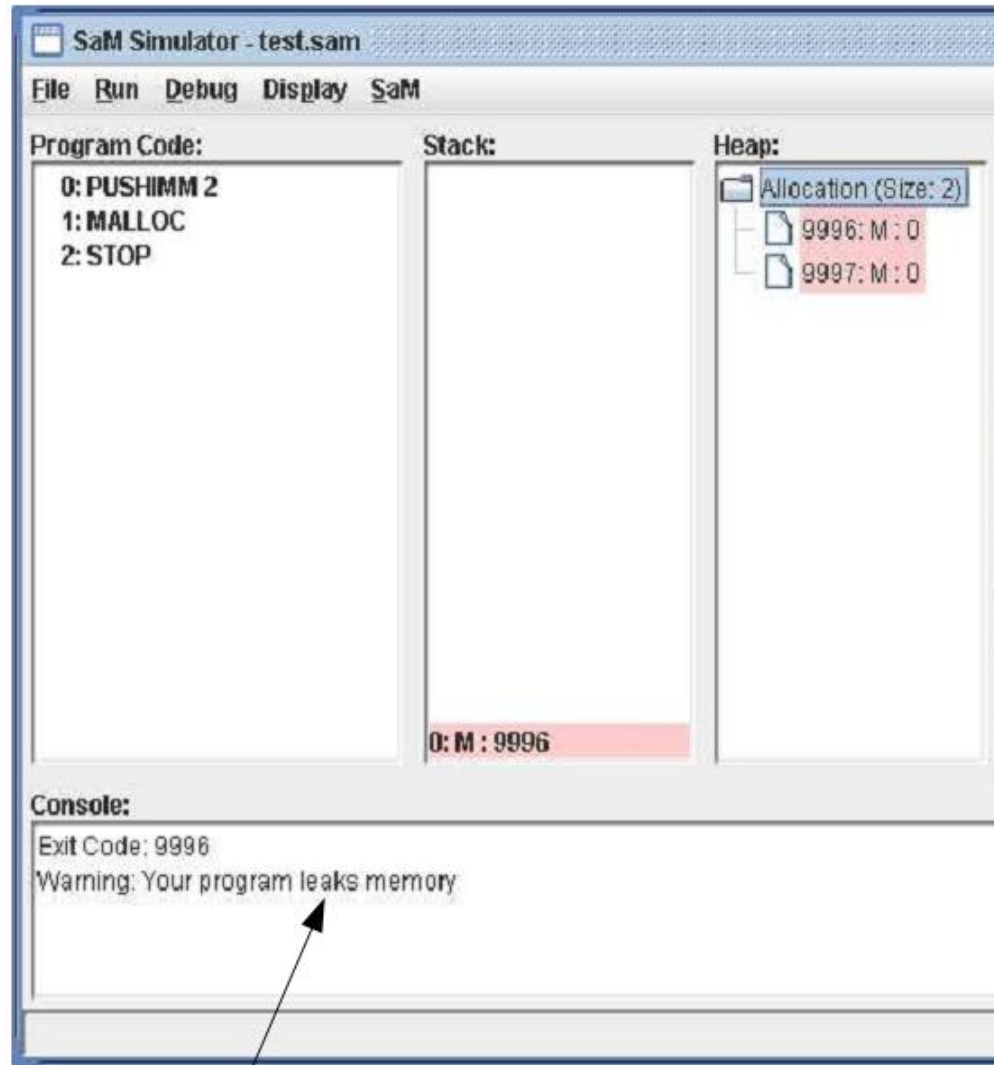


Pointers – Part 1

Dynamic Memory Allocation

What is a Pointer?

- Memory Regions
 - Stack
 - Heap
 - Program Code
- Each *memory location* has an *address*
- A “pointer variable” stores the *address* of a *memory location*



Pointer Operations

- Two types of pointer operations:
 - Star Operator
 - Ampersand Operator
- Star Operator (*)
 - “Value At”
 - De-reference
- Ampersand Operator
 - “Address Of”

Value-At Operator Example 1

Bali Code:

```
int main()  
  int *p;  
{  
  p = malloc(1);  
  *p = 3;  
  return *p;  
}
```

Value-At Operator Example 1 (SAM)

ADDSP 1	// Allocate local variables – note: pointer is allocated on the stack
PUSHIMM 1	// Push the first (only) parameter for the malloc function call
MALLOC	// Call Malloc ... Where is the cast?
DUP	
STOREOFF 2	// store the result to the local variable
ADDSP -1	// need to remove the “dup-ed” value
PUSHIMM 3	// push the constant
DUP	
PUSHOFF 2	// push the value of “p” – which is a memory location
SWAP	// need to swap them for correct parameter order to storeind
STOREIND	
ADDSP -1	// remove the “dup-ed” value
PUSHOFF 2	// want to “return *p;” so get the memory location that p points to
PUSHIND	// get the value at *p
STOREOFF -1	// set the return value
ADDSP -1	// remove the “dup-ed” value
RST	// end a subroutine!

Sample 1 - Executing

The screenshot displays a debugger interface with four main panels: Program Code, Stack, Heap, and Registers. The Program Code panel shows a list of assembly instructions, with instruction 12, `STOREIND`, highlighted. The Stack panel shows a list of stack frames, with frame 6, `M : 9996`, highlighted. The Heap panel shows a memory allocation of size 1 at address 9996, with a blue arrow pointing from the highlighted stack frame 6 to this allocation. The Registers panel shows the current values of the PC, FBR, and SP registers.

Program Code:

- 1: JUMP EtanBukiet_KarinH
- 2: ADDSP 1 (<= EtanBukiet
- 3: PUSHIMM 1
- 4: MALLOC
- 5: DUP
- 6: STOREOFF 2
- 7: ADDSP -1
- 8: PUSHIMM 3
- 9: DUP
- 10: PUSHOFF 2
- 11: SWAP
- 12: STOREIND
- 13: ADDSP -1
- 14: PUSHOFF 2
- 15: PUSHIND
- 16: STOREOFF -1
- 17: JUMP EtanBukiet_KarinH
- 18: ADDSP -1 (<= EtanBuki
- 19: RST
- 20: PUSHIMM 0 (<= EtanBu
- 21: LINK
- 22: JSR EtanBukiet_KarinH
- 23: UNLINK

Stack:

- 7: I : 3
- 6: M : 9996
- 5: I : 3
- 4: M : 9996
- 3: P : 23
- 2: M : 0
- 1: I : 0
- 0: I : 0

Heap:

- Allocation (Size: 1)
- 9996: M : 0

Registers:

- PC: 12
- FBR: 2
- SP: 8

Buttons:

- Open
- Step
- Run
- Capture
- Stop
- Reset

Heap Functions and Bali

- Malloc is the memory allocation function
- It get's you a “valid” memory location
- What is “Valid”?
 - Similar to the stack, in that when you add to the stack pointer you get a non-assigned stack position, so to here with the “heap”
- What is the parameter passed to malloc?
 - How many contiguous “memory locations” that the user wants.
 - Contiguous? = Next to each other, in a block
 - How big is a memory location? (we'll talk about this later with sizeof function)

Heap Functions and Bali – Part 2

- Why do we need to cast a malloc?
 - Want to catch more errors at compile time
 - Type Safety
 - Malloc returns a “void *”
 - Bali Programmer, for his or her own safety must ***explicitly*** do the cast to indicate that he or she knows the implications.

Heap Functions and Bali – Part 3

- Cleaning up the heap?
Why is there still an object on the heap after my program exits?
- Need to “free” memory that was allocated (malloc-ed) on the heap.
- Use “free” function in bali and “FREE” SAM instruction
- Similar to leaving extra values on the stack

Heap Functions and Bali – Part 4

- How big are variables?
- We know that all variables (int/boolean/char) that we've seen only take up a single stack location. Since Stack locations and memory locations are the same size, they also take up only one memory location.
- A pointer stack variable only takes up one stack location
- So does anything **not** have a size of “1”? – Yes, structures/classes – next week.

Casting from an Void* to an int

```
int IntFromVoidStar(void*v)
```

```
    int *a0; int **a1; int* a2;
```

```
{
```

```
    a0 = <int*>v;
```

```
    a1 = &a0;
```

```
    a2 = <int*> a1;
```

```
    return *a2;
```

```
}
```

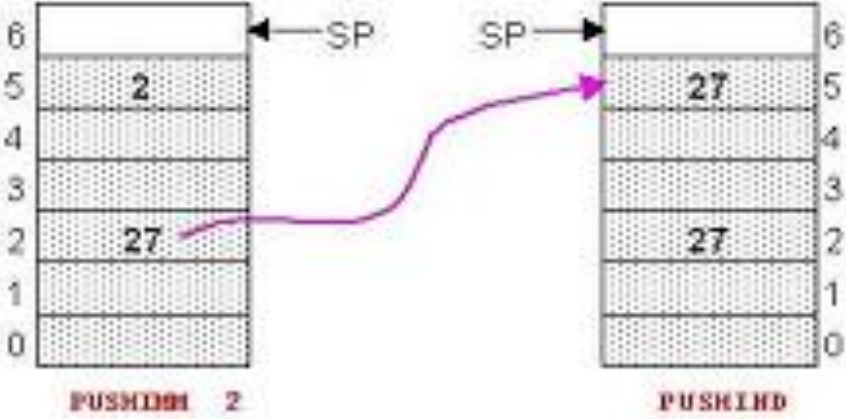
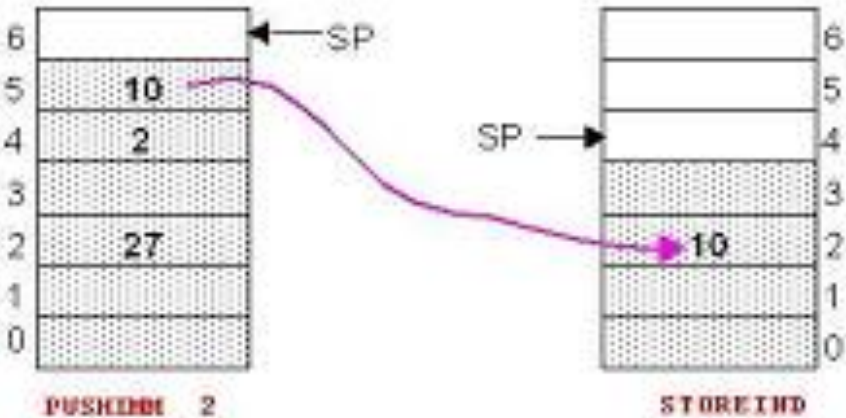
Casting from an Void* to an int (SAM)

```
ADDSP 3           // allocate local variables
PUSHOFF -1        // push the paramter "v"
STOREOFF 2        // put it in local variable (a0)
PUSHFBR          // compute &(a0) – It's local
PUSHIMMMA 2      // remember that a0 is at offset 2
ADD
STOREOFF 3        // put that result in a1
PUSHOFF 3         // get a1
STOREOFF 4        // and put its value in a2
PUSHOFF 4         // push a2
PUSHIND          // and get *a2
STOREOFF -2       // return that
ADDSP -3         // de-allocate local variables
```

Computing the Address of?

- Local Variable:
 - PUSHFBR
 - PUSHIMMMA (*offset*)
- Global Variable:
 - PUSHIMMMA (*offset*)
- Pointer Variable
- Eg: `int*p; &(*p)`
 - This is just “p” itself!
- The operand that the “&” is operating on behaves differently if it is in an “&” or not – this is similar to an expression on the Left Hand Side of an Assignment

PUSHABS using PUSHIND?

Instruction	Example	Demonstration
<p>PUSHIND</p> $V_{SP-1} \leftarrow V_{V_{SP-1}}$	<p>PUSHIMM 2</p> <p>PUSHIND</p>	 <p>The diagram illustrates the execution of the PUSHIND instruction. It shows two states of a stack with addresses 0 to 6. In the initial state, the stack pointer (SP) points to address 6. After the PUSHIMM 2 instruction, the value 2 is stored at address 5. The PUSHIND instruction then increments the stack pointer to 5 and stores the value at the new SP (27) into the memory location at address 5. A pink arrow indicates the data movement from address 2 to address 5.</p>
<p>STOREIND</p> $V_{V_{SP-2}} \leftarrow V_{SP-1}$ $SP \leftarrow SP - 2$	<p>PUSHIMM 2</p> <p>PUSHIMM 10</p> <p>STOREIND</p>	 <p>The diagram illustrates the execution of the STOREIND instruction. It shows two states of a stack with addresses 0 to 6. In the initial state, the stack pointer (SP) points to address 6. After the PUSHIMM 2 instruction, the value 2 is stored at address 5. The PUSHIMM 10 instruction then stores the value 10 at address 4. The STOREIND instruction then increments the stack pointer to 4 and stores the value at the new SP (2) into the memory location at address 2. A pink arrow indicates the data movement from address 5 to address 2.</p>

“Address of” Example

```
int main()
{
    int *p;
    p = <int*>malloc(1);
    print &p;
    print p;
    print &>(*p);
}
```

Malloc and Arrays