

# CMSC 341

## Lecture 2

### Dynamic Memory and Pointers

Prof. Park's section

# Today's Topics

Stack vs Heap

Allocating and freeing memory

`new` and `delete`

Memory Leaks

Valgrind

Pointers

Dynamic Memory and Classes

# Program Memory

The memory a program uses is typically divided into four different areas:

1. The `.text` section, where the executable code sits in memory.
2. The `.data/.bss` area, where global variables are stored.
3. The **stack**, where parameters and local variables are allocated from.
4. The **heap**, where dynamically allocated variables are allocated from.

# .data vs. Heap

- .data/.bss contains variables that are global or static
  - Things that exist over the entire lifetime of the program's execution
  - .bss is for uninitialized data (technically, initialized to 0)
  - The size of the data section is fixed
- Heap holds data that is dynamically allocated during execution
  - The heap can grow and shrink during the life of the program

# Stack vs Heap

What is stored:

- Stack is used to store:
  - Automatic variables: i.e., non-static local variables
  - Function arguments
  - Basically, the data relevant to a specific instance of a function call
  - Allows recursion (since each new call pushes a new “stack frame”)
- Heap stores things created with **new** or by calling **malloc()**
  - Must be accessed through pointers

# Stack vs Heap

## Space management:

- Stack space management is implicitly handled by the system/compiler:
  - grows automatically, every time a nested function/method is called
    - has an upper limit; heap growth must be requested
  - When a function returns, the stack frame is “deleted”
    - You *might* be still able to access it, but you *should* not!
- Heap space is managed by the programmer
  - Memory allocated with **new** must be cleared with **delete**; and **malloc ()** calls must have matching **free ()**
  - Failing to free heap space when done causes *memory leaks*

# Declaring Stack and Heap Variables

- Stack variable declaration

What you've been doing all along

```
int counter;  
double scores[10];
```

(Note: “static int foo” would not be on the stack: it's in data section)

- Heap variable declaration

Must use a pointer

```
int *values = new int[numVals];
```

(a technical point: the new array is in the heap, but the pointer `values` is actually still a stack variable)

# Allocating and Freeing Memory



# new and delete

Used to dynamically allocate and free memory on the heap

**new** must be assigned to a pointer variable

```
int *calendar = new int[12];
```

```
SomeClass *newItem = new SomeClass;
```

**delete** releases memory previously allocated with **new**

can only be used on pointer variables

```
delete newItem;
```

```
delete[] calendar;
```

# Good Programming Practices

- C++ does not have garbage collection
- After memory has been freed, set the pointer equal to **NULL**
  - Must be done after `delete` is called
  - Why do this?

# Memory Leaks

Occur when data is allocated, but not freed

Calling `new` over and over, but never `delete`

Not freeing new memory before exiting a function

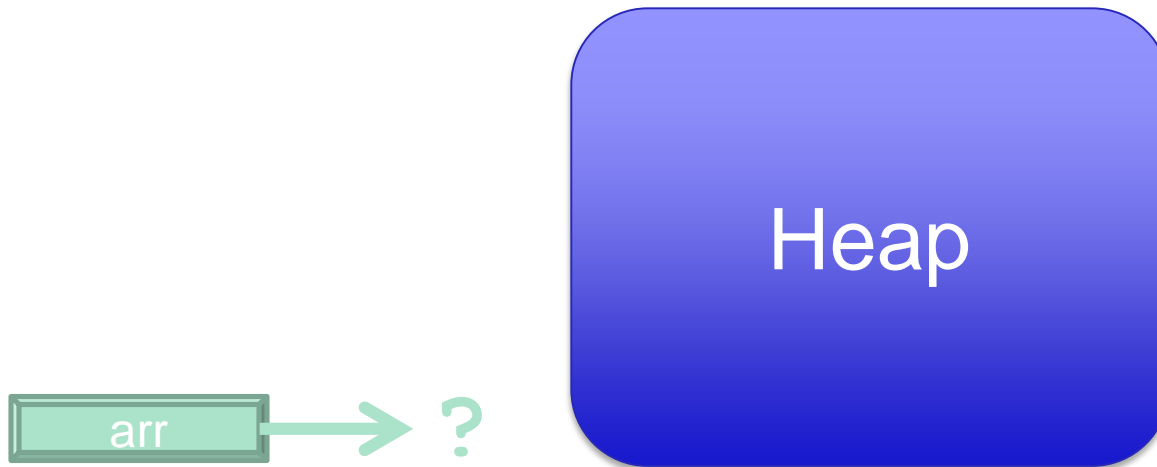
Access to the previous memory is lost

The location of that memory was overwritten

Eventually the program runs out of memory,  
and the program will crash

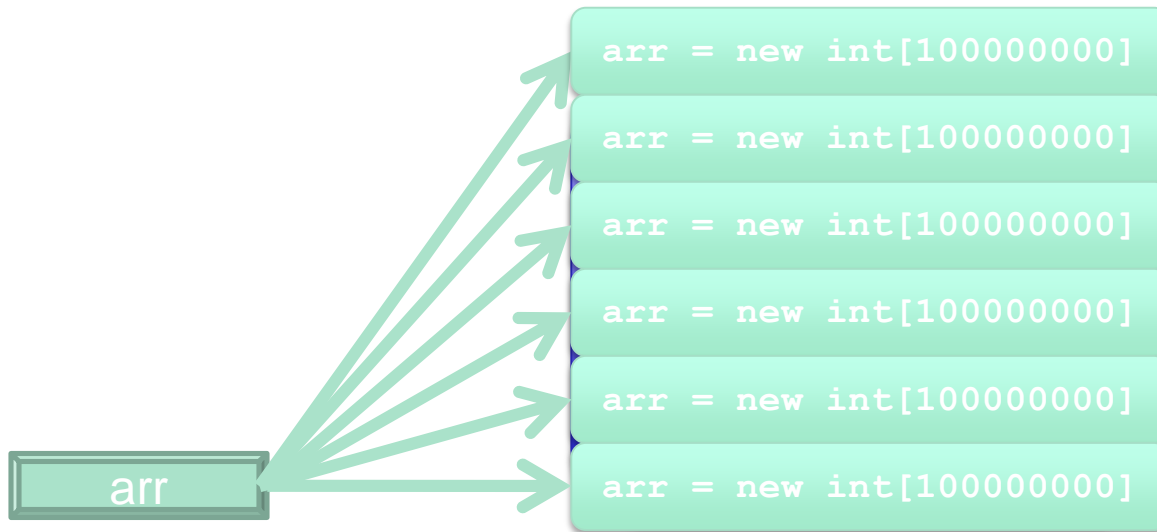
# Memory Leak Example

```
int *arr, var = 1000;  
for (int i = 0; i < var; i++) {  
    arr = new int[1000000000];  
}
```



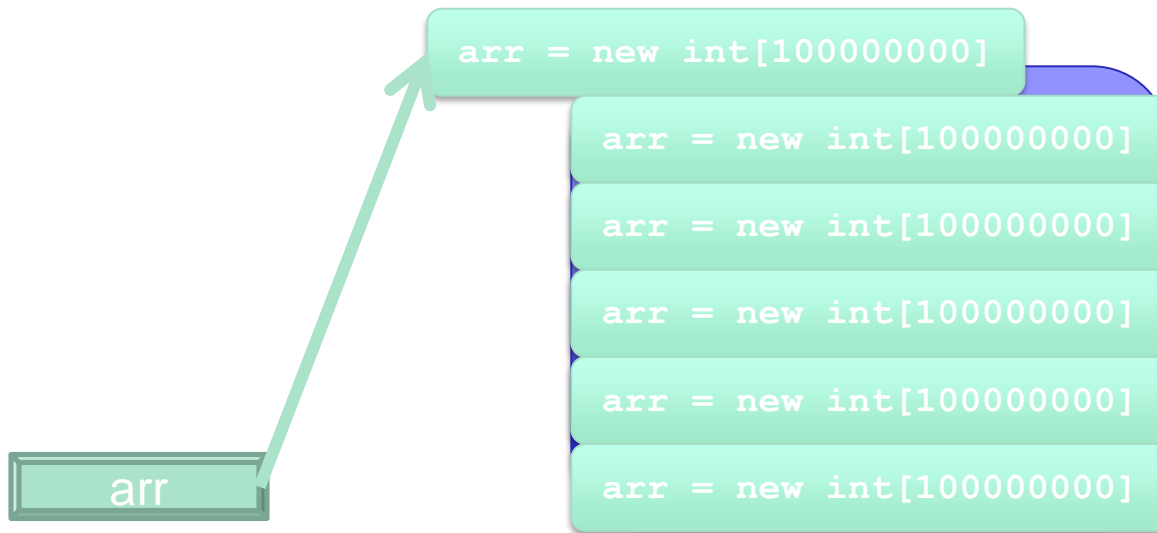
# Memory Leak Example

```
int *arr, var = 1000;  
for (int i = 0; i < var; i++) {  
    arr = new int[100000000];  
}
```



# Memory Leak Example

```
int *arr, var = 1000;  
for (int i = 0; i < var; i++) {  
    arr = new int[100000000];  
}
```



# Valgrind

Assists with dynamic memory management

Memory allocated using `new`

And therefore on the heap

Must compile with the `-g` flag (for debugging)

Detects memory leaks and write errors

Running valgrind significantly slows program down

program to run on

```
valgrind --leak-check=yes  proj1 arg
```

# Example `valgrind` Run – Code

```
#include <stdlib.h>

void f(void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;           // problem 1: heap block overrun
}                       // problem 2: memory leak--x not freed

int main(void)
{
    f();
    return 0;
}
```

Please note:  
This is C code, not C++.



# Example `valgrind` Run-Results 1

Describes problem 1 (heap block overrun)

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182== Address 0x1BA45050 is 0 bytes after a block
of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc
(vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

# Example `valgrind` Run-Results 1

Describes problem 1 (heap block overrun)

```
==19182== Invalid write of size 4
```

First line: type of error

```
==19182== at 0x804838F: f (example.c:6)
```

```
==19182== by 0x80483AB: main (example.c:11)
```

```
==19182== Address 0x1BA45050 is 0 bytes after a block  
of size 40 alloc'd
```

```
==19182== at 0x1B8FF5CD: malloc  
(vg_replace_malloc.c:130)
```

```
==19182== by 0x8048385: f (example.c:5)
```

```
==19182== by 0x80483AB: main (example.c:11)
```

Stack trace  
(read from bottom up)

# Example `valgrind` Run-Results 2

Describes problem 2 (memory leak)

```
==19182== 40 bytes in 1 blocks are definitely lost in  
          loss record 1 of 1  
  
==19182==      at 0x1B8FF5CD: malloc  
==19182==      by 0x8048385: f (a.c:5)  
==19182==      by 0x80483AB: main (a.c:11)
```

# Example valgrind Run-Results 2

Describes problem 2 (memory leak)

```
==19182== 40 bytes in 1 blocks are definitely lost in  
loss record 1 of 1  
==19182== at 0x1B8FF5CD: malloc  
==19182== by 0x8048385: f (a.c:5)  
==19182== by 0x80483AB: main (a.c:11)
```

First line: type of error

Stack trace tells you where the leaked memory was allocated (in function `f` on line 5 of file `a.c`)

Your program is **definitely** leaking memory!  
(May also see “probably,” “possibly,” or “indirectly.”)

# Pointers: Quick Review

*(Not meant to teach you the  
concept from scratch!)*

# Pointers

Used to “point” to locations in memory

```
int x;  
int *xPtr;  
x = 5;  
xPtr = &x; /* xPtr points to x */  
*xPtr = 6; /* x's value is 6 now */
```

Pointer type must match the type of the variable whose location in memory it points to

# Pointers – Ampersand

Ampersand ('&') returns the address of a variable

Asterisk ('\*') dereferences a pointer to get to its value (also used when initially declaring a pointer)

```
int x = 5, y = 7;
```

```
int *varPtr;
```

```
varPtr = &x;
```

```
*varPtr = 0;
```

```
varPtr = &y;
```

```
x = *varPtr;
```

# Examples – Ampersand and Asterisk

```
int x = 5;
```

```
int *xPtr;           [* used to
```

```
xPtr = &x;          [& used to
```

```
*xPtr = 10;         [* used to
```

```
cout << &xPtr;      [& used to
```



# Examples – Ampersand and Asterisk

```
int x = 5;
```

```
int *xPtr;           [* used to declare ptr]
```

```
xPtr = &x;          [& used to get address]
```

```
*xPtr = 10;         [* used to get value]
```

```
cout << &xPtr;      [& used to get address]
```

# Pointer Assignments

Pointers can be assigned to one another using =

```
int x = 5;
int *xPtr1 = &x; /* xPtr1 points
                  to address of x */
int *xPtr2; /* uninitialized */

xPtr2 = xPtr1; /* xPtr2 also points
               to address of x */

(*xPtr2)++; /* x is 6 now */
(*xPtr1)--; /* x is 5 again */
```

# NULL Pointers

NULL is a special value that does not point to any address in memory

It is a “non” address

Uninitialized pointers are like any new memory – they can contain *anything*

Setting a pointer to NULL will prevent accidentally accessing a garbage address (but dereferencing a null pointer will still give a segfault—that’s a Good Thing!)

# Pointer Visualization Exercise

```
int x = 5;
```

<b>variable name</b>	<b>x</b>		
<b>memory address</b>	0x7f96c		
<b>value</b>	5		

# Pointer Visualization Exercise

```
int  x = 5;  
int *xPtr = &x; /* xPtr points to x */  
int  y = *xPtr; /* y's value is ? */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	?

# Pointer Visualization Exercise

```
int x = 5;  
int *xPtr = &x; /* xPtr points to x */  
int y = *xPtr; /* y's value is ? */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	?

# Pointer Visualization Exercise

```
int x = 5;  
int *xPtr = &x; /* xPtr points to x */  
int y = *xPtr; /* y's value is ? */
```

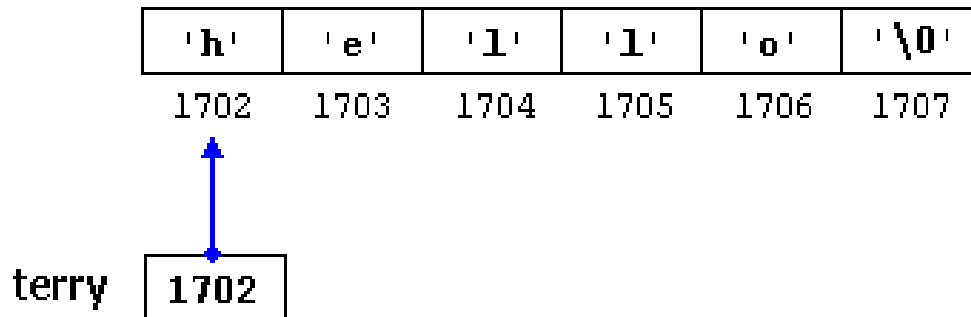
variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	5

# Pointers and Arrays

Arrays are built by pointers

Array name equivalent to address of first element

```
char terry[6] = "hello";
```





# Dynamic Memory and Classes

# Dynamically Allocating Instances

Stack:

```
Date today;
```

Heap:

```
Date *todayPtr = new Date(2016, 2, 7);
```

In both cases, constructor called (different versions, though)

# Dynamically Allocating Instances

Stack:

```
Date today;
```

nothing – handled for you

What to do when  
freeing memory?

Heap:

```
Date *todayPtr = new Date(2016, 2, 7);
```

call delete and set pointer to NULL

```
delete todayPtr;
```

```
todayPtr = NULL;
```

# Accessing Member Variables

Objects/structs (non-dynamic)

Use the “dot” notation

```
today.m_day = 2;
```

Heap (dynamic), or any other pointers

Use the “arrow” notation

```
todayPtr->m_year = 2015;
```

Shorthand for “dereference and use ‘dot’”

```
(*todayPtr).m_year = 2015;
```

# Passing Class Instances

## Stack

Normal variable; works as expected

```
cout << x;
```

## Heap

Need to dereference variable first

```
cout << xPtr;           // prints address
```

```
cout << *xPtr;         // prints value
```

# Destructor

All classes have a built-in destructors

Created for you by C++ automatically

Called when instance of class ceases to exist

Explicit `delete`, or end of program (`return 0`)

Classes can have member variables that are dynamically allocated

Built-in destructors do not free dynamic memory!

Must code one for the class yourself

# Coding Destructors

Named after class, and has no parameters

In source (.cpp file)

```
Student::~~Student() {  
    // free array of class name strings  
    delete classList; }  
}
```

In header (.h file)

```
~Student();    // denotes destructor
```

# Calling Destructors

## Stack

```
Student GV37486;
```

Automatically called at end of scope (function);

## Heap

```
Student *FY18223 = new Student();
```

Called only when memory is freed

```
delete FY18223; // destructor called  
FY18223 = NULL;
```



# Segmentation Fault FAQ

What is a segmentation fault (“segfault”)?

It happens when you access a memory address that is not legal (i.e., not in one of: data, heap, or stack)

Why is my program killed?

The operating system shows no mercy (real answer: few other options)

What causes a segfault?

An erroneous pointer

# Segmentation Fault FAQ (cont)

Does a bad pointer always cause a segfault?

No: if the bad pointer coincidentally points into some random-but-legal memory space, you will end up corrupting memory instead

Will dereferencing a pointer after deleting what it is pointing to cause a segfault?

Not necessarily; in fact, usually no; that is why we always set it to NULL

# Segmentation Fault FAQ (cont)

Why do you say a segfault is a Good Thing?

It's not good: just better than bad.

Assuming you already have a bad pointer, would you rather:

a) silently corrupt random memory; OR

b) have your program admit it's broken, and allow GDB to say exactly where?