The Heap and Destructors

Dynamic memory

Michael Burrell

May 2, 2024

Readings

Chapter 12 — 12.2 (*not* 12.1)

Objectives

- Learn how objects can be stored in two different regions of memory
- Solidify our understanding of the call stack
- Learn how to allocate objects on the heap
- Learn how to deallocate objects from the heap

Problems with the stack

- We have already seen some limitations with the call stack
- Notably, we cannot return a pointer to something in the current stack frame

```
int *foo() {
    int x = 5;
    return &x; // undefined behaviour!! bad news!!
}
```

Motivations for the heap

- The call stack is managed for us automatically
    - New stack frame is created when we call a function
    - Current stack frame is destroyed when we return
- This is convenient, but limiting
- The heap is less convenient, but offers total freedom

Heap mechanism

- The heap mechanism is simple
  - Nothing happens implicitly
  - We must be explicit about all memory management in the heap
- Memory is created explicitly (using the `new` keyword)
- Memory is freed explicitly (using the `delete` keyword)

A note about Python

- Python uses the heap, as well, for objects
- However, Python has a feature called *automatic garbage collection*
    - In Python, memory does not need to be freed
- C++ does not have this feature, and memory management is done manually

Allocating a single object

- When allocating a single object (*not* an array), use the `new` keyword following by a constructor or type name
- In C++, even primitive data types can be constructed this way

```
int *returns_pointer_to_five() {
    int *r = new int;
    *r = 5;
    return r;
}
```

Explanation of code

```
int *returns_pointer_to_five() {
    int *r = new int;
    *r = 5;
    return r;
}
```

- The local variable `r` is allocated in the local stack frame
- However, it is pointing to a region of memory (big enough to hold an `int`) in the heap
- That memory in the heap will *not* be deallocated when this function returns
- We can return a pointer to this newly-allocated integer

Allocating an array

- Allocating an array on the heap is done similarly
- There is extra square bracket syntax needed to set the size of the array
- We get back a pointer to the first element of the array

```
int *x = new int[2];
x[0] = 5;
x[1] = 10;
```

Allocating arrays on the heap

- For large arrays or arrays where the size isn't known beforehand, we usually allocated on the heap
- On many operating systems, the call stack is of a limited size (often only a few megabytes)
  - The heap is big enough to store several billion gigabytes (limited by your computer's available RAM)
- For this reason, it's a common C++ idiom to allocate arrays on the heap unless they're small

Finally we can return an array!

```
int *nums_up_to(int max) {
    int *r = new int[max];
    for (int i = 0; i < max; i++) {
        r[i] = i;
    }
    return r;
}
int main() {
    int *array = nums_up_to(10);
    cout << array[5] << endl; // will print out "5"
    return 0;
}
```

Deallocating

- The code that we've seen so far in these slides has *memory leaks*
- A *memory leak* is when memory is allocated on the heap but never deallocated
  - It is especially bad if memory is allocated within a loop
  - The program's memory usage will keep increasing
- All memory allocated on the heap *should be* deallocated when it's no longer necessary

Example

```
int *nums_up_to(int max) {
    int *r = new int[max];
    for (int i = 0; i < max; i++) {
        r[i] = i;
    }
    return r;
}
int main() {
    int *array = nums_up_to(10);
    cout << array[5] << endl; // will print out "5"
    delete [] array;  // this memory is not needed any
        more!
    return 0;
}
```

Delete syntax

There are two forms of `delete`

`delete x` — when `x` points to memory that was allocated as a single object

`delete [] x` — when `x` points to memory that was allocated as an array

If you use the wrong one, your program will (probably) crash!

Dangling pointers

- The two most common mistakes when working with the heap are

  memory leak — when memory is not freed when it's not used any more

  dangling pointer — when memory *is* freed before it's finished being used

Dangling pointer example

```
int *x = new int;
*x = 5;
delete x;
cout << *x << endl; // undefined behaviour!! bad news!!
```

Constructing and destroying objects

- Constructing and destroying objects is simple if you are comfortable already with allocating and deallocating primitive data

```
class student {
    string name;
public:
    student(string name = "") : name(name) {}
};
int main() {
    student *s = new student("Joe Student");
    delete s;
    return 0;
}
```

Default constructor

- Unlike in Python, the default constructor in C++ can be called without any parentheses

```
student *s = new student;
delete s;
```

Destructors

- Things get more complicated when an object has a pointer to somewhere on the heap
- We have seen how to destroy the object itself, but not how to deallocate everything that the object is referring to

```
class student {
    double *grades; // what happens to the memory this is
        pointing to?????
public:
    student(int num_grades = 5) : grades(new
        double[num_grades]) {}
};
```

Destructors

- To get around this problem, C++ has the concept of *destructors*
- Destructors are *not* called explicitly
- They are called *implicitly* when an object *falls out of scope*
    - Stack-allocated objects will have their destructors called when their stack frame is destroyed
    - Heap-allocated objects will have their destructors called when the object is `deleted`

Example managing array

```
class student {
    string name;
    double *grades;
    size_t num_grades;
public:
    student(string name = "", size_t num_grades = 5)
        : name(name), grades(new double[num_grades]),
            num_grades(num_grades) {}
    ~student() {  // destructors start with a ~ tilde and
        cannot have parameters
        delete [] grades; // now we won't have a memory
            leak!
    }
};
```

Rule of 3

- The *Rule of 3* is a convention followed in C++ object-oriented programming
  - Note that it is specific to C++ and not applicable to other OO languages
- It is used to manage memory in the heap
- It says "if any of the following 3 are implemented, then <u>all</u> 3 should be implemented":
  1. destructor
  2. copy constructor
  3. assignment operator

Rule of 3 example

```
class pointer_to_int {
    int *data;
public:
    pointer_to_int() : data(new int) {} // default
        constructor
    ~pointer_to_int() {   // destructor
        delete data;
    }
    pointer_to_int(pointer_to_int const &o)
        : data(new int) { // copy constructor
        *data = *o.data;
    }
    pointer_to_int &operator=(pointer_to_int const &o) {
        // assignment operator
        *data = o.data;
    }
```

Rule of 3

- As we see more complex data types, we will get more practice with the Rule of 3
- We have not used the copy constructor or assignment operator yet, but we will
- In all 3 methods, be sure to deallocate/allocate any memory necessary

Conclusion

- We can allocate memory on the stack or in the heap
- The heap is more free, and can store more memory
- The heap in C++ requires explicit allocation and explicit deallocation
- We can now create objects which are not bound to any particular stack frame