C++ Syntax Notes: Uninitialized Pointers, Segfaults, and Undefined Behavior

This reading discusses some details of pointers and safe memory access. We hope to help you avoid common crashes and bugs in your code.

You should refer back to the lecture on heap memory, which gave an example of an uninitialized variable producing an unpredictable result. Let's go over a few definitions and concepts first.

Segmentation fault (Segfault)

Various kinds of programming bugs related to pointers and memory can cause your program to crash. On Linux, if you dereference an address that you shouldn't, this is often called "segmentation fault," or "segfault." For example, if you dereference a pointer that is set to nullptr, it will almost certainly cause a segfault and crash immediately. This code will segfault:

```
// This code compiles successfully, runs, and CRASHES with a s
int* n = nullptr;
std::cout << *n << std::endl;
```

On other operating systems, the error message may say something different. *Some* C++ compilers may respond to this in other unpredictable ways, but this is one type of error that has a fairly reliable symptom. It is actually an example of what we talk about in the next paragraph.

Undefined behavior

Sometimes it's possible to write a program that compiles, but that is not really a safe or valid program. There are some improper ways to write C++ that are not strictly forbidden by the C++ standard, but the C++ standard doesn't define how compilers are supposed to handle those situations, so your compiler may generate a program that works, or not! This is called "**undefined behavior**." Many beginning programmers make the mistake of thinking that just because their program compiles and runs for them on their system, that it must be valid and safe code. **This isn't true.** "It works for me" isn't an excuse, so be sure to proofread your code, use safe practices, and avoid relying on undefined behavior.

Many times, undefined behavior is caused by the careless use of uninitialized variables. Let's talk about initialization.

Initialization

Initialization is specifying a value for a variable from the moment it is created. Remember that if you don't initialize a pointer, you really should *not* try to dereference it. If you dereference an uninitialized pointer, even just to read from it and not to write to it, you may cause your program to crash, or something else unexpected might happen later. Here are some examples.

Examples with stack memory

This pointer "x" is uninitialized. It contains a seemingly random memory address. (However, this is also **not** a good way to get a source of randomness, even for those situations where we want random numbers!) Dereferencing this pointer would cause undefined (unpredictable) behavior:

```
// Dangerous; this can lead to careless mistakes and crashes!
int* x;
```

This pointer is explicitly initialized to nullptr. Dereferencing this pointer would cause the program to crash, immediately and predictably. (On Linux, this is often called a "segmentation fault," or "segfault.") As we'll discuss below, it's a good practice to set a pointer to nullptr if you aren't setting it to any other value immediately.

```
// Explicitly initializing a pointer to nullptr
int* y = nullptr;
```

You can also initialize a value with the () syntax following the variable name. If the type is a class, the parameters will be given to the class type's corresponding constructor. For built-in types such as int, which are not class types, there are no constructors; however, we can still specify an initialization value this way. Sometimes you'll see initialization done with the {} syntax instead of (). This is a new feature since C++11. It can be a good way to make it clear that you are performing an initialization, not some kind of function call. Later in this course sequence, you'll see some other good ways to use the {} initialization syntax.

```
// Other ways to initialize
int* y2(nullptr);
int* y3{nullptr};
```

Plain built-in types, such as int, that are not initialized will have unknown values. However, if you have a class type, its default constructor will be used to initialize the object. Here, int h has an unknown value, but supposing we have some well-defined class type Box, Box b will be given reasonable default values. (It's the responsibility of the Box class type to ensure that.)

```
// h is uninitialized!
int h;
// b will be default initialized
Box b;
```

Here we create integer "i" on the stack, safely initialized with value 7, and then we create a pointer "z" on the stack, initialized to the address of i. This way, z points to i. It's safe to dereference z.

```
int i = 7;
int* z = &i;
```

Examples with heap memory

When you want to use heap memory, you'll use the "new" operator. As with the stack memory case, you should be wary when you use "new" for a built-in type like int, since these types may not have default initialization. Therefore, you shouldn't assume they'll have an expected default value (such as 0). For those cases, be sure to initialize the value manually. Here are some examples.

The pointer "q" is created and initialized with the address for a newly-allocated integer on the heap. However, the integer that q points to does not have a predictable value. It depends on the compiler. We shouldn't rely on this integer to have any particular value at the beginning.

```
int* q = new int;
```

You can specify initialization parameters at the end of the "new" expression. This will create pointer "r" initialized with newly-allocated memory for an integer with the value "0" explicitly.

```
int* r = new int(0);
```

There are a lot of other special situations in C++ where different factors may slightly change how an object is initialized. You don't need to get into all those details. If you're unsure, the easiest thing to do is make sure that you explicitly initialize your variables. For a more exhaustive reference on initialization, you can refer to this page: https://en.cppreference.com/w/cpp/language/initialization

Resetting deleted pointers to nullptr

Now that we've reviewed initialization, especially for pointers, let's talk about why you should manually reset pointers to nullptr when you're done with them.

Note that using "delete" on a pointer frees the heap memory allocated at that address. However, deleting the pointer does not change the pointer value itself to "nullptr" automatically; you should do that yourself after using "delete", as a safety precaution. For example:

```
// Allocate an integer on the heap:
int* x = new int;
// Now x holds some memory address to a valid integer.
// Do some kind of work with the integer.
// We'll just set that integer to 7:
*x = 7;
// Now delete the pointer to deallocate the heap memory:
delete x;
// This destroys the integer on the heap and frees the memory.
// But now x still holds the memory address!
// Set x to nullptr for safety:
x = nullptr;
```

The idea here is that by manually setting x to nullptr after "delete x", we help prevent two kinds of problems:

1. We don't want to delete the same allocated address more than once by mistake, which could cause errors. Using "delete" on nullptr does nothing, so this way, if we accidentally try to delete the same address twice, nothing further happens.

2. We must never dereference a pointer to deallocated memory. This could cause the program to crash with a segfault, exhibit strange behavior, or cause a security vulnerability. However, attempting to dereference a nullptr will almost always cause a segfault and terminate the program immediately. This is actually better than causing a silent security vulnerabilty or another problem that is harder to detect! Therefore, it makes sense to set the deleted pointer to nullptr, thus ensuring that if we dereference it carelessly, then it will cause a very obvious runtime error that we can fix.

You should also note that we only need to use delete and nullptr with pointers, of course. Variables in stack memory don't need to be manually deleted. Those are automatically deallocated when they go out of scope. (Please refer to the other reading lesson on scope.)

In summary, remember that if you use "new," you will also need to "delete," and after you delete, you should set to nullptr.

Growing beyond pointers

As you go further in your lessons on C++, at first you may be frustrated that the use pointers and raw memory is very tedious. However, class types can be designed handle all the new and delete operations for you, invisibly. As you create your own robust data structures, and as you use libraries such as the C++ Standard Template Library, you will find that you very rarely have to use "new" and "delete" anymore.