Advanced Dynamic Memory

Chapter 4

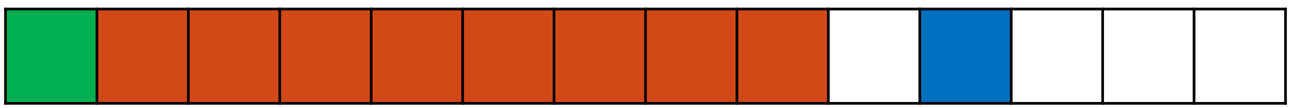
Pages 153-167

Topics

- Resizing Dynamic Arrays
- Returning a Dynamic Array From a Function
- 2D Dynamic Arrays
- Comparison of Dynamic vs Static Memory
- C Style Pass by Reference

Resizing Dynamic Arrays

- Often there is a need to add elements to an array but the array is not large enough to accommodate them



- Resize the array may not be possible in the heap due to conflicts with other allocations
  - Lets assume the figure above represents memory
  - White is unreserved memory
  - Each color is a unique allocation
  - Here it is not possible to grow the brown array by two because of the blue allocation

Resizing Dynamic Arrays

- There the standard practice is to perform a new allocation of the requested size and copy the contents.
- Resizing steps:

1. Allocate an array of the desired size
2. Copy the contents of the old array into the new array
3. Deallocate the old array
4. Optional: Point the pointer of the original array to the new allocation

- Step 4 allows you to use the original pointer/array name for future operations which is often more convenient

Example of Resize

```
int i, *org, *tmp;
org = new int[5];

// Fill with random number
for(i = 0; i < 5; i++)
   org[i] = rand() % 10;

// Now resize, Step 1: create an array of desired size
tmp = new int[6];
// Step 2: Copy contents from original
for(i = 0; i < 5; i++)
    tmp[i] = org[i];
// Step 3: Deallocate orignal array
delete [] org;
// Optional Step 4: point org to the new array
org = tmp;
```

Resizing Arrays

- Alternatively you can use the C++ copy function to do the copying for you

copy (pOrgArr, pOrgArr + size, pNewArr);

- pOrgArr: source array
- Size: number of elements to copy
- pNewArr: destination array

- Copy assumes pNewArr has already been allocated

Example Resize using Copy

```
int i, *org, *tmp;
org = new int[5];

// Fill with random number
for(i = 0; i < 5; i++)
org[i] = rand() % 10;

// Now resize, Step 1: create an array of desired size
tmp = new int[6];

// Step 2: Copy contents from original
copy(org, org + 5, tmp);

// Step 3: Deallocate orignal array
delete [] org;
// Optional Step 4: point org to the new array
org = tmp;
```

Returning a Dynamic Array

- Allocations occur in the heap not the stack
- As a result: allocations to the heap are not automatically deallocated on a function exit
  - This means it is possible to create a dynamic array in a function and allocate it
- To return an array, we can return a pointer to the dynamic memory

Example of Returning an Array

```
int* createRandomArray(int size)
{
   int i, *arr;
   arr = new int[size];

   // Fill with random numbers
   for(i = 0; i < size; i++)
     arr[i] = rand() % 10;

   return arr;
}
```

- Here memory is allocated to the pointer *arr*
- The contents of array *arr* is filled with random numbers
- The pointer is then returned to the calling function

Example of Returning an Array

```
int main ()
{
  int i, *myArr;

  // Create a random array and
  // store it in pointer myArr
  myArr = createRandomArray(5);

  // Print the contents
  for(i = 0; i < 5; i++)
    cout << myArr[i] << " ";

   // Need to deallocate it
   delete [] myArr;

   return 0;
}
```

- To use the function a pointer is created, *myArr*
- *myArr* will point to the memory that our *createRandomArray*() function returns
- Since the memory is allocated it will still exist until it is deallocated

2D Dynamic Arrays

- The new operator can only return a 1D array
  - There is no direct method to create a 2D dynamic array
- It is possible to emulate a 2D array through a series of 1D allocations

- Allocation Steps:

1. Create a pointer of pointers (**)
2. Allocate an array of pointers whose size is the desired number of rows
3. For each pointer element in the previous allocation create another allocation of the desired data type whose size is the number of desired columns

2D Dynamic Array of Ints

```
int i, **myArr;

// Step 1: Allocation of Rows
myArr = new int*[3];
```

- This creates an array of integer pointers with which we can perform further allocations

0x8008	myArr[0]=
0x8004	myArr[1]=
0x8000	myArr[2]=

Mem Addr	Variable
0x1000	**myArr = 0x8000
0x1004	i
The Stack	

2D Dynamic Array of Ints

```
int i, **myArr;

// Step 1: Allocation of Rows
myArr = new int*[3];

for(i = 0; i < 3; i++)
   myArr[i] = new int[2];
```

- For each element in the row an allocation of columns is created
- Here an array of size 3x2 is created

Memory View

Mem Addr	Variable
0x1000	**myArr = 0x8000
0x1004	i
The Stack	

0x8008	myArr[0]=0x5000
0x8004	myArr[1]=0x6000
0x8000	myArr[2]=0x5080

The Heap

0x5000	0x5004
myArr[0][0]	myArr[0][1]

0x6000	0x6004
myArr[1][0]	myArr[1][1]

0x5080	0x5084
myArr[2][0]	myArr[2][1]

2D Dynamic Array

- Notice the actual contents is kept in three separate arrays
  - Always equal to the number of rows
- The three arrays are not sequential to one another
  - Locations of the contents within the arrays is sequential
- Accessing elements within the array can be done using traditional array notation
- Since the array's rows are not sequential, there is likely to be a performance penalty because of processor cache architectures

2D Deallocation

- Occurs in reverse

- Deallocation Steps:
  1. Deallocate each row
  2. Deallocate the row itself

- At the end of the process there should be one *delete* for each *new*

2D Deallocation

```
// Step 1: Deallocate each row
for(i = 0; i < 3; i++)
   delete [] myArr[i];
```

Mem Addr	Variable
0x1000	**myArr = 0x8000
0x1004	i
The Stack	

0x8008	myArr[0]=0x5000
0x8004	myArr[1]=0x6000
0x8000	myArr[2]=0x5080

~~0x5000~~	~~0x5004~~
~~myArr[0][0]~~	~~myArr[0][1]~~

~~0x6000~~	~~0x6004~~
~~myArr[1][0]~~	~~myArr[1][1]~~

~~0x5080~~	~~0x5084~~
~~myArr[2][0]~~	~~myArr[2][1]~~

The Heap

2D Deallocation

```
// Step 1: Deallocate each row
for(i = 0; i < 3; i++)
   delete [] myArr[i];
// Step 2: Deallocate the row
// itself
delete [] myArr;
```

Mem Addr	Variable
0x1000	**myArr = 0x8000
0x1004	i
The Stack	

~~0x8008~~	~~myArr[0]=0x5000~~
~~0x8004~~	~~myArr[1]=0x6000~~
~~0x8000~~	~~myArr[2]=0x5080~~

The Heap

2D Deallocation

```
// Step 1: Deallocate each row
for(i = 0; i < 3; i++)
   delete [] myArr[i];
// Step 2: Deallocate the row
// itself
delete [] myArr;
```

Mem Addr	Variable
0x1000	**myArr = 0x8000
0x1004	i
The Stack	

The Heap

Static vs Dynamic Arrays

Topic	Static Arrays	Dynamic Array
Allocation:	Fast Allocation: Requires simply stack pointer addition	Slow Allocation: Requires accessing a lookup table
Performance after allocation:	Nearly identical for 1D Arrays, though dynamic arrays have a higher probability of a cache miss	
Memory Footprint:	Often required to be overly large to prevent out of bounds issue	Requires an additional pointer in memory. Otherwise sizes to demand
Flexibility:	Fixed size at compile time	Size determined at runtime
Stability:	Not possible to have a memory leak	Source of memory leak

- There are tradeoffs and which array should be used is dependent on the application

C Style Pass by Reference

- Since the aliasing operator (&) is not available in C pointers were used as an alternative
  - A pointer to memory in the stack was passed to the function
  - Since the function had addresses to memory outside the function it could directly modify those locations using dereference

- Due to time constraints this material will not be included in this class
  - If your interested read pages: 387 to 389
  - Or google C style pass by reference