19.1 — Dynamic memory allocation with new and delete

learncpp.com/cpp-tutorial/dynamic-memory-allocation-with-new-and-delete/

The need for dynamic memory allocation

C++ supports three basic types of memory allocation, of which you've already seen two.

- **Static memory allocation** happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
- **Automatic memory allocation** happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary.
- **Dynamic memory allocation** is the topic of this article.

Both static and automatic allocation have two things in common:

- The size of the variable / array must be known at compile time.
- Memory allocation and deallocation happens automatically (when the variable is instantiated / destroyed).

Most of the time, this is just fine. However, you will come across situations where one or both of these constraints cause problems, usually when dealing with external (user or file) input.

For example, we may want to use a string to hold someone's name, but we do not know how long their name is until they enter it. Or we may want to read in a number of records from disk, but we don't know in advance how many records there are. Or we may be creating a game, with a variable number of monsters (that changes over time as some monsters die and new ones are spawned) trying to kill the player.

If we have to declare the size of everything at compile time, the best we can do is try to make a guess the maximum size of variables we'll need and hope that's enough:

```
char name[25]; // let's hope their name is less than 25 chars!
Record record[500]; // let's hope there are less than 500 records!
Monster monster[40]; // 40 monsters maximum
Polygon rendering[30000]; // this 3d rendering better not have more than 30,000
polygons!
```

This is a poor solution for at least four reasons:

First, it leads to wasted memory if the variables aren't actually used. For example, if we allocate 25 chars for every name, but names on average are only 12 chars long, we're using over twice what we really need. Or consider the rendering array above: if a rendering only uses 10,000 polygons, we have 20,000 Polygons worth of memory not being used!

Second, how do we tell which bits of memory are actually used? For strings, it's easy: a string that starts with a \0 is clearly not being used. But what about monster[24]? Is it alive or dead right now? Has it even been initialized in the first place? That necessitates having some way to tell the status of each monster, which adds complexity and can use up additional memory.

Third, most normal variables (including fixed arrays) are allocated in a portion of memory called the **stack**. The amount of stack memory for a program is generally quite small -- Visual Studio defaults the stack size to 1MB. If you exceed this number, stack overflow will result, and the operating system will probably close down the program.

On Visual Studio, you can see this happen when running this program:

```
int main()
{
    int array[1000000]; // allocate 1 million integers (probably 4MB of memory)
}
```

Being limited to just 1MB of memory would be problematic for many programs, especially those that deal with graphics.

Fourth, and most importantly, it can lead to artificial limitations and/or array overflows. What happens when the user tries to read in 600 records from disk, but we've only allocated memory for a maximum of 500 records? Either we have to give the user an error, only read the 500 records, or (in the worst case where we don't handle this case at all) overflow the record array and watch something bad happen.

Fortunately, these problems are easily addressed via dynamic memory allocation. **Dynamic memory allocation** is a way for running programs to request memory from the operating system when needed. This memory does not come from the program's limited stack memory -- instead, it is allocated from a much larger pool of memory managed by the operating system called the **heap**. On modern machines, the heap can be gigabytes in size.

Dynamically allocating single variables

To allocate a *single* variable dynamically, we use the scalar (non-array) form of the **new** operator:

```
new int; // dynamically allocate an integer (and discard the result)
```

In the above case, we're requesting an integer's worth of memory from the operating system. The new operator creates the object using that memory, and then returns a pointer containing the *address* of the memory that has been allocated.

Most often, we'll assign the return value to our own pointer variable so we can access the allocated memory later.

```
int* ptr{ new int }; // dynamically allocate an integer and assign the address to ptr
so we can access it later
```

We can then dereference the pointer to access the memory:

```
*ptr = 7; // assign value of 7 to allocated memory
```

If it wasn't before, it should now be clear at least one case in which pointers are useful. Without a pointer to hold the address of the memory that was just allocated, we'd have no way to access the memory that was just allocated for us!

Note that accessing heap-allocated objects is generally slower than accessing stack-allocated objects. Because the compiler knows the address of stack-allocated objects, it can go directly to that address to get a value. Heap allocated objects are typically accessed via pointer. This requires two steps: one to get the address of the object (from the pointer), and another to get the value.

How does dynamic memory allocation work?

Your computer has memory (probably lots of it) that is available for applications to use. When you run an application, your operating system loads the application into some of that memory. This memory used by your application is divided into different areas, each of which serves a different purpose. One area contains your code. Another area is used for normal operations (keeping track of which functions were called, creating and destroying global and local variables, etc…). We'll talk more about those later. However, much of the memory available just sits there, waiting to be handed out to programs that request it.

When you dynamically allocate memory, you're asking the operating system to reserve some of that memory for your program's use. If it can fulfill this request, it will return the address of that memory to your application. From that point forward, your application can use this memory as it wishes. When your application is done with the memory, it can return the memory back to the operating system to be given to another program.

Unlike static or automatic memory, the program itself is responsible for requesting and disposing of dynamically allocated memory.

Key insight

The allocation and deallocation for stack objects is done automatically. There is no need for us to deal with memory addresses -- the code the compiler writes can do this for us.

The allocation and deallocation for heap objects is not done automatically. We need to be involved. That means we need some unambiguous way to refer to a specific heap allocated object, so that we can request its destruction when we're ready. And the way we reference such objects is via memory address.

When we use operator new, it returns a pointer containing the memory address of the newly allocated object. We generally want to store that in a pointer so we can use that address later to access the object (and eventually, request its destruction).

Initializing a dynamically allocated variable

When you dynamically allocate a variable, you can also initialize it via direct initialization or uniform initialization:

```
int* ptr1{ new int (5) }; // use direct initialization
int* ptr2{ new int { 6 } }; // use uniform initialization
```

Deleting a single variable

When we are done with a dynamically allocated variable, we need to explicitly tell C++ to free the memory for reuse. For single variables, this is done via the scalar (non-array) form of the **delete** operator:

```
// assume ptr has previously been allocated with operator new
delete ptr; // return the memory pointed to by ptr to the operating system
ptr = nullptr; // set ptr to be a null pointer
```

What does it mean to delete memory?

The delete operator does not *actually* delete anything. It simply returns the memory being pointed to back to the operating system. The operating system is then free to reassign that memory to another application (or to this application again later).

Although it looks like we're deleting a *variable*, this is not the case! The pointer variable still has the same scope as before, and can be assigned a new value just like any other variable.

Note that deleting a pointer that is not pointing to dynamically allocated memory may cause bad things to happen.

Dangling pointers

C++ does not make any guarantees about what will happen to the contents of deallocated memory, or to the value of the pointer being deleted. In most cases, the memory returned to the operating system will contain the same values it had before it was returned, and the pointer will be left pointing to the now deallocated memory.

A pointer that is pointing to deallocated memory is called a **dangling pointer**. Dereferencing or deleting a dangling pointer will lead to undefined behavior. Consider the following program:

```
#include <iostream>

int main()
{
    int* ptr{ new int }; // dynamically allocate an integer
    *ptr = 7; // put a value in that memory location

    delete ptr; // return the memory to the operating system.  ptr is now a dangling
pointer.

    std::cout << *ptr; // Dereferencing a dangling pointer will cause undefined
behavior
    delete ptr; // trying to deallocate the memory again will also lead to undefined
behavior.

    return 0;
}
```

In the above program, the value of 7 that was previously assigned to the allocated memory will probably still be there, but it’s possible that the value at that memory address could have changed. It’s also possible the memory could be allocated to another application (or for the operating system’s own usage), and trying to access that memory will cause the operating system to shut the program down.

Deallocating memory may create multiple dangling pointers. Consider the following example:

```
#include <iostream>

int main()
{
    int* ptr{ new int{} }; // dynamically allocate an integer
    int* otherPtr{ ptr }; // otherPtr is now pointed at that same memory location

    delete ptr; // return the memory to the operating system.  ptr and otherPtr are
now dangling pointers.
    ptr = nullptr; // ptr is now a nullptr

    // however, otherPtr is still a dangling pointer!

    return 0;
}
```

There are a few best practices that can help here.

First, try to avoid having multiple pointers point at the same piece of dynamic memory. If this is not possible, be clear about which pointer “owns” the memory (and is responsible for deleting it) and which pointers are just accessing it.

Second, when you delete a pointer, if that pointer is not going out of scope immediately afterward, set the pointer to nullptr. We’ll talk more about null pointers, and why they are useful in a bit.

Best practice

Set deleted pointers to nullptr unless they are going out of scope immediately afterward.

Operator new can fail

When requesting memory from the operating system, in rare circumstances, the operating system may not have any memory to grant the request with.

By default, if new fails, a *bad_alloc* exception is thrown. If this exception isn’t properly handled (and it won’t be, since we haven’t covered exceptions or exception handling yet), the program will simply terminate (crash) with an unhandled exception error.

In many cases, having new throw an exception (or having your program crash) is undesirable, so there’s an alternate form of new that can be used instead to tell new to return a null pointer if memory can’t be allocated. This is done by adding the constant std::nothrow between the new keyword and the allocation type:

```
int* value { new (std::nothrow) int }; // value will be set to a null pointer if the
integer allocation fails
```

In the above example, if new fails to allocate memory, it will return a null pointer instead of the address of the allocated memory.

Note that if you then attempt to dereference this pointer, undefined behavior will result (most likely, your program will crash). Consequently, the best practice is to check all memory requests to ensure they actually succeeded before using the allocated memory.

```
int* value { new (std::nothrow) int{} }; // ask for an integer's worth of memory
if (!value) // handle case where new returned null
{
    // Do error handling here
    std::cerr << "Could not allocate memory\n";
}
```

Because asking new for memory only fails rarely (and almost never in a dev environment), it’s common to forget to do this check!

Null pointers and dynamic memory allocation

Null pointers (pointers set to nullptr) are particularly useful when dealing with dynamic memory allocation. In the context of dynamic memory allocation, a null pointer basically says “no memory has been allocated to this pointer”. This allows us to do things like conditionally allocate memory:

```
// If ptr isn't already allocated, allocate it
if (!ptr)
    ptr = new int;
```

Deleting a null pointer has no effect. Thus, there is no need for the following:

```
if (ptr) // if ptr is not a null pointer
    delete ptr; // delete it
// otherwise do nothing
```

Instead, you can just write:

```
delete ptr;
```

If ptr is non-null, the dynamically allocated variable will be deleted. If it is null, nothing will happen.

Best practice

Deleting a null pointer is okay, and does nothing. There is no need to conditionalize your delete statements.

Memory leaks

Dynamically allocated memory stays allocated until it is explicitly deallocated or until the program ends (and the operating system cleans it up, assuming your operating system does that). However, the pointers used to hold dynamically allocated memory addresses follow the normal scoping rules for local variables. This mismatch can create interesting problems.

Consider the following function:

```
void doSomething()
{
    int* ptr{ new int{} };
}
```

This function allocates an integer dynamically, but never frees it using delete. Because pointers variables are just normal variables, when the function ends, ptr will go out of scope. And because ptr is the only variable holding the address of the dynamically allocated integer, when ptr is destroyed there are no more references to the dynamically allocated memory. This means the program has now “lost” the address of the dynamically allocated memory. As a result, this dynamically allocated integer can not be deleted.

This is called a **memory leak**. Memory leaks happen when your program loses the address of some bit of dynamically allocated memory before giving it back to the operating system. When this happens, your program can't delete the dynamically allocated memory, because it no longer knows where it is. The operating system also can't use this memory, because that memory is considered to be still in use by your program.

Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well. Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash. Only after your program terminates is the operating system able to clean up and "reclaim" all leaked memory.

Although memory leaks can result from a pointer going out of scope, there are other ways that memory leaks can result. For example, a memory leak can occur if a pointer holding the address of the dynamically allocated memory is assigned another value:

```
int value = 5;
int* ptr{ new int{} }; // allocate memory
ptr = &value; // old address lost, memory leak results
```

This can be fixed by deleting the pointer before reassigning it:

```
int value{ 5 };
int* ptr{ new int{} }; // allocate memory
delete ptr; // return memory back to operating system
ptr = &value; // reassign pointer to address of value
```

Relatedly, it is also possible to get a memory leak via double-allocation:

```
int* ptr{ new int{} };
ptr = new int{}; // old address lost, memory leak results
```

The address returned from the second allocation overwrites the address of the first allocation. Consequently, the first allocation becomes a memory leak!

Similarly, this can be avoided by ensuring you delete the pointer before reassigning.

Conclusion

Operators new and delete allow us to dynamically allocate single variables for our programs.

Dynamically allocated memory has dynamic duration and will stay allocated until you deallocate it or the program terminates.

Be careful not to perform dereference a dangling or null pointers.

In the next lesson, we'll take a look at using new and delete to allocate and delete arrays.