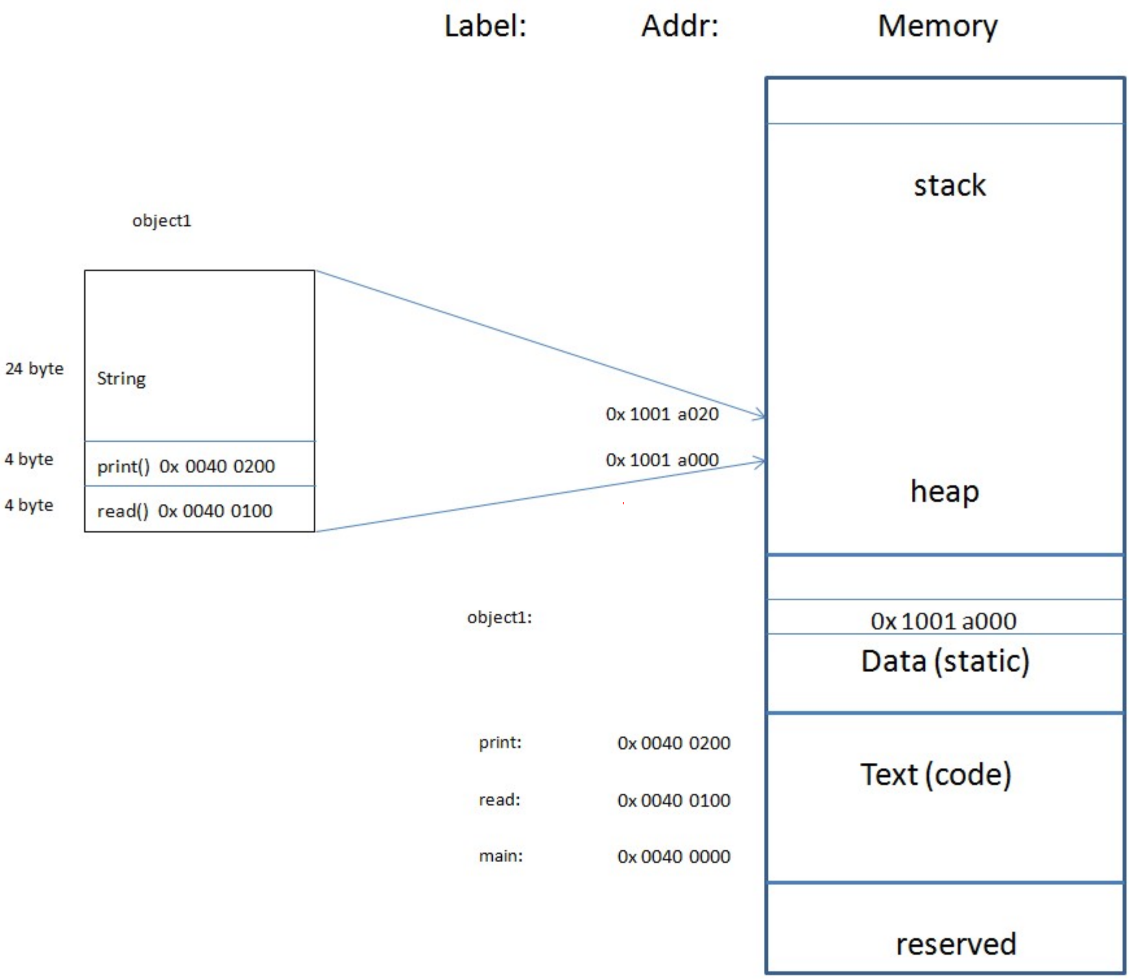


```
@ read integers from a file and insert them into a linked list to get sorted
@ and print the sorted integers to the screen (stdout).

.text
main:
                @ open an input file to read integers
                ldr r0, =InFileName
                mov r1, #0
                swi 0x66                        @ open file
                ldr r1, =InFileHandle
                str r0, [r1]

  Loop:
                @ read integer from file
                ldr r1, =InFileHandle
                ldr r0, [r1]
                swi 0x6c        @ read an integer put in r0
                BCS CloseF
                mov r3, r0      @ copy r0 to r3

                mov r1, r3
                MOV r0, #1    @ Load 1 into register r0 (stdout handle)
                SWI 0x6b     @ Print integer in register r1 to stdout
                mov r0, #1
                ldr  r1,  =Space
                swi  0x69

                B Loop
   CloseF:
                @close infile
                ldr r0, =InFileHandle
                ldr r0, [r0]
                swi 0x68
exit:           SWI 0x11     @ Stop program execution

.data
MyList: .word 0
InFileName: .asciz "list.txt"
InFileHandle: .word 0
Space: .ascii " "
```

Object-Oriented Programming

Example (in pseudo java code)

```
// this main function is in some other class.
Public static void main(String[] args) {
        Object object1;
        object1 = new object();
        object1.read();
        object1.print();
}

Class Object {
        String string;
        Public void read() {
                System.out.println(“Enter data”);
                this.string = System.in.read();   // this is not java code
        }

        Public void print() {
                System.out.println(this.string);
        }
}
```



Label: Addr: Memory

object1

24 byte String

4 byte print() 0x 0040 0200

4 byte read() 0x 0040 0100

0x 1001 a020

0x 1001 a000

stack

heap

object1: 0x 1001 a000

Data (static)

print: 0x 0040 0200

read: 0x 0040 0100

main: 0x 0040 0000

Text (code)

reserved

```
.globl   main
        .text
main:     mov       r0 #32              @ request 32 bytes space for object1 = new object();
          swi       0x12                @ r0 now contains pointer to the allocated space
          ldr       r1, =object1        @ (pseudo instruction) save the address at label: object1
          str       r0, [r1, #0]
          ldr       r1, =read           @ load pointer to read()
          str       r1, [r0, #0]        @ assign to object1
          ldr       r1, =print          @  load pointer to print()
          str       r1, [r0, #4]        @  assign to object1
          ldr       r0, =object1        @ get address of object1
          ldr       r0, [r0]
          ldr       r1, [r0, #0]        @ get address of read method
          blx       r1                  @ call read() by jump-and-link-register
          ldr       r0, =object1        @ get address of first object
          ldr       r0, [r0]
          ldr       r1, [r0, #4]        @  get address of print method
          blx       r1                  @  call the method
.data
object1:  .word    0                    @ declared data, known at compile time
```

```
@ read() method
@ Parameter: r0 == address of the object  (this)

        .text
read:
        mov    r3,r0                    @ save object's address to r3
        mov    r0, #1                   @ r0 = 1 print to stdout
        ldr    r1, =prompt               @ r1 = address of object's string
        swi    0x69

        add    r1, r3, #8               @ r1= address of buffer
        mov    r2, #24                  @ r2 = size of buffer
        mov    r0, #0                   @ r0 = 1 means to read from stdin
        swi    x6a

        mov    pc, lr                   @ return to caller

        .data
prompt:   .asciiz  "Enter data:"
```

```
@ print() method
@ Parameter: r0 == address of the object
        .text
print:
        add   r1, r0, #8               @ offset 8 bytes, r1 -> string buffer
        mov  r0, #1                    @ r0 = 1 means to print to stdout
        swi    0x69

        mov   pc,   lr
```