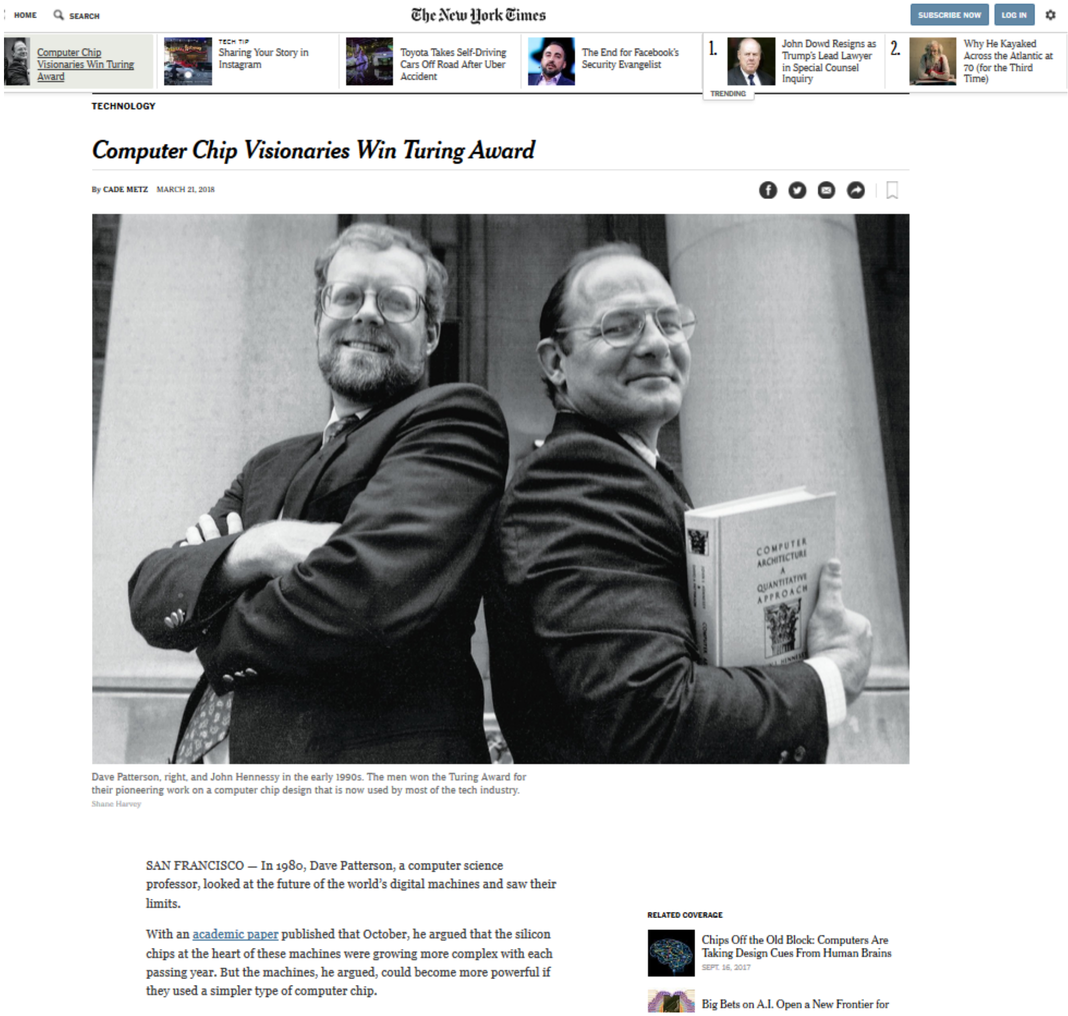
# ARM Assembly Programming

# Dynamic Data Structure
# &
# OOP



Computer Chip Visionaries Win Turing Award

TECH TIP Sharing Your Story in Instagram

Toyota Takes Self-Driving Cars Off Road After Uber Accident

The End for Facebook's Security Evangelist

1. John Dowd Resigns as Trump's Lead Lawyer in Special Counsel Inquiry

2. Why He Kayaked Across the Atlantic at 70 (for the Third Time)

TRENDING

TECHNOLOGY

## Computer Chip Visionaries Win Turing Award

By CADE METZ MARCH 21, 2018

Dave Patterson, right, and John Hennessy in the early 1990s. The men won the Turing Award for their pioneering work on a computer chip design that is now used by most of the tech industry.
Shane Harvey

SAN FRANCISCO — In 1980, Dave Patterson, a computer science professor, looked at the future of the world's digital machines and saw their limits.

With an academic paper published that October, he argued that the silicon chips at the heart of these machines were growing more complex with each passing year. But the machines, he argued, could become more powerful if they used a simpler type of computer chip.

RELATED COVERAGE

Chips Off the Old Block: Computers Are Taking Design Cues From Human Brains
SEPT. 16, 2017

Big Bets on A.I. Open a New Frontier for

# Note: The following is for the 32-bit ARM7, see Chapter 02_COD 4e ARM

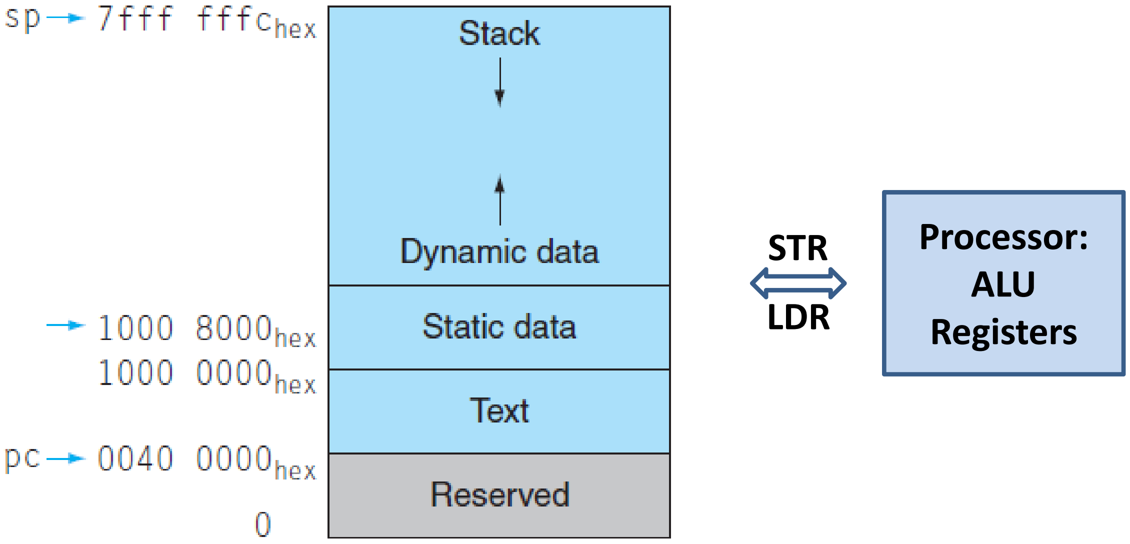


**FIGURE 2.13 Typical ARM memory allocation for program and data.** These addresses are only a software convention, and not part of the ARM architecture. The stack pointer is initialized to `7fff fffc`$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at `0040 0000`$_{hex}$. The static data starts at `1000 0000`$_{hex}$. Dynamic data, allocated by `malloc` in C and by `new` in Java, is next. It grows up toward the stack in an area called the heap.

Array initialization

In C language,

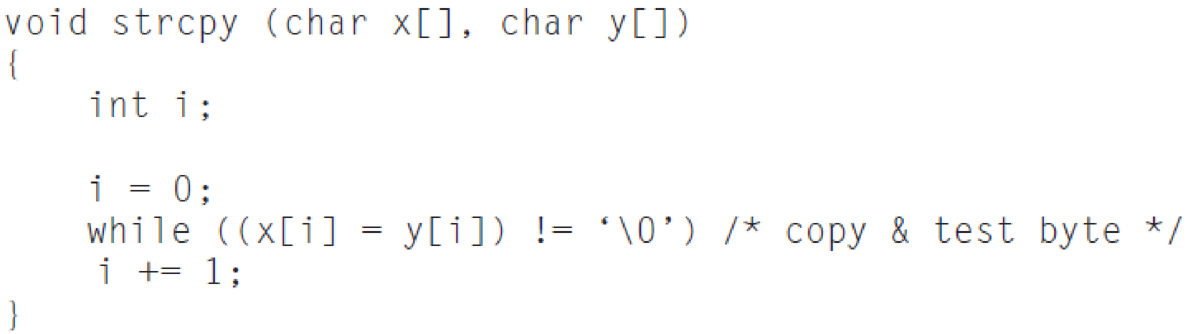
```
  int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  char pattern = “ould”;
  char pattern1 = {‘o’, ‘u’, ‘l’, ‘d’, ‘\0’};
```

In assembly,

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

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

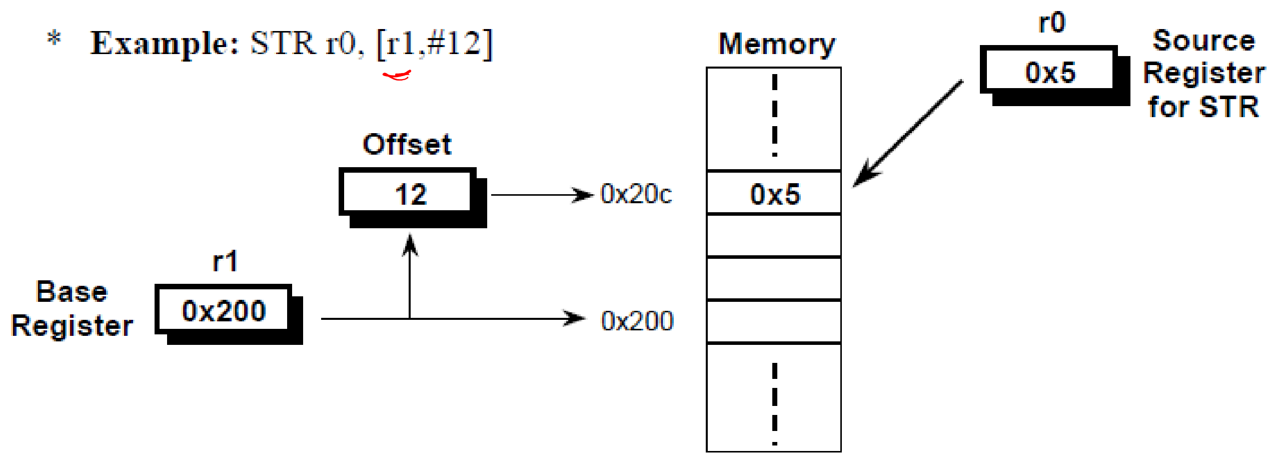
```
.text
  @mov r0, #345          @ see this number cannot be used as immediate value

  ldr r0, =0x12345678    @ 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
```



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

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

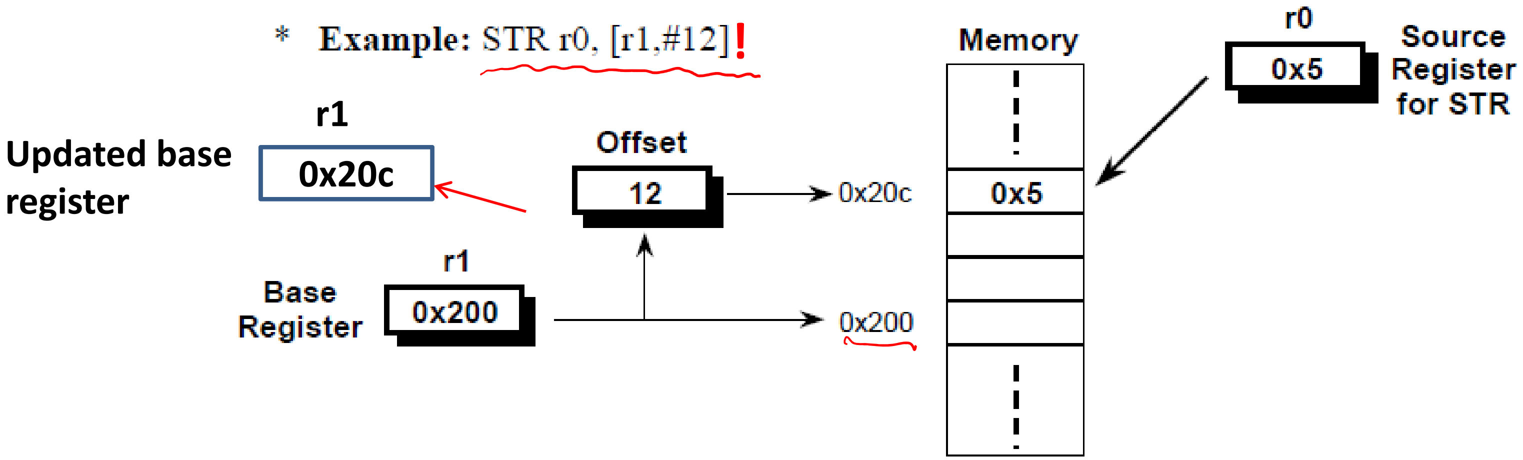
```
strcpy:  sub     sp,  #4
         str     r4, [sp, #0]
         mov     r4, #0
L1:      add     r2, r4, r1
         ldrsb   r3, [r2, #0]
         add     r12, r4, r0
         strb    r3, [r12, #0]
         beq     L2
         add     r4, r4, #1
         b       L1
L2:      ldr     r4, [sp, #0]
 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

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



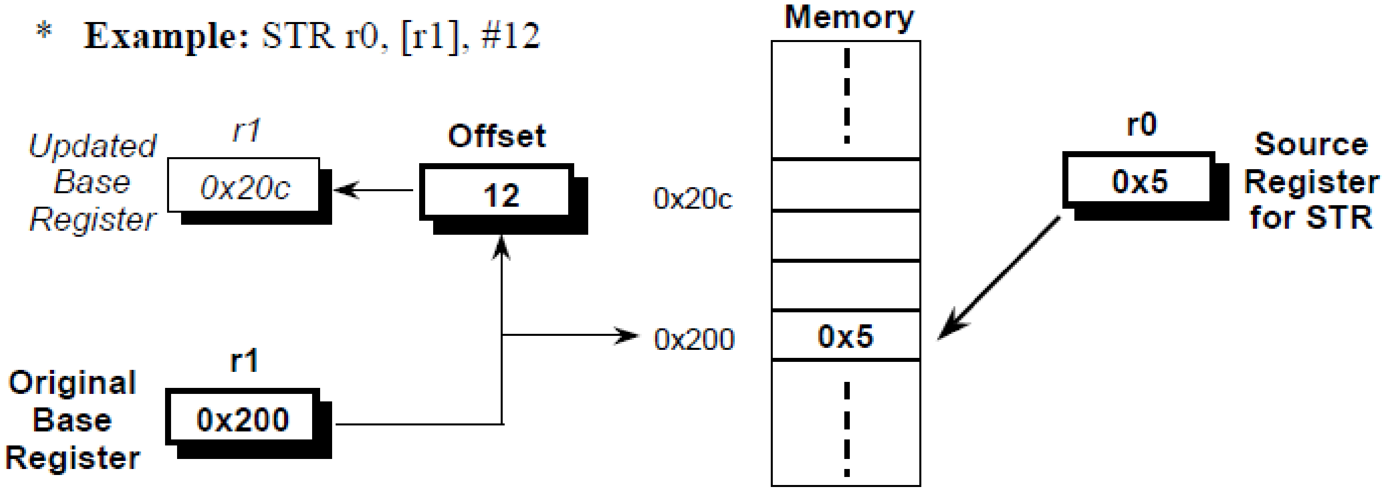
```
STR r0, [r1, r2, LSL #2]     @  address = r1 + 4 x r2
                             @ if r2 has value 3, this has the same effect of STR r0, [r1,#12].

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


STR r0 [r1], #12             @ post-indexing
```

# Pre-indexing



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

# Post-indexing



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

```
clear1:          @ r0 = pointer to array;
                 @ r1 = size
  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
```

```
clear1:          @ r0 = pointer to array;
                 @ r1 = size
  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;
}
```

```
clear1:              @ r0 = pointer to
array;
                     @ r1 = size
  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
```

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

# Arrays vs. Pointers

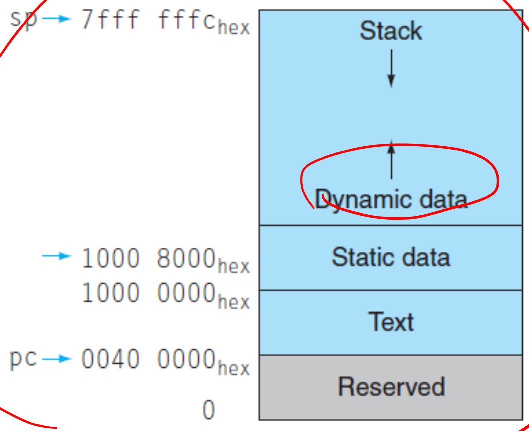
- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

# Comparison of Array vs. Ptr

- 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

# Dynamic Data Structures: linked-list, tree, …

# Dynamic memory allocation on the heap



In C language, we use

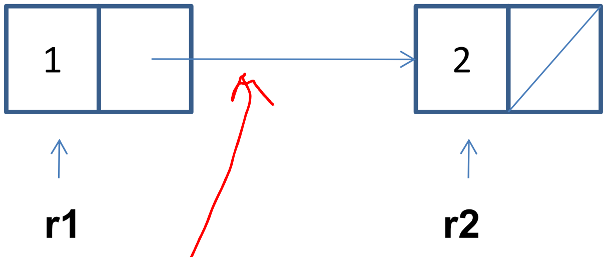
*malloc(unsigned, nbytes)

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

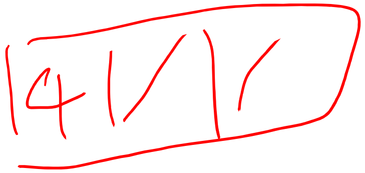
```
MOV     r0, #12  @ r0 = 12 bytes, the requested size
SWI     0x12     @ SWI instruction to request memory space from the heap
                 @ r0 contains the address of the allocated space.
```

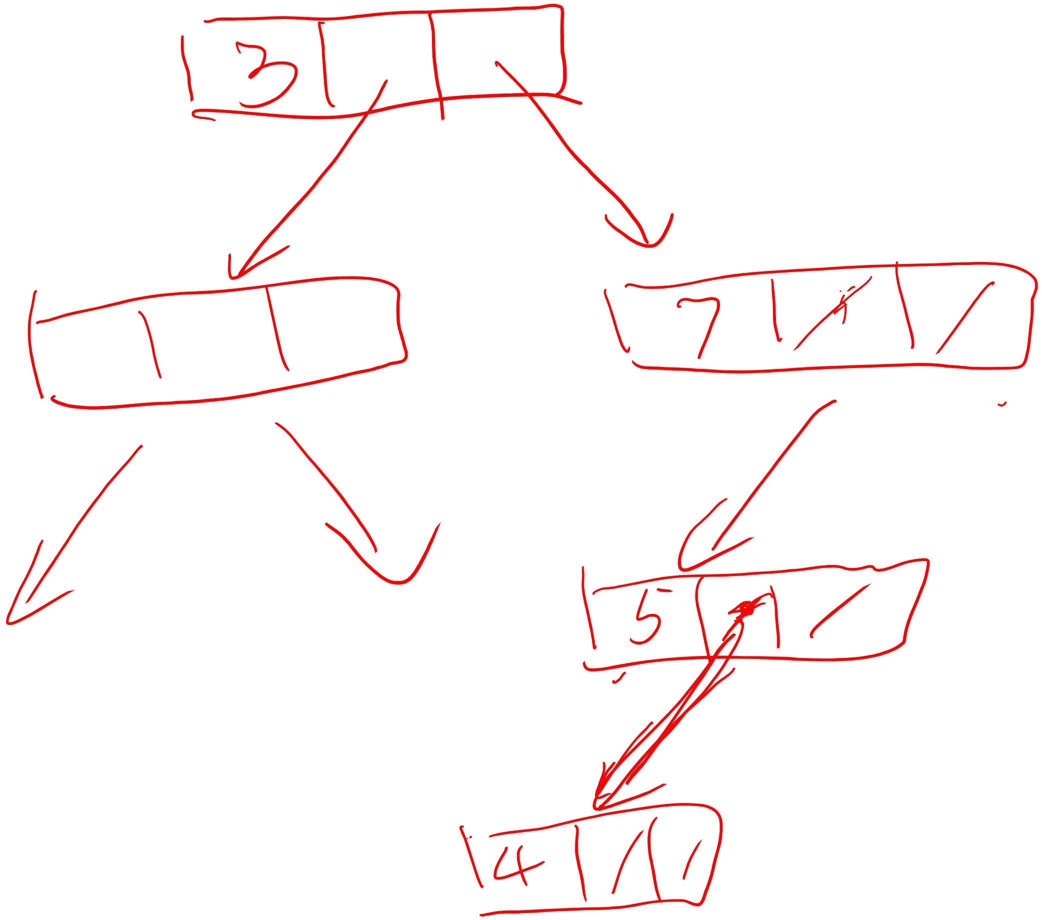
# Linked List

```
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]
```



mov r0, #12



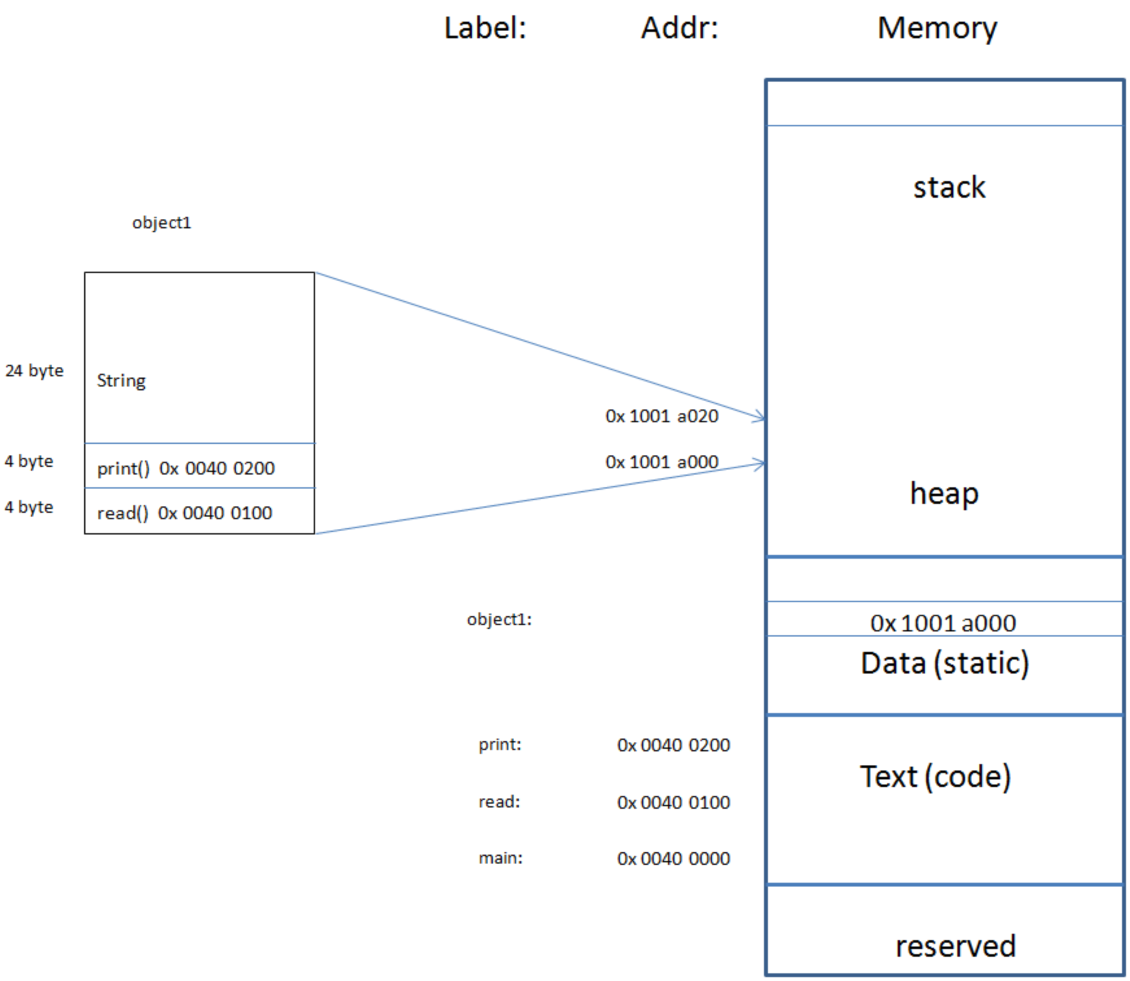


```
@ read integers from a file and insert them into a binary tree to get sorted
@ and print the sorted integers to the screen (stdout).

.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 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
OutFileName: .asciz "sorted_list.txt"
OutFileHandle: .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        @ 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 (pseudo)
          ldr       r0, [r0]
          ldr       r1, [r0, #4]        @  get address of print method
          blx       r1                  @  call the method
.data
object1:  .word   0
```

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

        .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
```