6.9 — Dynamic memory allocation with new and delete

BY ALEX ON JULY 13TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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? That necessitates having some way to tell active from inactive items, 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
```

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!

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.

Initializing a dynamically allocated variable

When you dynamically allocate a variable, you can also initialize it via direct initialization or uniform initialization (in C++11):

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

Deleting single variables

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 = 0; // set ptr to be a null pointer (use nullptr instead of 0 in C++11)
```

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 dangli
    ptr = 0; // 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 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 0 (or nullptr in C++11). We'll talk more about null pointers, and why they are useful in a bit.

Rule: Set deleted pointers to 0 (or nullptr in C++11) 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 allo
```

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 memory, 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::cout << "Could not allocate memory";
}
```

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 address 0 or 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:

```cpp
if (ptr)
    delete ptr;
```

Instead, you can just write:

```cpp
delete ptr;
```

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

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:

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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 no scope and will stay allocated until you deallocate it or the program terminates.

Be careful not to dereference dangling or null pointers.

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

6.9a -- Dynamically allocating arrays

Index

6.8b -- C-style string symbolic constants

C++ TUTORIAL | PRINT THIS POST

305 comments to 6.9 — Dynamic memory allocation with new and delete

« Older Comments 1 2 3 4

davidbear
January 2, 2020 at 3:22 pm · Reply

Hello Alex,

Pointers, dangling pointers and delete are inherently confusing to new programmers. Since you are introducing other C++11 and C++14 concepts, please consider introducing std::unique_ptr (and maybe std::make_unique) as a magical black-box for removing the confusion. You no longer need to use delete, there is no risk of dangling or memory leaks. You can say that one "problem" is that it cannot be copied, but the solution is found in chapter 15.

nascardriver
January 3, 2020 at 6:11 am · Reply

I agree. The lessons in question are due for a reordering and overhaul to focus more on the safer containers and wrappers.

Alex
January 6, 2020 at 3:54 pm · Reply

Thanks for the feedback! I partially agree. Dynamic allocation and C-style arrays can (and will be) moved later in the lesson ordering as part of the rewrite.

Pointers are still useful even outside of dynamic allocation (e.g. function pointers, or function parameters where NULL is a valid option, and let's not forget *this) and dangling pointers (or references) can still result from misuses (e.g. see the new lesson on lambda captures) even without dynamic allocation.

Anil
December 25, 2019 at 10:27 pm · Reply

Can we say, A Null pointer is a special type of Dangling Pointer?
Dangling Pointer: A pointer which is not pointing to a valid memory address. i.e it is not pointing any valid value.
Null Pointer: A null value is a special value that means the pointer is not pointing at anything. A pointer holding a null value is called a null pointer.

If we see, Null Value means, it is not pointing to anything i.e a dangling Pointer.

A little bit of confusion here. Please correct me.

nascardriver
December 28, 2019 at 6:59 am · Reply

A dangling pointer is pointer to something, but that something is no longer valid. A null pointer is pointing to nothing.

Charan
December 25, 2019 at 7:26 am · Reply

```
#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 dan
    ptr = 0; // ptr is now a nullptr

    // however, otherPtr is still a dangling pointer!

    return 0;
}
```

Hey,is otherPtr acting as a reference for ptr here;

nascardriver
December 27, 2019 at 7:56 am · Reply

No, it's a copy of `ptr` that points to the same location. If it was a reference, line 9 would change the value of `otherPtr` too, but it doesn't.

Mhz93
October 12, 2019 at 11:24 am · Reply

Hi Alex & nascardriver
I read this lesson, according to your sentence " (and the operating system cleans it up, assuming your operating system does that)" I like to know what will happen if my OS does NOT clean up the leaked memory?

nascardriver
October 12, 2019 at 11:31 pm · Reply

Most operating systems free the memory when the process dies. I think I said Debian doesn't, that's wrong (Unless it's modified not to free memory).
There should be no difference between running out of memory because of an uncleaned leak and running too many processes or not having enough ram.
If your system has swap memory, the swap memory will be used (Drive will be utilized as ram) and you might notice that your system slows down (Because your disk is way slower than ram). When you're out of swap memory, your os will probably prevent you from doing anything that takes up more ram (eg. you can't start applications, `new` will fail). If I recall correctly, Windows shows a message box telling you that you don't have enough memory.

alfonso
October 3, 2019 at 7:38 am · Reply

```
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"

How about making the pointer constant ... somehow?

nascardriver
October 3, 2019 at 7:42 am · Reply

You could mark the pointer const by adding `const` after the asterisk.

```
int* const ptr{ new int{} };
```

You'd still have to delete `ptr`, but it prevents reassignment. This may or may not be desirable.

Samira Ferdi
September 5, 2019 at 4:52 pm · Reply

Hi, Alex and Nascardriver!

what happen if I allocate memory larger than my variable's size?

```
int *newMemory{ new int[10] };
```

nascardriver
September 6, 2019 at 12:34 am · Reply

`newMemory` is an array now. Next lesson.

Justin
July 31, 2019 at 12:30 pm · Reply

I used the idea of dangling pointers to try to do something interesting. Look at the code.

```
#include <iostream>  //For input and output

int main() {
  int *ptr;
  {
    int x{0};
    ptr = &x;
  }
  std::cout << "Waiting for a program to use the memory spot " << ptr << '\n';
  while (*ptr == 0)
    ;  //Wait for a program to change *ptr so we can see what it is
  std::cout << "The value in that spot has been changed to " << *ptr << '\n';
  return 0;
}
```

I open the program, and it just keep going through the loop. I have two questions. Could a program change the location of *ptr? If it could, could I read *ptr?

nascardriver
August 1, 2019 at 1:31 am · Reply

> Could a program change the location of *ptr?
On every major OS, not accidentally. It would need to purposefully access your program's memory at that location by os-specific means. Your program's memory is controlled by your program alone. You might be able to read data from an uninitialized location that was left in memory by another program when your program first starts. Once you override the memory, the old data is gone.

Justin
August 1, 2019 at 11:48 am · Reply

I was thinking that way, but variable x is inside of a scope. Shouldn't it get destroyed when it gets to line 8? If it gets destroyed, can't another program use it when an int variable is assigned?

nascardriver
August 1, 2019 at 11:52 pm · Reply

This is implementation and OS-specific. I'll just pick 1 example to show what could happen.

1 `main` gets called.
2 `main` reserves memory on the stack to store local variables, including `x`.
3 You're storing the address of `x`. That's an address on the stack, the memory is reserved for `main`.
4 `x` goes out of scope. It's memory location can be re-used by `main` alone, because `main` still owns that memory. It could now be overridden by line 9 if line 9 needs to (It doesn't).
5 You're waiting for something else to appear at `x`'s location, but your code doesn't do anything that tries to use memory. I don't think this location will be re-used by `main` at all.
(6) `main` is done and the reserved memory gets freed.

What might be more interesting is accessing random memory on the heap.

```
int *ptr{ new int{} };
delete ptr;

std::cout << "Waiting...
```

You'll have to read up on how your OS handles the heap, virtual memory, and access to invalid memory. Certainly the heap isn't shared between programs, but you might be able to allocate memory at a physical address that previously stored some other program's data.

This is all up to your OS, cpp is unrelated to the memory model.

Justin
August 2, 2019 at 9:11 am · Reply

I didn't know 2. Thanks for answering my questions.

David Cane
July 31, 2019 at 3:18 am · Reply

I have a question which kept me puzzled for quite a while.
Is there any reason that I should use new instead of malloc.
Hope I am not being very offensive.

nascardriver
July 31, 2019 at 4:18 am · Reply

`malloc` allocates memory but doesn't initialize anything and doesn't call constructors. `new` does.

David Cane
August 5, 2019 at 11:18 pm · Reply

thanks

DecSco

July 25, 2019 at 7:14 am · Reply

Uniform initialisation may apply to both the pointer and the variable it points to:

```
#include <iostream>

int main()
{
    // uniform initialisation for the integer, copy initialisation for the pointer
    int* i = new int { 1 };
    // uniform initialisation for the pointer, copy assignment for the integer
    int* j { new int };
    *j = 2;
    // uniform initialisation for both the integer and the pointer
    int* k { new int {3} };
    // works for arrays as well
    int* array { new int[*k] { 1, 2, 3 } };

    std::cout << *i << "\n";
    std::cout << *j << "\n";
    std::cout << *k << "\n";
    for ( int index {0}; index < *k; ++index )
        std::cout << array[index] << " ";
    std::cout << "\n";

    delete i;
    delete j;
    delete k;
    delete[] array;

    // they need not be set to nullptr, as they go out of scope immediately
    return 0;
}
```

VikFreeze
May 10, 2019 at 10:45 am · Reply

This site is a great resource!
Im making a DLL extension for NPS service and im stuck on how to ensure there are no memory leaks as there are scenarios were i dont understand if im suppose to deallocate or not and im hoping someone can help me understand how to deal with these scenarios, perhaps they can also be incorporated in the tutorial as the examples covered above are very basic.

Context: NPS Service starts -> Instance of dllhost.exe is invoked -> my dll gets loaded into the dllhost process
NPS then calls the RadiusExtensionProcess2 function and passes it a pointer to a Extension Control Block

```
DWORD RadiusExtensionProcess2(IN OUT PRADIUS_EXTENSION_CONTROL_BLOCK pECB)
{
        CONST RADIUS_ATTRIBUTE * Attribute;
    PRADIUS_ATTRIBUTE NewAttribute;
    CONST CHAR * ratPolicyNameString = "WLAN Manager - MAC Auth Extension";

    Attribute = RadiusFindFirstAttribute(pECB, ratNASPortType);

    if (Attribute != NULL && Attribute->dwValue == 19)
        {
           pECB->SetResponseType(pECB, rcAccessAccept);

           NewAttribute = new RADIUS_ATTRIBUTE;
           NewAttribute->dwAttrType = ratPolicyName;
```

```
            NewAttribute->fDataType = rdtString;
            NewAttribute->cbDataLength = 34;
            NewAttribute->lpValue = (BYTE *)ratPolicyNameString;

            RadiusReplaceFirstAttribute(pECB, NewAttribute);
        }
        else
        {
            pECB->SetResponseType(pECB, rcAccessReject);
        }

        return NO_ERROR;
}

CONST RADIUS_ATTRIBUTE * WINAPI RadiusFindFirstAttribute(PRADIUS_EXTENSION_CONTROL_BLOCK pE
{
    DWORD i;
    PRADIUS_ATTRIBUTE_ARRAY AttributesArray;

    i = RadiusFindFirstIndex(pECB, dwAttrType);

    if (i != RADIUS_ATTR_NOT_FOUND)
    {
       AttributesArray = pECB->GetRequest(pECB);
       return AttributesArray->AttributeAt(AttributesArray, i);
    }
    else
    {
       return NULL;
    }
}

DWORD WINAPI RadiusFindFirstIndex(PRADIUS_EXTENSION_CONTROL_BLOCK pECB, DWORD dwAttrType)
{
    DWORD i, AttributesArraySize;
    CONST RADIUS_ATTRIBUTE * Attribute;
    PRADIUS_ATTRIBUTE_ARRAY AttributesArray;

    AttributesArray = pECB->GetRequest(pECB);
    if (AttributesArray == NULL) { return NULL; }

    /* Get the number of attributes in the array */
    AttributesArraySize = AttributesArray->GetSize(AttributesArray);

    /* Iterate through the array ... */
    for (i = 0; i < AttributesArraySize; ++i)
    {
        /* ... looking for the first attribute that matches the type. */
        Attribute = AttributesArray->AttributeAt(AttributesArray, i);
        if (Attribute->dwAttrType == dwAttrType) { return i; }
    }

    return RADIUS_ATTR_NOT_FOUND;
}

DWORD WINAPI RadiusReplaceFirstAttribute(PRADIUS_EXTENSION_CONTROL_BLOCK pECB, CONST RADIUS
{
    DWORD i;
    PRADIUS_ATTRIBUTE_ARRAY AttributesArray;

    AttributesArray = pECB->GetRequest(pECB);

    if ((AttributesArray == NULL) || (pSrc == NULL))
    {
        return ERROR_INVALID_PARAMETER;
    }

    i = RadiusFindFirstIndex(pECB, pSrc->dwAttrType);

    if (i != RADIUS_ATTR_NOT_FOUND)
    {
        /* It already exists, so overwrite the existing attribute. */
        return AttributesArray->SetAt(AttributesArray, i, pSrc);
    }
    else
    {
        /* It doesn't exist, so add it to the end of the array. */
        return AttributesArray->Add(AttributesArray, pSrc);
    }
}
```

From the tutorials, pECB and i are created and destroyed along with the stack frame so i dont have to deallocate them but Attribute, AttributesArray and NewAttribute arent as clear...
Attribute and AttributesArray are returned via calls to functions in the control block, but i dont know if i should deallocate them.
NewAttribute is a new attribute i create and add to the attribute array but should i deallocate it? The array lives on in the NPS so i dont think i should release the memory.

nascardriver
May 11, 2019 at 3:48 am · Reply

> Attribute
Don't delete, it already exists before the call to @AttributeAt.

> AttributesArray
Don't delete, it already exists before the call to @GetRequest.

> NewAttribute
I couldn't find a documentation of @RadiusReplaceFirstAttribute. I don't see why @RadiusReplaceFirstAttribute would want a pointer when it creates a copy of the object, so don't delete it right away.

If you're working with someone else's code, you need to read their documentations. If there is no documentation, you need to read their code. If their code isn't public, you can try to see if someone else figured it out already, or figure it out yourself (by trying or reverse engineering).

VikFreeze
May 13, 2019 at 1:52 am · Reply

Oops, forgot to add @RadiusReplaceFirstAttribute

```
DWORD WINAPI RadiusReplaceFirstAttribute(PRADIUS_EXTENSION_CONTROL_BLOCK pECB, CO
{
    DWORD i;
    PRADIUS_ATTRIBUTE_ARRAY AttributesArray;

    AttributesArray = pECB->GetRequest(pECB);

    if ((AttributesArray == NULL) || (pSrc == NULL))
    {
        return ERROR_INVALID_PARAMETER;
    }

    i = RadiusFindFirstIndex(pECB, pSrc->dwAttrType);

    if (i != RADIUS_ATTR_NOT_FOUND)
    {
        /* It already exists, so overwrite the existing attribute. */
        return AttributesArray->SetAt(AttributesArray, i, pSrc);
    }
    else
    {
        /* It doesn't exist, so add it to the end of the array. */
        return AttributesArray->Add(AttributesArray, pSrc);
    }
}
```

So i should leave the objects from the control block to be manages by the NPS service.
Im guessing the same applies to the NewAttribute allocating as well since it will be part of the attribute array after my code drops out of scope.
Is want im doing with the ratPolicyNameString string sensible? Its a fixed string that i only need a reference to for the lifetime of the service so it should be statically allocated.

nascardriver
May 13, 2019 at 2:31 am · Reply

> Im guessing the same applies to the NewAttribute
I don't know. It depends on whether the API deletes the elements when it doesn't need them anymore (eg. when @SetAt is called), you'll have to search through the documentation.

> Is want im doing with the ratPolicyNameString string sensible?
Just from looking at your code, no. You're casting away the constness, which leads to undefined behavior. Then again, it's the Windows API, it's filled with bad decisions, so this might just be the way they indent it to be used. Check the documentation of @lpValue, if it's never modified, you can keep your code the way it is.

VikFreeze
May 13, 2019 at 6:29 am · Reply

Well the attribute is the policy name and i dont think its changed but i dont know how to verify that...

Aside from that, given the definition of Attribute below, how does one assign a string value to lpValue in a sane way?

```
typedef struct _RADIUS_ATTRIBUTE {
    DWORD dwAttrType;              /* Attribute type */
    RADIUS_DATA_TYPE fDataType;    /* RADIUS_DATA_TYPE of the value */
    DWORD cbDataLength;            /* Length of the value (in bytes) */
    union {
        DWORD dwValue;             /* For rdtAddress, rdtInteger, and rdtTim
        CONST BYTE* lpValue;       /* For rdtUnknown, rdtIpv6Address and rdt
    };
} RADIUS_ATTRIBUTE, *PRADIUS_ATTRIBUTE;
```

nascardriver

May 13, 2019 at 6:31 am · Reply

It's a CONST BYTE*, so your cast to BYTE* is wrong.

```
NewAttribute->lpValue = (CONST BYTE *)ratPolicyNameString;
//                       ^^^^^
```

Alireza
May 7, 2019 at 11:48 am · Reply

Hi there,
Sorry for my unrelated question.

I wrote this snippet of code and I don't know why a pointer can be added with 1 but its value can't be. If the pointer can, where does that pointer point to exactly ?

```
#include <iostream>

void does(int val, int *ptr, int &ref)
{
    val += 1;
    *ptr += 1;
    ref += 1;
    std::cout << "value_func: " << val << "\t\t" << "address: " << val << "\n";
    std::cout << "ptr_func: " << *ptr << "\t\t" << "address: " << ptr << "\n";
    std::cout << "ref_func: " << ref << "\t\t" << "address: " << &ref << "\n";
    val += 1;
    *ptr += 1;
//    ref += 1;
}

int main()
{
    int value{0};

    int *ptr_value { new (std::nothrow) int { value }};
    if(!ptr_value)  std::cout << "allocation error!\n";

    int &ref_value {value};

    std::cout << "value_before: " << value << "\t\t" << "address: " << value << "\n";
    std::cout << "ptr_before: " << *ptr_value << "\t\t" << "address: " << ptr_value << "\n"
    std::cout << "ref_before: " << ref_value << "\t\t" << "address: " << &ref_value << "\n"

    std::cout << "\n";

    does(value, ptr_value, ref_value);

    std::cout << "\n";

    std::cout << "value_after: " << value << "\t\t" << "address: " << value << "\n";
    std::cout << "ptr_after: " << *ptr_value << "\t\t" << "address: " << ptr_value << "\n";
    std::cout << "ref_after: " << ref_value << "\t\t" << "address: " << &ref_value << "\n";

    delete ptr_value;
    ptr_value = nullptr;


    return 0;
}
```

nascardriver
May 8, 2019 at 2:36 am · Reply

Hi Alireza!

This was covered in lesson 6.8a, give it another read.

Enso
April 24, 2019 at 5:17 am · Reply

I think the following statement is phrased sub-optimally:

„Dynamically allocated memory effectively has no scope.“

Doesn't „scope“ refer to a particular area of the code in which a variable (name) is visible? So instead you could write something like:

„Dynamically allocated memory has no predetermined duration“

Wouldn't that be more appropriate?

Alex
April 24, 2019 at 12:42 pm · Reply

Yes, I'd misused the term "scope" here. I've updated the lesson to amend this. Thanks for pointing out the error!

Dylan
April 17, 2019 at 12:07 pm · Reply

hey... I can't seem to wrap my head around why this code produces different results.

```
1   int main()
2   {
3
4       int *ptr = new int;
5
6       *ptr = 3;
7
8       std::cout << &ptr << '\n';
9       std::cout << ptr << '\n';
10
11      return 0;
12  }
```

nascardriver
April 18, 2019 at 1:19 am · Reply

As with every variable, you need space to store a pointer.
Line 8 prints the address of @ptr (Where @ptr is stored in memory).
Line 9 prints the address stored in @ptr (Where @ptr is pointing to).

Arie
March 20, 2019 at 9:00 am · Reply

I was wondering. Is there a use case for a freed pointer that doesn't point to null? Why don't the C++ comittee introduce a standard delete ptr to also set ptr = nullptr?

nascardriver
March 20, 2019 at 10:27 am · Reply

Hi Arie!

See Stroustrup's answer over here http://www.stroustrup.com/bs_faq2.html#delete-zero
TLDR:
@delete needs to be able to handle non-l-values.
Zeroing the pointer wouldn't help all too much, as there's more that could go wrong.

Arie
March 20, 2019 at 10:31 pm · Reply

Thank you.

It took me a while to understand the meaning of deleting an r-value, but I get it now.

Behrooz
February 23, 2019 at 10:37 am · Reply

Hi ! I've wrote a code like this :

```
#include "pch.h"
#include <iostream>

struct BILL
{
    int yearStart;
    int yearFinish;
    int monthStart;
    int monthFinish;
    int dayStart;
    int dayFinish;
    int days;
    int *daysArray = new int[days];
};

void getBillData(BILL billData);

int main()
{
    BILL billData;
    getBillData(billData);
}

// Getting BILL Data From User And Find Values Of Variables

void getBillData(BILL billData)
{
    int &yearStart = billData.yearStart;
    int &yearFinish = billData.yearFinish;
    int &monthStart = billData.monthStart;
    int &monthFinish = billData.monthFinish;
    int &dayStart = billData.dayStart;
    int &dayFinish = billData.dayFinish;

    while (true)
    {
        std::cout << " Enter The Start Date Of Your Bill (Year Month Day) : ";
        std::cin >> yearStart >> monthStart >> dayStart;
        if (std::cin.fail())
        {
            std::cout << " Sorry, Wrong Value!\n";
            std::cin.clear();
            std::cin.ignore(32767, '\n');
        }
        else
            if ((monthStart > 12) || (dayStart > 31) || (monthStart < 1) || (dayStart < 1))
            {
                std::cout << " Sorry, Wrong Value!\n";
                std::cin.clear();
                std::cin.ignore(32767, '\n');
            }
            else
                break;
    }

    while (true)
    {
        std::cout << " Enter The End Date Of Your Bill (Year Month Day) : ";
        std::cin >> yearFinish >> monthFinish >> dayFinish;
        if (std::cin.fail())
        {
            std::cout << " Sorry, Wrong Value!\n";
            std::cin.clear();
            std::cin.ignore(32767, '\n');
        }
        else
            if ((monthFinish > 12) || (dayFinish > 31) || (monthFinish < 1) || (dayFinish <
            {
                std::cout << " Sorry, Wrong Value!\n";
                std::cin.clear();
                std::cin.ignore(32767, '\n');
            }
            else
                break;
    }

    int &days = billData.days;

    days = (((yearFinish - yearStart) * 365) + (((monthFinish - monthStart) * 30) - 1) + (d

    for (int i = 0; i < days; ++i)
    {
        billData.daysArray[i] = 1;
    }
    std::cout << billData.daysArray << '\n';
    delete[] billData.daysArray;
}
```

I get runtime error for it.
Can you help me with this poblem ?
Thanks!

nascardriver
February 23, 2019 at 10:50 am · Reply

* Line 13, 81: Initialize your variables with brace initializers. You used copy initialization.
* Line 16: Initialize your variables with brace initializers. You used direct initialization.
* Line 6, 7, 8, 9, 10, 11, 12, 20: Initialize your variables with brace initializers.
* Line 46, 67, 79: Limit your lines to 80 characters in length for better readability on small displays.
* Line 43, 50, 64, 71: Don't pass 32767 to @std::cin.ignore. Pass @std::numeric_limits<std::streamsize>::max().
* Enable compiler warnings, read them, fix them.

@daysArray uses @days, which is uninitialized.

Jeroen P. Broks
February 18, 2019 at 7:58 am · Reply

Since I was a bit spoiled by languages such as Go and C# and many others having a built-in garbage collector which cleans everything up automatically when nothing points to it anymore, but when taking this into C++ (and I have a bit of the same issue in C), how can I prevent this in rather complex programs:

```
1     int *a = new int{5};
2     int *b;
3     b = a;
4     delete a;
5     a = nullptr;
6     std::cout << *b; // ack I forgot b is still pointing to that data, but that memory addres
```

Now this has been done deliberately, but in complex programs, I want to prevent memory leaks, or pointing to memory blocks already released. I guess it will also be one terrible experience to debug to find out where a delete should have been and where not. ;)

nascardriver
February 18, 2019 at 8:08 am · Reply

* Line 1: Initialize your variables with brace initializers. You used copy initialization.
* Line 2: Initialize your variables with brace initializers.
* Line 6: Limit your lines to 80 characters in length for better readability on small displays.

If you've got somewhat decent memory management and think about your code, this should never happen.
C++ has smart pointers (covered later), which take care of freeing the memory when it's no longer used.
You're rushing through these lessons, which is fine, since you already know other languages, but you're making the same mistakes over and over. Take a break, eat something, watch a video.

Alex
February 19, 2019 at 7:54 pm · Reply

std::shared_ptr is designed to help prevent this kind of thing. It's covered in chapter 15.

Thagrous
January 16, 2019 at 9:25 am · Reply

When declaring a ptr to be a nullptr is it better practice to do

```
int ptr* = new int;
delete ptr;
ptr = 0;
```

or

```
int ptr* = new int;
delete ptr;
ptr = nullptr;
```

many thanks

Piyush
January 16, 2019 at 10:53 pm · Reply

If your compiler is not C++ 11 compatible you can use the first case. If it is then you should prefer the second case. Its mentioned in this lesson take a look again.

nascardriver
January 17, 2019 at 7:40 am · Reply

You shouldn't allocate a new int at all.

```
int *ptr{ nullptr };
```

Chris
December 8, 2018 at 4:29 am · Reply

Does that mean if I create something via a function with new it is better to give it better via reference / pointer or just create it on the stack and make a copy of it on the return?

nascardriver
December 8, 2018 at 5:26 am · Reply

If you create something using @new inside a function, you have to either delete that object in the same function or return by pointer (and add a comment telling the caller that manual deletion is required).
Returning large objects by value should be avoided. It's better to let the caller create the object and pass a pointer/reference to the function. Usually this happens automatically (Return value optimization).

Denys
December 2, 2018 at 1:21 pm · Reply

Hello Alex,

I have got question regarding the statement above:

"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!"

From a designer of a programming language point of view I still do not get one thing: why should a user be bothered with pointer details?

Why not to provide, for example, the following syntax:
[code]
dyn int myDynamicVariable;
[\code]

If compared to automatic variables, in their nature they are also nothing else, but just memory location (but in stack, however), and yet a user does not need to explicitly ask a compiler, hey, give me please 4 bytes of memory for my integer and here is the pointer where I expect you to place the requested memory address (if you find enough).

Instead, a user simply defindes a variable and vua la, no memory address details (yet possible to find out using oeprator&).

Thank You in advance!

Kind Regards,
Denys

nascardriver
December 3, 2018 at 10:34 am · Reply

Because sometimes you need control over memory (eg. when accessing another process' memory). C++ isn't designed to be especially easy or intuitive. It's designed to let the programmer do whatever they please while still being a rather high-level language.

Yasser
November 16, 2018 at 4:35 am · Reply

```
 #include <iostream>
using namespace std;




int main ()
{


    int* pvalue = new int;
    *pvalue = 20;
    cout << endl;

    cout << pvalue << endl;
    cout << *pvalue << endl;


    delete pvalue;

    cout << endl;

    cout << pvalue << endl;
    cout << *pvalue << endl;



28
29      return 0;
30  }
```

Output:
0x7fb039c02830
20

0x7fb039c02830
20

Is this means the address and the value moved from heap to stack after deleting the pvalue variable?

Thank you

nascardriver
November 16, 2018 at 5:07 am · Reply

> the address and the value moved from heap to stack after deleting the pvalue variable?
C++ doesn't have guides for when the heap or stack is used. Use of head and stack is implementation defined. @delete doesn't move anything, it marks the memory that was referenced by the pointer as free. Whether or not that memory is zeroed out is implementation defined. The pointer itself remains untouched.

Line 23 will always print the same value as line 15.
The behavior of line 24 is undefined. It might crash, it might print seemingly random data, it might print old data.

Erdem Tuna
November 7, 2018 at 11:37 am · Reply

The explanations of the concept and the examples are very clear. Thanks for such a documentation.

Nguyen
September 25, 2018 at 7:45 pm · Reply

Hi nascardriver,

```
1  new int;
2  int *ptr = new int;
3  std::cout << new int << std::endl;
4  std::cout << ptr;
```

Output:

00994D58
00998C30

I really expected to see the same addresses but they are not. Can you tell me why?

nascardriver
September 26, 2018 at 12:43 am · Reply

@new allocates new memory every time you call it. It can't allocate memory in the same location again unless that memory was deleted.
Line 1 creates memory for an int, the address isn't stored, you're leaking memory.
Line 2 creates memory for an int, it can't do so at the same location as the previous call, because that

memory is in use. The address is stored in @ptr.
Line 3 creates memory for an int, again, it can't use any of the previous locations, because that memory is in use. The address is printed to @stdout, possibly leaking memory.

Nguyen
September 26, 2018 at 9:06 pm · Reply

Hi nascardriver,

I'd like to make sure I understand your explanation well in Line1, Line2, and Line3 as followings:

Line1: new (new operator) creates memory for an int, the address isn't stored, you are leaking memory.

Line2: new (new operator) creates memory for an int, new operator can't do so at the same location as the previous call, because that memory is in use; therefore, a different(new) address is stored in ptr.

Line3: new operator creates memory for an int, again, new operator can't use any of the previous locations, because that memory is in use; therefore, a different(new) address is printed to @stdout, possibly leaking memory.

Another question:

```
int *ptr2 = new int { 6 }; // use uniform initialization
```

Does it mean the address is initialized with the value of 6?

nascardriver
September 27, 2018 at 6:08 am · Reply

> I understand your explanation well in Line1, Line2, and Line3 as followings
Correct

> Does it mean the address is initialized with the value of 6?
No. You don't choose an address, your computer chooses the address. The 6 is the value the new memory is supposed to be initialized to. @ptr2 holds the address, let's say 0x123456, and at that address in memory the value 6 is stored.

Nguyen
September 24, 2018 at 8:33 pm · Reply

Hi,

Pointers are variables that hold a memory address.

```
int x;
int *ptr1 = &x;       // initialize ptr1 with address of variable x.

new int;              //dynamically allocate an integer.
                      //In other words, is it the address of the memory that has been alloca
int *ptr2 = new int;  //assign allocated the address to ptr2 so we can access it later.
```

This means both &x and new int are the memory addresses in general. Am I right? Please review my comments, let me know if my comments are right.

Could please tell me the difference between &x and new int?

Since both pointers can only hold the memory address, then why

```
1  int *ptr1 = &x(1)       // it is not legal.
2  int *ptr2 = new int(1)  // it is legal.
```

Thanks, have a great day

nascardriver
September 24, 2018 at 11:32 pm · Reply

Hi Nguyen!

> initialize ptr1 with address of variable x.
Yes, but you should use uniform initialization

> dynamically allocate an integer
Yes.

> is it the address of the memory that has been allocated to int?
"new int" allocates memory to hold an int (usually 4 bytes) and returns the address of that memory.
You're not storing the address anywhere, resulting in a memory leak, because now there's no way for you to delete the dynamically allocated memory.

> assign allocated the address to ptr2 so we can access it later.
You're allocating memory for an int and storing the address of that memory in @ptr2.

> This means both &x and new int are the memory addresses in general.
> Could please tell me the difference between &x and new int?
&x returns the address of @x. @x lives on the stack and will be removed from it once it goes out of scope, ie. it is a temporary variable.
"new int" allocates memory on the heap, that memory will stay there until you delete it. It won't be freed automatically.

> Since both pointers can only hold the memory address [...]
Line 1 assumes that @x is a function which you're calling with the argument 1, applying @operator& to the return value of @x. Since @x is a variable, this is invalid syntax and unrelated to pointers.
Line 2 allocates memory for an int, initializes that memory to 1 and returns the address, storing it in @ptr2.
If you have another example for what you meant by line 1, I'll try to answer it.

Nguyen
September 25, 2018 at 7:27 pm · Reply

Thank you for your time to answer my questions in detail.

Let's me put all the code together.

```
int x;
int *ptr1 = &x{123};      //this is not legal

new int;
int *ptr2 = new int{123};//this is legal
```

Conclusion:
Both &x and new int are the memory addresses.
 . &x is the memory address of x that lives on the stack.
 . new int is the memory address that has been allocated to int that we request from the operating system (heap).

Although both ptr1 and ptr2 hold the memory addresses, ptr1 can not be initialized but ptr2 can be initialized via direct initialization or uniform initialization.

Please let me know my conclusion is correct.
Thanks again.

nascardriver
September 26, 2018 at 12:14 am · Reply

> &x is the memory address of x that lives on the stack.
Unary @operator& returns the address of @x. @x lives on the stack. It's memory exists before the call.

> new int is the memory address that has been allocated to int that we request from the operating system (heap).
@new is an expression, calling @operator new, accepting a type and initializer. It allocates memory on the heap with the size of the given type and initializes it using the given initializer.

> ptr1 can not be initialized but ptr2 can be initialized
@ptr1 and @ptr2 don't care about the value they're pointing to, they only hold the address. You're trying to initialize the memory that they're pointing to, which already exists in case 1 so it can't be initialized. Case 2 creates new memory, which can be initialized.

{123} isn't applied to an address, it's used as a part of a @new expression.

If you want to assign a value to @x while applying @operator& you can use

```
int i{ 0 }; // @i's memory exists and is initialized to 0
int *p{ &(i = 9) }; // Assign 9 to @i. @operator= returns @i, we can apply @op
// Same as
// int i{ 0 };
// i = 9;
// int *p{ &i };
```

« Older Comments 1 2 3 4