Pointers – Part 2

Review

Review

Why are we bothering with all this stuff about pointers, strings, and structs?

Review

What is a pointer? How does one declare an instance of one?

Review

In Bali, what sort of expression does the * operator act on, and what is the value of the resulting expression?

Review

In Bali, what sort of expression does the **&** operator act on, and what is the value of the resulting expression?

Review

What is the heap? What is the name of the operator one uses to reserve memory on the heap?

Review

In Bali, what does the **`malloc`** operator take in and yield?

Review

Suppose we have the fragment

```
int *p;
...
p = <int*> malloc(3 * sizeof(int))
```

Why the funny angle bracket thing?

Review

In Bali, what sorts of types can we cast between?

Review

In Bali, what is `void`? `void*`?

Review

In Bali, what is `null`?

Review

Suppose we have the fragment

```
int *p;
...
p = <int*> malloc(3 * sizeof(int))
```

What does p hold?

Review

Suppose we have the fragment

```
int *p;
...
p = <int*> malloc(3 * sizeof(int))
```

How do we get to the 1st slot that we allocated?

Review

Suppose we have the fragment

```
int *p;
...
p = <int*> malloc(3 * sizeof(int))
```

How do we get to the 2nd slot that we allocated?

Review

Suppose we have the fragment

```
int *p;
...
p = <int*> malloc(3 * sizeof(int))
```

How do we get to the 3rd slot that we allocated?

Review

What is pointer arithmetic?

Review

In Bali, what does the **`free`** operator take in and do?

Review

What is a memory leak?

Review

Suppose we have the fragment

```
int *p;
...
p = <int*> malloc(3 * sizeof(int))
```

What expression will return the three allocated slots to the system's memory?

Review

What's with the `sizeof(int)`? What does the sizeof operator take in and return? In Bali, what is the size of everything that you've dealt with so far?

Review

A string is an array of what? How is a string stored in memory?

For today

- More on pointer arithmetic
- Arrays
- Strings
- Structures
- P4C

Pointer arithmetic

Pointer arithmetic allows us to access memory locations relative to a pointer

Pointer arithmetic: addition/sutraction with ints

Suppose p is a pointer expression, and c is an integer expression.

(p ± c) is the memory address at

p ± c*sizeof(X)

where X is the type to which p points

Pointer arithmetic: addition of two pointers

The colorless green ideas are still sleeping furiously, if you were wondering.

Pointer arithmetic: subtraction of two pointers

Suppose p and q are pointer expressions of the same type.

(p - q) is the memory address at

(p - q)/sizeof(X)

where X is the type to which p and q point

Pointers as arrays

- Using malloc, you can allocate contiguous memory addresses.
- Allocating more than one “block” gives you a “chunk” that can be accessed using pointer arithmetic
- Always on the heap in our Bali

Array Example:

```
int main()
    int *array; int arrayLength; int i;
{
    arrayLength = 4;
    array = <int*> malloc(sizeof(int) * arrayLength);
    *(array+0) = 1;
    *(array+1) = 2;
    *(array+2) = 3;
    *(array+3) = 4;
    printArray(array, arrayLength);
}
int printArray(int *array, int arrayLength)
    int i;
{
    // TODO
}
```

Strings

- Special type of array
- A contiguous region containing characters where the last one is the “null character”
- SaM Instructions:
  - READSTR
  - PUSHIMMSTR
    Allocates the string on the heap, and returns the memory address of the first character
  - WRITESTR

Strings

- When working with strings, it is the Bali programmer's responsibility to free them
- The program

  int main() print “hello world”;

leaks memory. Why?

Structures

- Similar to arrays:
  - Fields are in contiguous memory locations
  - Name based access (ie: by fields) vs Numeral based access (ie: by index)
  - **Always on the heap**
  - Only need to free the “struct” and not non-pointer fields
- Fields are similar to local variables
  - Compiler keeps track of offsets

Structure definition examples

- Use the “structdef” keyword

```
structdef LinkedList {
    struct LinkedListNode* m_FirstNode;
    struct LinkedListNode* m_LastNode;
    int m_Count;
}
structdef LinkedListNode {
    struct LinkedListNode* m_Next;
    struct LinkedListNode* m_Previous;
    void* m_Value;
}
structdef LinkedListIterator {
    struct LinkedList* m_CurrentList;
    struct LinkedListNode* m_CurrentNode;
    boolean m_IsInitialized;
}
```

Using defined structures

- Recall that structures are always stored on the heap. What's the implication?
- Use the **->** operator to access members
- The sizeof operator, given a structure, will return how many members it has.

Using defined structures

```
structdef person {
        char *m_name;
        int m_age;
  }
  ...
  struct person* buildPersonFromInput()
        struct person *p;
  {
        p = <person *> malloc(sizeof(struct person));
        (p->m_name) = readString();
        (p->m_age) = readInt();

        return p;
  }
  int destroyPerson(struct person *p)
  {
        free (p->m_name);
        free p;
  }
```

Problems:

```
structudef matrix {
        int m_Rows, m_Cols; int **vals;
}
```

Write a method

```
int newMatrix(int n, int m)
```

that creates a n x m matrix.

Problems:

```
structudef matrix {
        int m_Rows, m_Cols; int **vals;
}
```

Write a method

```
int valueAt(struct matrix *m, int row, int col)
```

that retrieves the value at the specified location of m.

Problems:

```
structudef matrix {
        int m_Rows, m_Cols; int **vals;
}
```

Write a method

```
int destroyMatrix(struct matrix *m)
```

that returns all memory used by m to the system.

More problems

- Given also:
  structdef vector {
  int m_Length; int *m_vals;
  }
- Write a function that given a matrix gets a “row” vector with index i. (not a copy)
- Write a function that does “matrix multiplication” given 2 matrices. (square)
- Write a function that clones a matrix.