Memory Management

CS 241
Data Organization using C
Malloc (Memory Allocation)

Instructor: **Joel Castellanos**
e-mail: joel@unm.edu
Web: http://cs.unm.edu/~joel/
Office: Farris Engineering Center
Room 2110

10/8/2019

Quiz: What is the Output?

```
1)  #include <stdio.h>
2)
3)  static int a = 1;
4)  int b = 2;
5)
6)  void foo(int e)
7)  {
8)    static int c = 4;
9)    int d = 8;
10)   e = a | b | c | d | e;
11)   printf("%d, ", e);
12)   a = b = c = d = e = 0;
13) }
14)
15) void main(void)
16) { static int e = 16;
17)   foo(e);
18)   foo(e);
19) }
```

a) 31, 0,
b) 31, 16,
c) 31, 24,
d) 31, 28,
e) 31, 31,

Quiz: Substring Search

```
char *findSubstring(char *str, char *needle)
{
  int len = strlen(needle);
  int n = 0;
  while (*str)
  { if ( *(needle+n) == *str)
    { n++;
      if (n == len) return (str-n) + 1;
    }
    else
    { str -= n;
      n = 0;
    }
    str++;
  }
  return 0;
}
```

What is the output of:

```
printf("%s\n",findSubstring("mississippi","sip"))
a) 0
b) 6
c) 7
d) sip
e) sippi
```


<stdlib.h>: malloc()

`void *malloc(byteSize)`

The `malloc` function allocates unused space for an object whose size in bytes is specified by `byteSize`.

The order and contiguity of storage allocated by *successive* calls to `malloc` is unspecified.

The return value is a *void pointer* (not `void`) A void pointer is aligned so that it may be assigned to a pointer of any object type.

For example, on some machines `int x` can only start on addresses where `&x % 4 = 0`.

The pointer returned points to the start (lowest byte address) of the allocated space.

If the requested space cannot be allocated, a `NULL` pointer is returned.

<stdlib.h>: calloc()

`void *malloc(byteSize)`

Allocates a continuous block of **`byteSize`** bytes of memory.

`void *calloc(elmentCount, elementSize)`

Allocates a continuous block of memory large enough to hold **`elmentCount`** items each of **`elementSize`** bytes.

Also initializes all elements to zero.

```
int n = 50;
int *a = malloc(n * sizeof(int));
int *b = calloc(n,  sizeof(int));
```


<stdlib.h>: free()

`void free(void *ptr);`

The `free` function *deallocates* a block of memory previously allocated using a call to `malloc`, (or `calloc` or `realloc`).

After being deallocated, the memory is available for further allocations.

Always call `free` when done with every pointer returned by `malloc`.

Never use a pointer after it has been freed.

Never call `free` with any value other than one returned by `malloc` (or `calloc` or `realloc`).

Dynamic Array Allocation: main()

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define END_CODE INT_MIN

void main(void)
{
  int *num = atoiArray("42 427 3 1234 88");
  int i=0;
  while (num[i] != END_CODE)
  { printf("num[%d]=%d\n", i, num[i]);
    i++;
  }
  free(num);
}
```

int *num only allocates memory for a ***pointer to an* int**.

atoiArray must allocate memory for the actual array.

7

7

Dynamic Array Allocation: atoiArray 1 of 4

```c
int* atoiArray(char *line)
{
   //Given: line -> "42 427 3 1234 88"
   //1) Find n, the number of numbers are in line.
  //2) Allocate memory for an array of int of size n+1 (for the
        terminating symbol.
  //3) Convert the characters in the string to numbers in the
        array.
   //4) Add the terminating symbol.
   //5) Return a pointer to the allocated int array.
}
```

8

8

Dynamic Array Allocation: atoiArray 2 of 4

```
int* atoiArray(char *line)
{
  //1) Find n, the number of numbers are in line.
  int n=1; //Always need space for terminating value.
  char* charPt = line;
  while (*charPt)
  { if (*charPt == ' ') n++;
    charPt++;
  }
  printf("%d numbers in string.\n", n);

  //2) Allocate memory for an array of int of size n+1
  int *num = malloc(sizeof(int)*(n+1));
```

9

Dynamic Array Allocation: atoiArray 3 of 4

```
//3) Convert string to numbers in the array.
int* intPt = num;
*intPt = 0;
charPt = line; //reset to beginning of line
int startedNewNumber = 0;
while (*charPt)
{ if (*charPt == ' ')
  { printf("Done with: %d\n", *intPt);
    intPt++;
    *intPt = 0;
    startedNewNumber = 0;
  }
  else
  { *intPt = *intPt*10 + (*charPt-'0');
    startedNewNumber = 1;
  }
  charPt++;
}
```

10

Dynamic Array Allocation: atoiArray 4 of 4

```
  //4) Add the terminating symbol.
  if (startedNewNumber) intPt++;
  *intPt = END_CODE;

  //5) Return a pointer to the allocated int array.
  return num;

}
```

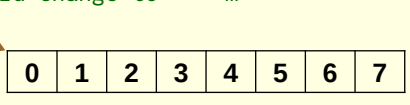

Fixed Size, 2D Array (print inner loop x)

```
void main(void)
{ int m[2][4];

  int* pt = &m[0][4];  //could change to  = *m

  for (int i=0; i<8;  i++)
  {
    *pt = i;
    pt++;
  }

  for (int y=0; y<4; y++)
  {
    for (int x=0; x<2; x++)
    { printf("x[%d][%d]=%d\n", x, y, m[x][y]);
    }
  }
}
```




```
x[0][0]=0
x[1][0]=4
x[0][1]=1
x[1][1]=5
x[0][2]=2
x[1][2]=6
x[0][3]=3
x[1][3]=7
```

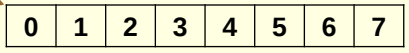
Fixed Size, 2D Array (print inner loop y)

```
void main(void)
{ int m[2][4];

  int* pt = &m[0][0];  //could change to  = *m

  for (int i=0; i<8;  i++)
  {
    *pt = i;
    pt++;
  }

  for (int x=0; x<2; x++)
  { for (int y=0; y<4; y++)
    {
      printf("x[%d][%d]=%d\n", x, y, m[x][y]);
    }
  }
}
```




```
x[0][0]=0
x[0][1]=1
x[0][2]=2
x[0][3]=3
x[1][0]=4
x[1][1]=5
x[1][2]=6
x[1][3]=7
```


Dynamically Sized, 2D Array with malloc

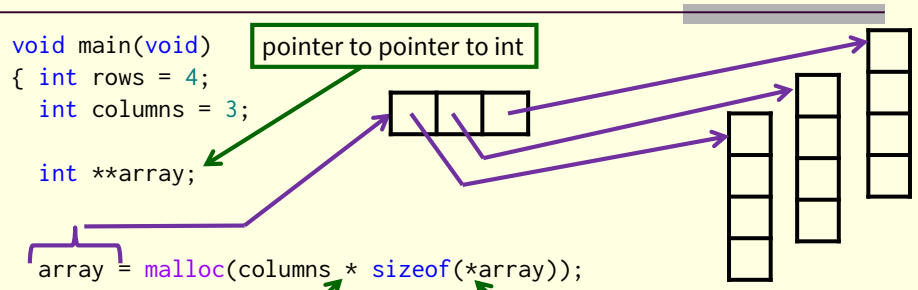
```
#include <stdio.h>
#include <stdlib.h>
void set2DArray(int **array, int columns, int rows)
{ //body
}

void print2DArray(int **array, int columns, int rows)
{ //body
}

void free2DArray(int **array, int columns)
{ //body
}

void main(void)
{ //body (calls malloc)
}
```

Dynamically Sized, 2D Array: `main`



```
void main(void)
{ int rows = 4;
  int columns = 3;

  int **array;

  array = malloc(columns * sizeof(*array));
  int x;
  for (x = 0; x<columns; x++)
  { array[x] = malloc(rows * sizeof(int));
  }
  set2DArray(array, columns, rows);
  print2DArray(array, columns, rows);
  free2DArray(array, columns);
}
```

pointer to pointer to int

Multiplication operator

(print inner loop x)

Dynamically Sized, 2D Array: `set2DArray`

```
void set2DArray(int **array, int cols, int rows)
{
  int x, y;
  for (x=0; x<cols; x++)
  {
    for (y=0; y<rows; y++)
    {
      array[x][y] = x*10 + y;
    }
  }
}
```

down one column on inner loop improves cache hits.

Every cell gets a different data value.

Dynamically Sized, 2D Array: print2DArray

```
void print2DArray(int **array, int cols, int rows)
{

  for (int y=0; y<rows; y++)
  {
    printf("row %d [", y);
    for (int x=0; x<cols; x++)
    {
      printf("%3d ", array[x][y]);
    }
    printf("]\n");
  }
}
```

Printing suffers cache misses because a full row must print on a line.

```
row 0 [  0  10  20 ]
row 1 [  1  11  21 ]
row 2 [  2  12  22 ]
row 3 [  3  13  23 ]
```

17

17

Dynamically Sized, 2D Array: free2DArray

```
void free2DArray(int **array, int columns)
{
  for (int x=0; x<columns; x++)
  { free(array[x]);
  }

  free(array);
}
```

```
valgrind a.out
==29599== Command: a.out
==29599== HEAP SUMMARY:
==29599==     in use at exit: 0 bytes in 0 blocks
==29599==   total heap usage: 4 allocs, 4 frees, 72 bytes allocated
==29599==
==29599== All heap blocks were freed -- no leaks are possible
```

18

18

Quiz: Segmentation Fault

When the code segment below is run, it results in a segmentation fault. On which line of code does this happen?

```
1) int *foo(int n, int myList[])
2) {
3)   int a = 0;
4)   int b = n / 2;
5)   int *c = myList + b;
6)   int *d = c + 5;
7)   int e = myList[b];
8)   return d + e;
9) }
```

a) Line 4
b) Line 5
c) Line 6
d) Line 7
e) Line 8

The Cause of Segmentation Fault

The ***ONLY*** cause of a segmentation fault is a program attempting to access memory (either a read or a write) when that access is blocked by the operating system.

```
1)  int *foo(int n, int myList[])
2)  {
3)    int a = 0;      <- Impossible to Seg Fault
4)    int b = n / 2;  <- Impossible to Seg Fault
5)    int *c = myList + b;  <- Impossible to Seg Fault
6)    int *d = c + 5;  <- Impossible to Seg Fault
7)    int e = myList[b];  <- Possible Seg Fault
8)    return d + e;  <- Impossible to Seg Fault
9)  }
```