Dynamic Memory Allocation
Nov 11, Ch 9 & 13.1

```
#include <stdio.h>
static char *ptr = "OKAY";

int main(int)
{
    ARG C.CHAR **A WHILE PRINTF("%c", (PTR){"n", *PTR++});

    THE SEGFAULT IN OUR CHAR STARS
```
Agenda

- Refresher on Pointers
- The Stack (Briefly)
- New and Delete on the Heap
- Examples and drawing them out
int x[ ] = {1, 2};
int *p = x;
*(p + 1) = 4;
*p = 3;
cout << x[0] << " , " << p[1];
The Stack - It’s Memory

- The stack is memory
- The place where all your variables are stored, and their values

```
Char f = 'F';
Int y[3] = {1,2,3};
Bool flag = False;
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Char f = ‘F’</td>
<td></td>
</tr>
<tr>
<td>Int y[0] = 1</td>
<td></td>
</tr>
<tr>
<td>Int y[1] = 2</td>
<td></td>
</tr>
<tr>
<td>Int y[2] = 3</td>
<td></td>
</tr>
<tr>
<td>Bool flag = False</td>
<td></td>
</tr>
</tbody>
</table>
• As soon as a function returns, its space on the stack goes away -- **any** variables created in the function go away if not returned.
So the stack exists... am I supposed to care?

Example:

Write a function called `makeRandomList` that does the following:

- Asks the user for how big they want their list to be
- Create that list with random numbers between 0 and 10
- Returns the list, and the size of the list
int* makeRandomList(int& size) {
    cin >> size;
    int list[size];
    for(int index = 0; index < size; index++)
        list[index] = rand() % 11;

    return list;
}
int main () {
    int list_size;
    int* rand_list = makeRandomList(list_size);
    for(int i = 0; i < list_size; list_size++)
        cout << rand_list[i] << endl;

    return 0;
}
SEGFAULTS

SEGFAULTS EVERYWHERE
Variables on the stack are temporary
Ideas for Solutions?

- Pass the array in
- Use a global variable for the array
- Do whatever we need the array for now

What do we really want?
The Heap -- Persistent Memory

- Section of memory unrelated to function calls
- When we put a variable on the heap instead of the stack, the variable will still be there even when the function ends
- Sounds like what we want for our array
How to get on the Heap-- New

Stack
-------
int x = 5;

Heap
---------
int* p = new int;
*p = 5;

What’s the difference?
New is asking at runtime for a chunk of memory for our variable.

This chunk is on the heap and will stick around (until we call delete)

New returns the memory address of the chunk of memory is creates for us
What holds memory addresses? Pointers!

You’re always going to have a pointer hold the result from `new`. 
Let's apply it: makeRandomList(int& size)

```c++
int* makeRandomList(int& size) {
    cin >> size;
    int * list = new int[size];
    for(int index = 0; index < size; index++)
        list[index] = rand() % 11;

    return list;
}
```
It worked! But we’re missing something

What happens to our list when we are done with it?
MEMORY LEAKS
MEMORY LEAKS EVERYWHERE
Memory Leaks

- Occurs when you dynamically allocate memory, but when it is no longer needed, but it is not released
- Performance impairment
- Ex: Longer running programs. Browsers, servers
Memory Leaks

- We need to release the memory we allocated-- need to take it off the heap

- Similar to how a function will “release” its variables on the stack when the function returns
How to get off the Heap -- **Delete**

Delete frees the memory we allocated using `new`:

```cpp
int *x = new int;
.
.
.
delete x; // Good! We free x here, no memory leak
return ...```

```cpp
int * p = new int;
// To delete p, which is a single variable:
delete p;

int * p2 = new int[5];
// To delete p2 which is an array:
delete[] p2;
```
EVERY NEW MUST HAVE A CORRESPONDING DELETE
Let’s apply it: makeRandomList(int& size)

```cpp
int main () {
    int list_size;
    int* rand_list = makeRandomList(list_size);
    for(int i = 0; i < list_size; list_size++)
        cout << rand_list[i] << endl;
    delete[ ] rand_list; // slightly different syntax for arrays
    return 0;
}
```
Examples and Diagramming

```c
int x = 4;
int* y = new int;
Int * z = new int;
*y = 8;
int *p1 = &x;
int *p2 = z;
int **p3 = &p1;
*z = 2;
x = *y;
*p1 = *p1 + *p2;
*p2 = x + 2;
**p3 = *p2 * 2;
*p3 = p2;
```
int x = 2;
int *y = new int;
*y = x + 2;
int *p1 = &x;
int *p2 = new int[*y];
int arr[] = {0,0,0};
int *arr2 = arr;
int *iter = p2;
for (int i = 0; i < *y; i++, iter++)
    *iter = i;
p2[1] = *p1;
arr = p2;
p2 = arr2;
int x = 5;
int * p1 = &x;
*p1 = *p1 + x;
cout << *p1 << endl;
cout << x << endl;