Dynamic Memory in C++

The "new" operator

We've already seen the "`new`" operator when creating object pointers. This operator is also used to allocate any kind of dynamic memory in C++. If you are writing a C++ application, it is proper to use "`new`" instead of "`malloc`".

Allocating memory

Here's the format for using new to allocate space for single (non-object) elements:

```
type *name = new type;
```

For example, we could allocate space for an int:

```
int *intPtr = new int;
```

This works almost exactly like `malloc` – new allocates space for the type, and returns a pointer to the spot in memory. The integer itself is not given any initial value – it has whatever garbage value was left in that position.

We can now use `intPtr` like any other pointer. For example, we could update the contents of the integer to 17:

```
*intPtr = 17;
```

C++ arrays

We can also use the new operator to allocate space for dynamic arrays. Here's the format:

```
type *arrayName = new type[size];
```

This format is very similar to how arrays are created in Java. The only difference is that the array is declared as a pointer. For example, we could create an array of 10 integers:

```
int *nums = new int[10];
```

None of the array elements will be given initial values (unlike Java). If there is not enough space to do the allocation, either an exception will be thrown or `NULL` will be returned (depending on the compiler).

We can access these array elements just like we did arrays in C. For example, the following code will initialize every element in `nums` to 0:

```
int i;
```

```
for (i = 0; i < 10; i++) {
     nums[i] = 0;
}
```

There is still no way to determine the length of an array in C++. If you pass one of these arrays to a function, you will probably need to pass its length as well.

Creating objects

We have already seen how to use the "new" operator to create object pointers. For completeness, let's review. Suppose we have the following `Person` class:

```
class Person {
     private:
          string name;
          int age;
     public:
          Person(void);
          Person(string, int);
};

Person::Person(void) {
     name = "John";
     age = 20;
}

Person::Person(string name, int age) {
     this->name = name;
     this->age = age;
}
```

We can create a default `Person` pointer like this:

```
Person *p1 = new Person;
```

This line allocates space for the object and calls the no-argument constructor (setting the person's name to John and age to 20). Or, we can create a specific `Person` pointer like this:

```
Person *p2 = new Person("Jill", 25);
```

This line allocates space for the object and calls the constructor with arguments, which sets the name to "Jill" and the age to 25.

The "delete" operator

Like C, C++ has no garbage collector, which means we must release dynamic memory when we are done using it. The C++ operator to release dynamic memory is "delete".

"delete" should be used to release any memory allocated with "new", while "free" should still be used to release any memory allocated with "malloc", "calloc", or "realloc". The consequences of mixing these operators vary by compiler, but the results are never good.

Releasing dynamic memory

Suppose we have a dynamically allocated integer:

```
int *num = new int;
*num = 4;
```

Now we are done using the pointer. To deallocate space, we do:

```
delete num;
```

This is the same format for releasing space for single variables like structs.

Deallocating arrays

We also call delete to release space for arrays, but the format is a bit different. For example, suppose we allocated an array of integers:

```
int *arr = new int[10];
```

Now we are done using the array. To release space for every element in the array, we do:

```
delete[] arr;
```

If you leave off the [], it will not release all of the memory for the array. With some compilers, deleting an array as if it were a single variable will cause a segmentation fault.

Delete with objects

We already discussed how delete works with object pointers, but we will discuss it again for completeness. Consider the Set class below:

```
class Set {
     private:
          int *nums;
          int pos;
          int max;
     public:
```

```
            Set(int);
            ~Set();
            void  add(int);
};

Set::Set(int max) {
     nums = new int[max];
     pos = 0;
     this->max = max;
}

Set::~Set() {
     delete[] nums;
}

void Set::add(int elem) {
     int i;
     if (pos >= max) return;
     for (i = 0; i < pos; i++) {
          if (nums[i] == elem) return;
     }
     nums[pos++] = elem;
}
```

Now, suppose we have a pointer to a `Set` object:

```
Set *s = new Set(3);
s->add(1);
s->add(2);
s->add(3);
```

We are done using our `Set` pointer, so we do:

```
delete s;
```

This has the same format as releasing space for a single variable, but it does more:

- Calls the `Set` destructor
- Releases space for the array instance variable
- Releases space for the `Set` object itself

Dynamic memory pitfalls

Here is a list of potential errors when dealing with dynamic memory:

- Do not mix `new`/`delete` with `malloc`/`calloc`/`realloc`/`free`. If you allocate memory using a C function, release it with `free`. If you allocate with `new`, release it

with `delete`. Mixing and matching can cause memory corruption and segmentation faults.

- Do not call "`delete`" on the same pointer more than once. The first call will release the memory, but the second call can either release other memory or cause a segmentation fault.

Multi-dimensional dynamic arrays

We can use the new operator to allocate multi-dimensional arrays in C++. However, we still have to allocate space for an array of pointers and then allocate space for each row.

Here's how to allocate space for a 5x8 array of integers:

```
int rows = 5;
int cols = 8;
int i;

int **matrix = new int*[rows];
for (i = 0; i < rows; i++) {
     matrix[i] = new int[cols];
}
```

Now we can use matrix like a normal two dimensional array. For example, we can do:

```
matrix[1][2] = 3;
```

When we're done using a dynamic multi-dimensional array, we must first release the memory for each row:

```
for (i = 0; i < rows; i++) {
     delete[] matrix[i];
}
```

Then, we must release the memory for the array of pointers:

```
delete[] matrix;
```

Just like in C, there must be a corresponding call to `delete` for every call to `new`. The calls to `delete` should be made in reverse order of the calls to `new`.

Stack example in C++

Let's do a bigger example that demonstrates dynamic memory in C++. We will write a stack application that consists of the following files:

- `node.h` – represents a node in a stack

- `stack.h` and `stack.cpp` – defines the stack itself
- `main.cpp` – tests the stack

Here is the node implementation:

```
//node.h
#ifndef NODE_H
#define NODE_H

class Node {
     public:
          int data;
          Node *next;
          Node(int d) {data = d; next = NULL;}
};

#endif
```

Here is the stack implementation:

```
//stack.h
#ifndef STACK_H
#include "node.h"

class Stack {
     private:
          Node *top;
     public:
          Stack(void);
          ~Stack();
          void push(int);
          int pop(void);
};

#endif
```

****

```
//stack.cpp
#include "stack.h"
#include "node.h"

Stack::Stack(void) {
     top = NULL;
}

Stack::~Stack() {
```

```
        while (top != null) pop();
}

void Stack::push(int val) {
        Node *newnode = new Node(val);
        if (top == NULL) top = newnode;
        else {
                newnode->next = top;
                top = newnode;
        }
}

int Stack::pop(void) {
        if (top == NULL) throw "Stack is empty";
        else {
                int data = top->data;
                Node *temp = top;
                top = top->next;
                delete temp;
                return data;
        }
}
```

And here is a main file that uses the stack:

```
//main.cpp
#include "stack.h"

int main() {
        Stack *s = new Stack;
        s->push(5);
        s->push(4);
        s->push(3);

        delete s;
        return 0;
}
```

Let's consider what happens at each step in the main function. When we do:

```
Stack *s = new Stack;
```

It allocates space for the `Stack` object and calls the `Stack` constructor. Then, for each:

```
s->push(5);
```

It calls the `push` function, which creates space for the new node and adds it to the top of the stack. Finally, when we have:

```
delete s;
```

It calls the destructor for the stack. This destructor pops every node off the stack, which also releases memory for each node. Finally, space for the `Stack` itself is released.