Dynamic Memory in C++

CSE 1325

Alex Dillhoff

University of Texas at Arlington

No More *alloc

In C++, static variables are created on the stack just like they are in C.

For heap storage, the calls `malloc`, `calloc`, and `realloc` are gone.

Intead, C++ offers `new` and `delete`.

Allocating Heap Storage

Consider the following code:

```
double *a = new double[100];

for (int i = 0; i < 100; i++) {
    a[i] = i;
}

delete a;
```

Allocating Heap Storage

The `new` operator will allocate the requested amount of memory for an object or collection.

The equivalent of `free` in C++ is the `delete` operator.

What About `realloc`?

If we have some collection, it is important to be able to resize it based on our dynamic requirements.

Since we can use many C functions in C++, why not call `realloc`?

It is bad practice! We have access to the `vector` class.

The `vector` Class

As mentioned before, the `vector` class hides the responsibility of memory management from the developer.

Elements can be inserted and removed without worrying about explicitly reallocating memory.

The vector Class

Consider the following example.

```
vector<double> v;

for (int i = 0; i < 100; i++) {
    v.push_back(i);
    cout << v[i] << endl;
}
```

The `vector` Class

There are two things to note in this example:

1. Elements are added through the use of the class function `push_back()`.
2. The memory needed to store the new element is automatically allocated internally.

Pointers and References

Pointers are useful in that we can refer to *where* the data is instead of having to copy it.

However, pointers came with additional responsibility:

- What if the pointer is `NULL`?
- Does the pointer refer to a type that is different than what is expected?
- Is the syntax really that readable?

Pointers and References

C++ provides an answer for these questions in the form of a *reference*.

Just like a pointer, a reference refers to a particular object in memory.

Pointers and References

Unlike a pointer, a reference

- allows accessing using the same syntax as the original variable (no `f(&x)`),
- always refers to the same object that it was initialized to,
- and cannot refer to a null reference.

Pointers and References

Example: `pointer_vs_reference.cpp`

The Pitfalls of Heap Memory

For collections, it is safer to use a standard library implementation such as `vector` which handles memory management for you.

We will look at other collection classes later on.

The Pitfalls of Heap Memory

When will we need to use `new` and `delete`?

In modern C++, it is not recommended to use "naked" `new` and `delete`.

That is, using them without any sort of *garbage collection*.

Smart Pointers

C++ offers several ways of safely handling pointers.
When we look at classes, we will see how internal properties can be managed through the *constructor* and *destructor*.

For any other object, C++ offers a useful tool called the **smart pointer**.

Smart Pointers

It is difficult, especially in large programs, to keep track of the many allocations and pointer references.

Programmers are human and will often make mistakes.

The creator of C++ understood this and implemented a class which manages the references to a pointer.

Smart Pointers

What are *smart pointers*?

A *smart pointer* is a class designed to help manage objects allocated on the heap.

The `memory` C++ library implements two versions:

1. `unique_ptr` - represents unique ownership.
2. `shared_ptr` - represents shared ownership.

Smart Pointers

Example: unique_ptr_example.cpp

Smart Pointers

A `unique_ptr` controls the lifetime of the objects that are allocated on the heap.

When passing in between functions, the pointer is simply moved.

In the previous example, there was no need to explicitly call `delete` when we were done with the object.

Smart Pointers

Example: unique_ptr_function.cpp

Shared Pointers

A `shared_ptr` is similar to a `unique_ptr` in that it manages the lifetime of an object.

However, a `shared_ptr` is copied rather than moved.

This means that the object will persist in heap storage for as long as there is a `shared_ptr` to it.

Shared Pointers

Example: shared_ptr_example.cpp

Smart Pointers

General Advice for Memory Allocations in C++:

1. Don't use the heap for the sake of using the heap: use scoped variables when possible.
2. If using the heap, use a *manager object* such as a class constructor and destructor, or a smart pointer.