Lecture 3b: Pointers and Dynamic Memory Allocation

CS 70: Data Structures and Program Development

Thursday, February 6, 2020

Learning Targets

1. I can write C++ code that uses `new` and `delete` (or `delete[]`) to allocate data on the heap.
2. I can write C++ code with no memory leaks or double deletes.
3. I can read and write C++ code that uses pointer arithmetic.
4. I can interpret and draw pictures of data involving local variables and heap memory.
5. I can explain differences between The Stack and The Heap

Heap Memory

What's wrong with this code?

```
void f() {
    size_t numElements;
    cin >> numElements; //wrong on the handout!
    int data[numElements];
}
```

Example: dynamically allocated array using `new`

```
void f() {
    size_t numElements;
    cin << numElements;
    int* data = new int[numElements];
}
```

Example: dynamically allocated array using `new`

```
void f() {
    size_t numElements;
    cin << numElements;
    int* data = new int[numElements];
}
```

- Keyword `new` allocates a chunk of memory on the heap
- Returns a *pointer*, the memory address of the heap memory
    - Used to access the memory on the heap

Pointers

Pointer Overview

- Each spot in memory has an associated *memory address*
  - A nonnegative integer that uniquely identifies that spot
  - In CS 70 we typically prefix a memory address on the heap with an `h`

Pointer Overview

- Each spot in memory has an associated *memory address*
  - A nonnegative integer that uniquely identifies that spot
  - In CS 70 we typically prefix a memory address on the heap with an `h`
- A *pointer* is a primitive type whose value is a memory address
  - The asterisk after a type name *in a declaration* denotes a pointer to something of that type
  - E.g., `int* myPtr`

Pointer Overview

- Each spot in memory has an associated *memory address*
  - A nonnegative integer that uniquely identifies that spot
  - In CS 70 we typically prefix a memory address on the heap with an `h`
- A *pointer* is a primitive type whose value is a memory address
  - The asterisk after a type name *in a declaration* denotes a pointer to something of that type
  - E.g., `int* myPtr`
- Access what is being pointed to by *dereferencing* the pointer
  - Use an asterisk in front of a pointer variable to dereference it
  - E.g., `*myPtr`

Pointer arithemetic

If `p` is a pointer to an `int`,

- `p+1` is a pointer to the following `int` in memory
- `p+2` is a pointer to the `int` after that in memory.

Pointer arithmetic

If `p` is a pointer to an `int`,

- `p+1` is a pointer to the following `int` in memory
- `p+2` is a pointer to the `int` after that in memory.

```
size_t size = 4;
int* myArray = new int[size];
for (size_t i = 0; i < size; ++i) {
    *(myArray + i) = 42;  // sets value of ith array elem
}
```

Accessing array elements on the heap

If `p` is a pointer to an `int`,

- `p+1` is a pointer to the following `int` in memory
- `p+2` is a pointer to the `int` after that in memory.

```
size_t size = 4;
int* myArray = new int[size];
for (size_t i = 0; i < size; ++i) {
    myArray[i] = 42;  // sets value of ith array element
}
```

Exercise: reasoning about types

```
int* myArray = new int[4];
```

For each expression below, what is the type of the result?

1. `myArray`
2. `myArray[1]`
3. `myArray + 1`
4. `*(myArray + 1)`

Stack memory addresses

Recall bracket vs. dereference syntax for arrays on the heap:

```
int* data = new int[2];
data[0] = 42;  // set value of first element
*(data + 1) = 24;   // set value of second element
```

Stack memory addresses

Recall bracket vs. dereference syntax for arrays on the heap:

```
int* data = new int[2];
data[0] = 42;  // set value of first element
*(data + 1) = 24;   // set value of second element
```

Parallels for arrays on the stack:

```
int stackData[2];
stackData[0] = 42;  // set value of first element
*(stackData + 1) = 24;   // set value of second element
```

Stack memory addresses

Recall bracket vs. dereference syntax for arrays on the heap:

```
int* data = new int[2];
data[0] = 42;  // set value of first element
*(data + 1) = 24;   // set value of second element
```

Parallels for arrays on the stack:

```
int stackData[2];
stackData[0] = 42;  // set value of first element
*(stackData + 1) = 24;   // set value of second element
```

What is the type of `stackData`?

Memory management with `new` and `delete`

Lifetime for data on the heap

```
// warning: this code has a memory leak
int f() {
    size_t size = 4;
    int* myArray = new int[size];
    for (size_t i = 0; i < size; ++i) {
        myArray[i] = 42;  // sets value of ith array elem
    }

    return 0;
}
```

Manual memory management

When we say `new`, the system

1. *allocates* an appropriate chunk of (previously free) space.
2. *initializes* that space by running the appropriate constructor.
3. returns the address of the newly initialized memory.

Manual memory management

When we say `new`, the system

1. *allocates* an appropriate chunk of (previously free) space.
2. *initializes* that space by running the appropriate constructor.
3. returns the address of the newly initialized memory.

When we say `delete` (`delete[]` for arrays), the system

1. takes an address (that came from `new`)
2. *destroys* the data at that address by running the appropriate destructor.
3. *deallocates* the data (records those bytes as free memory)

Lifetime for data on the heap

```cpp
// This code does not have a memory leak!
int f() {
    size_t size = 4;
    int* myArray = new int[size];
    for (size_t i = 0; i < size; ++i) {
        myArray[i] = 42;  // sets value of ith array element
    }

    delete[] myArray;

    return 0;
}
```

Example with `new` and `delete[]`

```
int main() {
    size_t size = 4;
    int* myArray = new int[size]{0}; // example initializ
    delete[] myArray;
    return 0;
}
```

What's wrong with this code?

```cpp
void makeArray(size_t size) {
    int* data = new int[size];
    cout << "Created array of size " << size << endl;
}
int main() {
    size_t dataSize = 2;
    makeArray(dataSize);
    delete[] data;
    return 0;
}
```

What's wrong with this code?

```
void makeArray(size_t size) {
    int* data = new int[size];
    cout << "Created array of size " << size << endl;
}
int main() {
    size_t dataSize = 2;
    makeArray(dataSize);
    delete[] data; //data is no longer in scope!
    return 0;
}
```

What's wrong with this code?

```cpp
int main() {
    int* data = new int[2]{42,42};
    delete[] data;
    data[0] = 70;
    delete[] data;
    return 0;
}
```

What's wrong with this code?

```
int main() {
    int* data = new int[2]{42,42};
    delete[] data; //The array is deallocated here!
    data[0] = 70; //(often this will still run!)
    delete[] data; //(usually this will halt execution)
    return 0;
}
```

One `delete` for every `new`

Heap data is allocated & initialized by `new`

Heap data is destroyed & deallocated by `delete`

- Unlike Java, there's no Garbage Collector!

One `delete` for every `new`

Heap data is allocated & initialized by `new`

Heap data is destroyed & deallocated by `delete`

- Unlike Java, there's no Garbage Collector!

Advantage: precise control over memory usage

One `delete` for every `new`

Heap data is allocated & initialized by `new`

Heap data is destroyed & deallocated by `delete`

- Unlike Java, there's no Garbage Collector!

Advantage: precise control over memory usage

Disadvantage: potential for serious mistakes

- *Memory leak*: forget to call `delete`
- *Double delete*: destroy & deallocate data multiple times.
- *Dangling pointer*: destroy & deallocate data, then use it.

Heap Allocation with arrays vs. non-arrays

Examples

```
int* nPtr = new int{5};
int** nPtrPtr = new int*{nPtr};
```

Examples

```
int* nPtr = new int{5};
int** nPtrPtr = new int*{nPtr};

Cow* cowPtr = new Cow{4, 9};
```

Examples

```
int* nPtr = new int{5};
int** nPtrPtr = new int*{nPtr};

Cow* cowPtr = new Cow{4, 9};

nPtr = new int{98};
```

Draw memory after this code executes:

```
int   n = 7;
int*  q = new int {12};
int*  r = new int {5};
int** s = new int* {q};

*s = r;
r = q;
**s += 1;
```

Also: What `delete`s do we need?

Draw memory after this code executes:

```
int   n = 7;
int*  q = new int {12};
int*  r = new int {5};
int** s = new int* {q};

*s = r;
r = q;
**s += 1;

delete *s;
delete s;
delete r;
```

Summary: Two Forms of `new`

```
// The "new" operator
Cow* bessie = new Cow;
Cow* bessie = new Cow{4, 9};
```

vs.

```
// The "array-new" operator
Cow* barn = new Cow[10];
```

Summary: Two forms of `delete`

Summary: Two forms of `delete`

```
Cow* bessie = new Cow{4, 9}
```

Summary: Two forms of `delete`

Summary: Two forms of `delete`

```
Cow* bessie = new Cow{4, 9}

delete bessie;  // Destroy & deallocate
```

Summary: Two forms of `delete`

Summary: Two forms of `delete`

```
Cow* bessie = new Cow{4, 9}
```

```
delete bessie;  // Destroy & deallocate
```

Arrays allocated with array-new must be deallocated with array-delete (!)

```
Cow* barn = new Cow[NUM_COWS];
...
delete[] barn;   // Destroy & deallocate
```

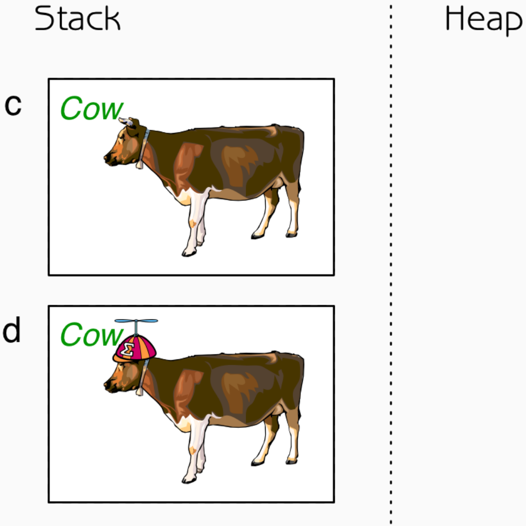
Stack vs. Heap in C++ vs. Java

Objects in C++

Cow c;

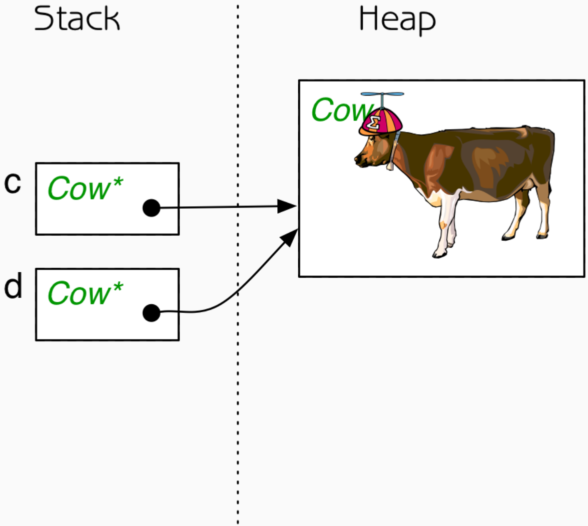
Cow d = c;

d.addHat();



Pointers to objects in C++

```
Cow* c = new Cow;
Cow* d = c;
(*d).addHat();
```



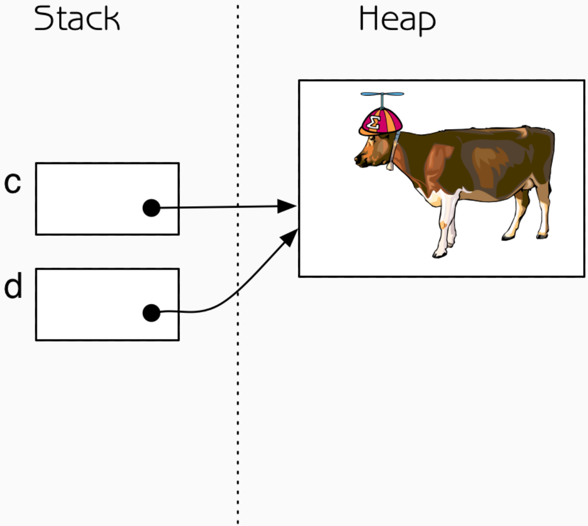
Objects in Java

```
// This is Java!

Cow c = new Cow();

Cow d = c;

d.addHat();
```



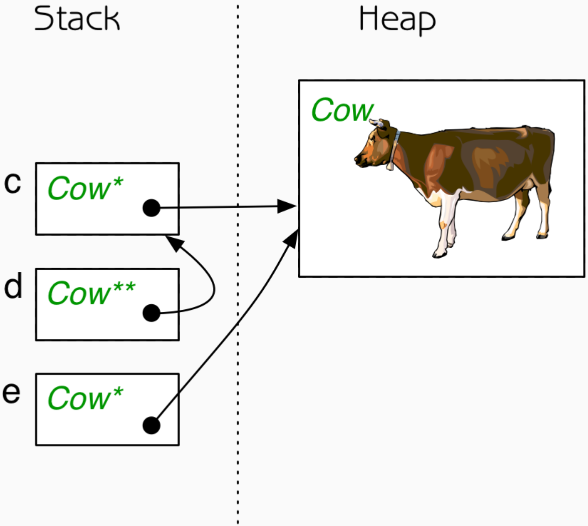
More about memory addresses

Address-of operator

*Cow** c = **new** *Cow*;

*Cow*** d = &c;

*Cow** e = &(*c);



Extra practice with `delete` and `delete[]`

Exercise 1

```
int main() {
    int* myInt = new int{0};
    const size_t myConstant = 4;

    Cow myCows[myConstant];

    // TODO: AVOID MEMORY LEAK
}
```

Exercise 2

```cpp
Cow** makeCowPtrArray(size_t n) {
    Cow** cowPtrArray = new Cow*[n];
    for (size_t i = 0; i < n; ++i) {
        cowPtrArray[i] = new Cow;
    }
    return cowPtrArray;
}

int main() {
    Cow** myCows = makeCowPtrArray(4);

   // TODO: AVOID MEMORY LEAK
}
```