Pointers and Arrays

Mark Slater

UNIVERSITY OF BIRMINGHAM

Raw Pointers

Introducing Raw Pointers

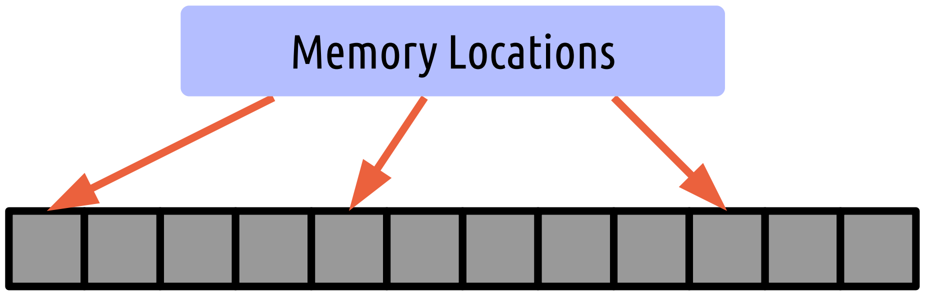
- Last week we covered the concept of Pointers and how they are used to enable dynamic allocation and for extending the lifetime of an object beyond the scope it was created in
- Before the Smart Pointers we saw last week were available, this functionality was only possible through 'raw' C-Style pointers
- These are very basic types that just hold the memory location of the object they point to and don't have any other functionality
- This means you have to be very careful about ensuring the pointer points at a valid object and deleting any unused objects when appropriate
- To create a new object and return a pointer to it, use the 'new' operator. When finished with this object, use the 'delete' operator to de-allocate the associated memory

nullptr

- As we will see, it's vitally important to initialise any pointers to null otherwise they could point anywhere
- Pre-C++11, the best way of doing this is to set it to 0 (or 'NULL' is sometimes defined) which would cause a crash if accessed and can be checked specifically do decide if it's valid
- In C++11 there is a keyword nullptr that designates a null pointer constant of type std::nullptr_t which is a lot more robust (and won't be converted to/from an integer on the fly) and less confusing
- This can be used for *any* pointer type and will resolve correctly

Pointers in Action (1)

```
#include <iostream>

int main()
{
    int *a{nullptr};
    int *b;    // VERY BAD!!

    a = new int {5};
    b = new int;

    *b = 7;

    int c {(*a) + (*b)};
    int *d{&c};
    int *e{a};

    delete a;

    // 0x7fff3f0fc6c4:  12
    std::cout << d << ":  " << *d
        << std::endl;

    return 0;
}
```

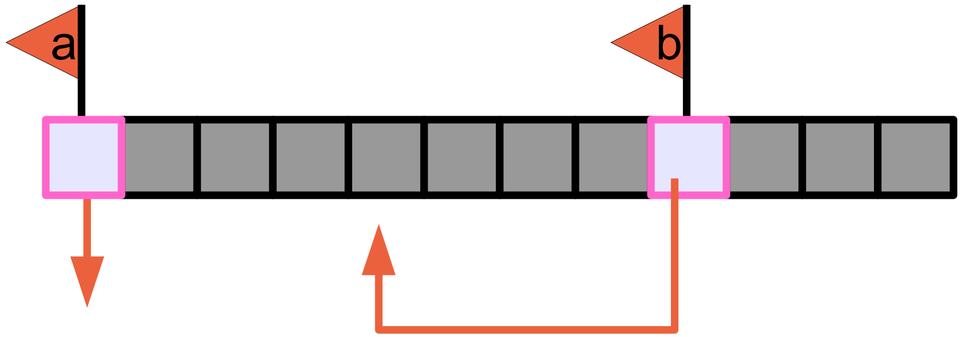


To demonstrate how pointers work, we will return to our basic memory picture shown in the first week

This code is just for teaching purposes – don't use it in practise!

Pointers in Action (2)

```cpp
#include <iostream>

int main()
{
    int *a{nullptr};
    int *b;    // VERY BAD!!

    a = new int {5};
    b = new int;

    *b = 7;

    int c {(*a) + (*b)};
    int *d{&c};
    int *e{a};

    delete a;

    // 0x7fff3f0fc6c4:  12
    std::cout << d << ":  " << *d
        << std::endl;

    return 0;
}
```

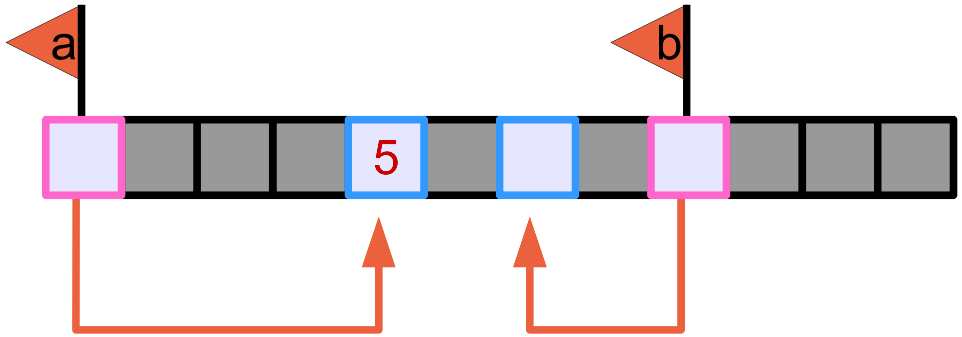


First, two pointers to ints (a and b) are declared

'a' is initialised with the nullptr but 'b' is not – if you accessed 'a' your program would crash which is actually better than if you accessed 'b' as this would be completely undefined!

Pointers in Action (3)

```
#include <iostream>

int main()
{
    int *a{nullptr};
    int *b;   // VERY BAD!!

    a = new int {5};
    b = new int;

    *b = 7;

    int c {(*a) + (*b)};
    int *d{&c};
    int *e{a};

    delete a;

    // 0x7fff3f0fc6c4:  12
    std::cout << d << ":  " << *d
        << std::endl;

    return 0;
}
```

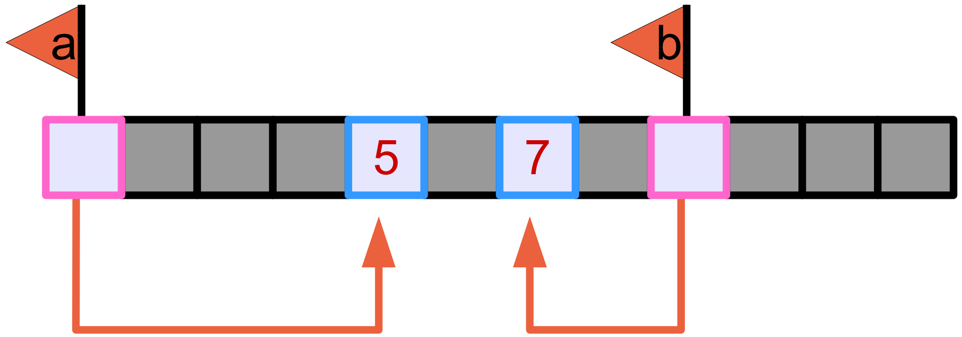


Next, *the memory* for two integers are allocated and the addresses of these allocations are assigned to the pointers using the 'new' operator

At this point, a and b now point to useful memory locations, but the value of *b has not been initialised (i.e. junk!)

Pointers in Action (4)

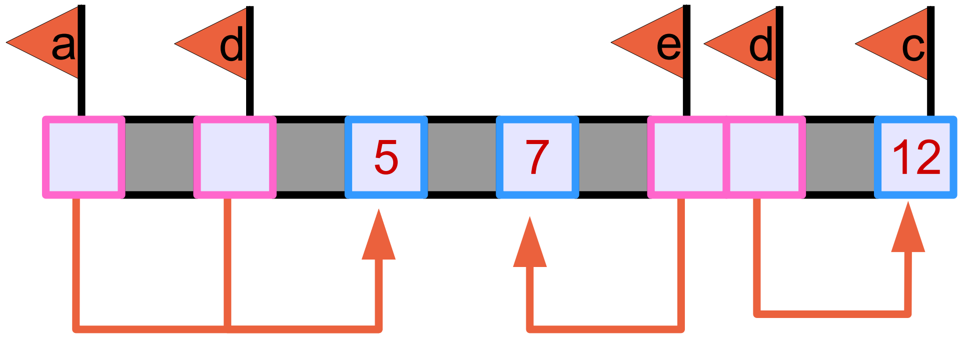
```
#include <iostream>

int main()
{
    int *a{nullptr};
    int *b;    // VERY BAD!!

    a = new int {5};
    b = new int;

    *b = 7;

    int c {(*a) + (*b)};
    int *d{&c};
    int *e{a};

    delete a;

    // 0x7fff3f0fc6c4:  12
    std::cout << d << ":  " << *d
        << std::endl;

    return 0;
}
```



To actually assign a value to the integer pointed at by b, we can *dereference* the pointer as we did with smart pointers and iterators

Pointers in Action (5)

```
#include <iostream>

int main()
{
    int *a{nullptr};
    int *b;    // VERY BAD!!

    a = new int {5};
    b = new int;

    *b = 7;

    int c {(*a) + (*b)};
    int *d{&c};
    int *e{a};

    delete a;

    // 0x7fff3f0fc6c4:  12
    std::cout << d << ":  " << *d
        << std::endl;

    return 0;
}
```



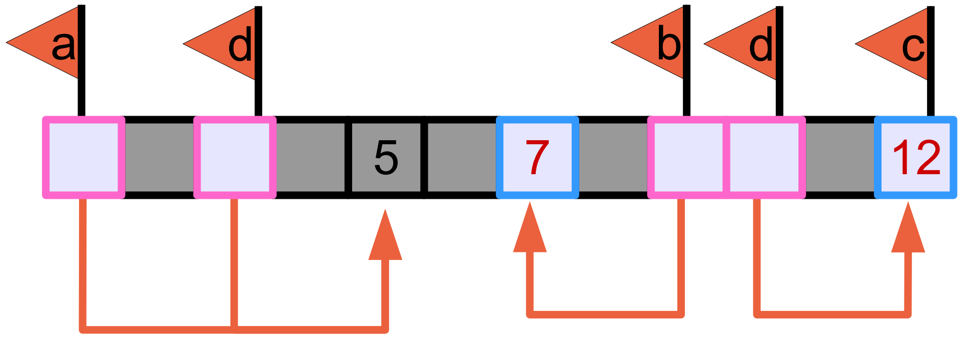
Next, we create a normal integer variable and assign the sum of the two integers pointed to by a and b

After this, we create another pointer (d) and assign it to the *address* of the variable c by pre-fixing with the *'address-of' operator*

We also create another pointer (e) and set it to the value of 'a'

Pointers in Action (6)

```
#include <iostream>

int main()
{
    int *a{nullptr};
    int *b;    // VERY BAD!!

    a = new int {5};
    b = new int;

    *b = 7;

    int c {(*a) + (*b)};
    int *d{&c};
    int *e{a};

    delete a;

    // 0x7fff3f0fc6c4:  12
    std::cout << d << ":  " << *d
        << std::endl;

    return 0;
}
```

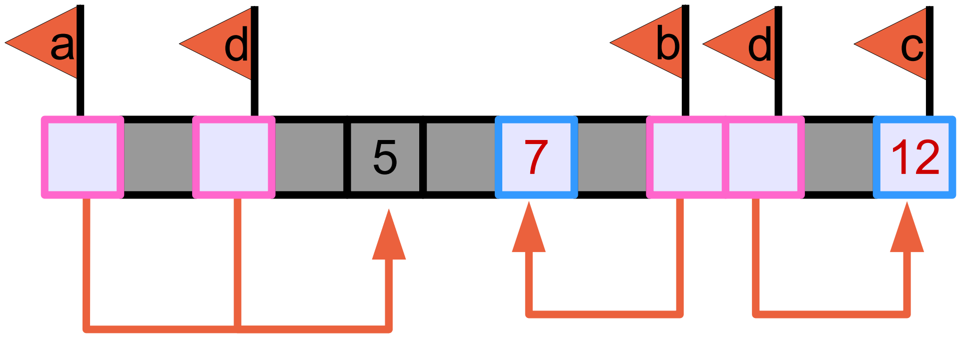


Just to show what happens, we now delete the memory allocated to the pointer 'a'

If we were to try to access either the 'a' or 'e' variables after this we would get unexpected results because the object they point to has now been deleted

Pointers in Action (7)

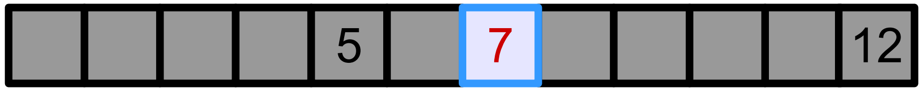
```
#include <iostream>

int main()
{
    int *a{nullptr};
    int *b;    // VERY BAD!!

    a = new int{5};
    b = new int;

    *b = 7;

    int c {(*a) + (*b)};
    int *d{&c};
    int *e{a};

    delete a;

    // 0x7fff3f0fc6c4:  12
    std::cout << d << ":  " << *d
        << std::endl;

    return 0;
}
```



To give you an idea of what the contents of these variables are, we now print out the pointer and the dereferenced pointer

Pointers in Action (8)

```
#include <iostream>

int main()
{
    int *a{nullptr};
    int *b;    // VERY BAD!!

    a = new int {5};
    b = new int;

    *b = 7;

    int c {(*a) + (*b)};
    int *d{&c};
    int *e{a};

    delete a;

    // 0x7fff3f0fc6c4:  12
    std::cout << d << ":  " << *d
        << std::endl;

    return 0;
}
```



Finally, on exit of this scope, we see that because we only deleted one of the integers created with 'new', we're left with memory allocated that is no longer referenced, *i.e. it does not go out of scope*

This is the other major problem with pointers: memory leaks!

Drawbacks of Using Raw Pointers

- There are several reasons why it's a bad idea to use raw pointers instead of smart pointers:
    - You have to remember to 'delete' anything you 'new'
    - There is a question over returned pointers as to who has ownership
    - It's much harder to ensure unique pointers
    - There is no protection for bad pointers
- In summary:

Only use Raw Pointers when you absolutely have to – Smart Pointers are better in almost every practical way!

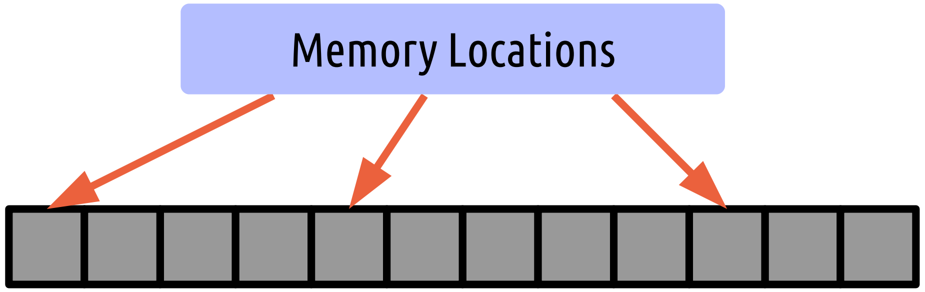
C-Style Arrays

Introducing C-Style Arrays

- We have currently only dealt with C++ containers that do all the memory (de)allocation for us
- Before these were available, there was only the 'C-Style' arrays to hold multiple objects. Due to their implementation, they are intimately linked to pointers
- Essentially, to create an array, you do exactly as was done for a single variable allocation but add the array size in square brackets afterwards. This produces a variable that represents the full memory allocated
- These have very similar drawbacks to raw pointers and also don't have any of the power of C++ containers, e.g. dynamic resizing, memory (de)allocation control, etc.

Arrays in Action (1)

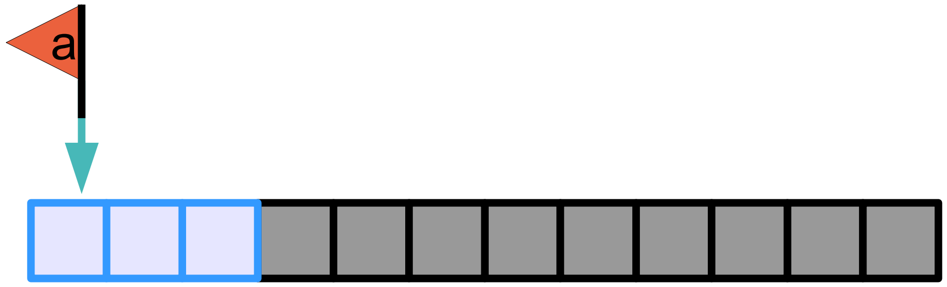
```
#include <iostream>

int main()
{
    int a[3];

    a[0] = 2;
    a[1] = 7;
    a[2] = a[0] * a[1];

    int *b = a;

    int *c = new int[3];

    c[2] = b[2] - a[0];

    c += 2;
    std::cout << *c << std::endl;

    // should have done:
    // c -= 2
    // delete [] c;
    return 0;
}
```



This time we will look at the memory allocation going on for a C-Style array

Arrays in Action (2)

```
#include <iostream>

int main()
{
    int a[3];

    a[0] = 2;
    a[1] = 7;
    a[2] = a[0] * a[1];

    int *b = a;

    int *c = new int[3];

    c[2] = b[2] - a[0];

    c += 2;
    std::cout << *c << std::endl;

    // should have done:
    // c -= 2
    // delete [] c;
    return 0;
}
```

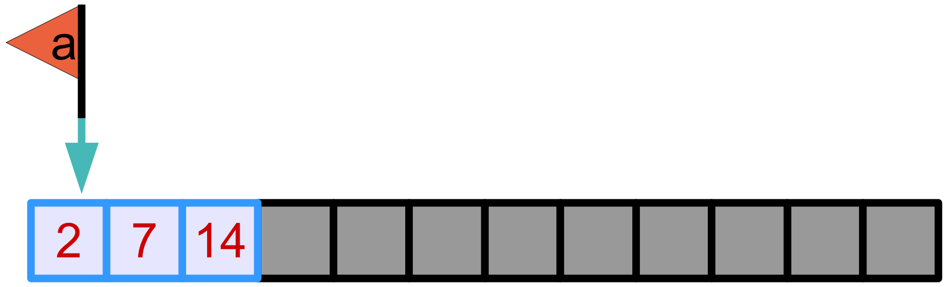


First, we declare an array

This allocates the memory requested and 'links' it to the variable a

Arrays in Action (3)

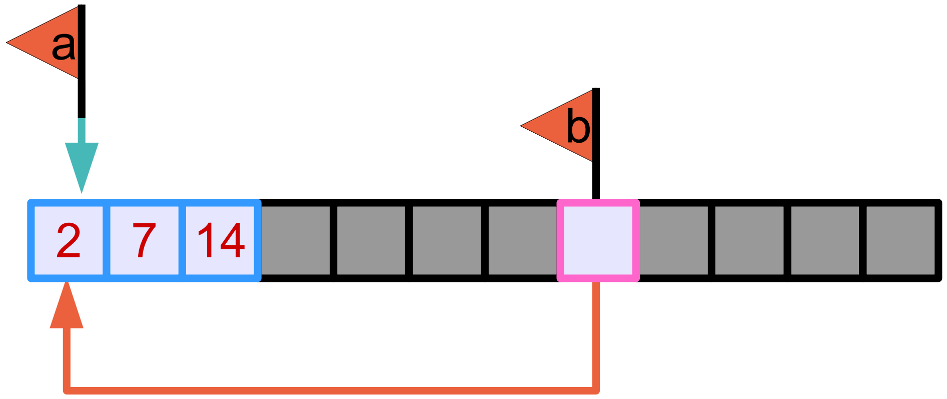
```
#include <iostream>

int main()
{
    int a[3];

    a[0] = 2;
    a[1] = 7;
    a[2] = a[0] * a[1];

    int *b = a;

    int *c = new int[3];

    c[2] = b[2] - a[0];

    c += 2;
    std::cout << *c << std::endl;

    // should have done:
    // c -= 2
    // delete [] c;
    return 0;
}
```



As always, the actual values are not initialised, so we now do that

Arrays in Action (4)

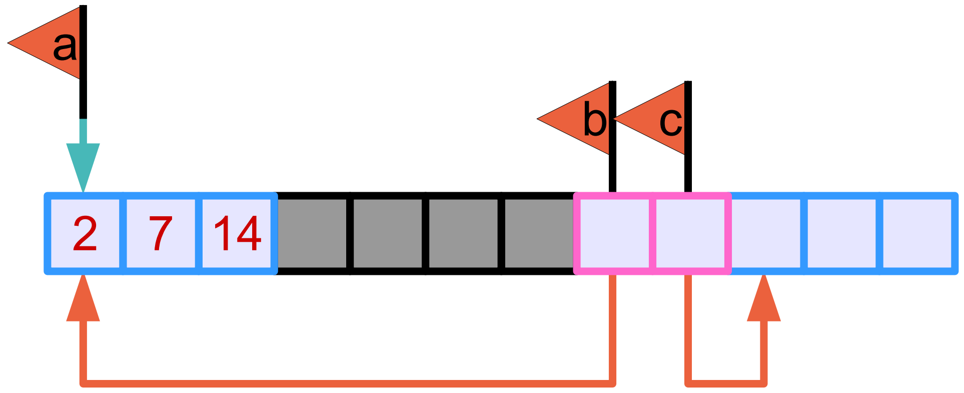
```
#include <iostream>

int main()
{
    int a[3];

    a[0] = 2;
    a[1] = 7;
    a[2] = a[0] * a[1];

    int *b = a;

    int *c = new int[3];

    c[2] = b[2] - a[0];

    c += 2;
    std::cout << *c << std::endl;

    // should have done:
    // c -= 2
    // delete [] c;
    return 0;
}
```



To show the similarities between arrays and pointers, we now create a basic pointer and assign it to point to the array

Arrays in Action (5)

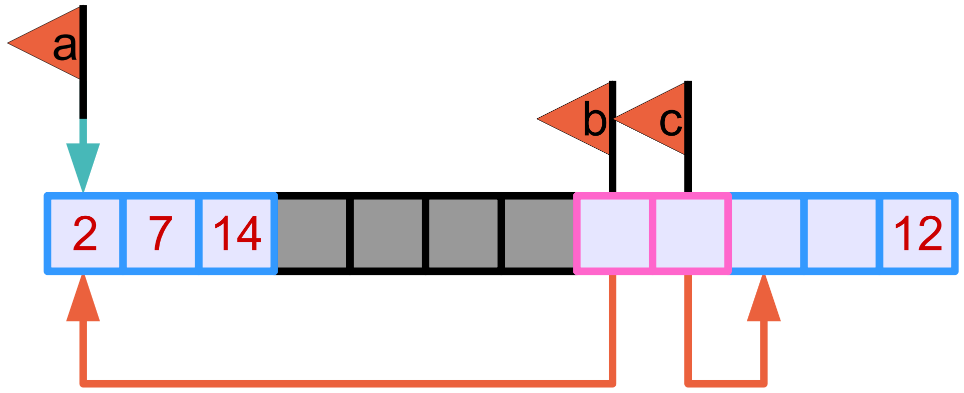
```
#include <iostream>

int main()
{
    int a[3];

    a[0] = 2;
    a[1] = 7;
    a[2] = a[0] * a[1];

    int *b = a;

    int *c = new int[3];

    c[2] = b[2] - a[0];

    c += 2;
    std::cout << *c << std::endl;

    // should have done:
    // c -= 2
    // delete [] c;
    return 0;
}
```



We now demonstrate the other way of creating arrays, by using the 'new' operator

Arrays in Action (6)

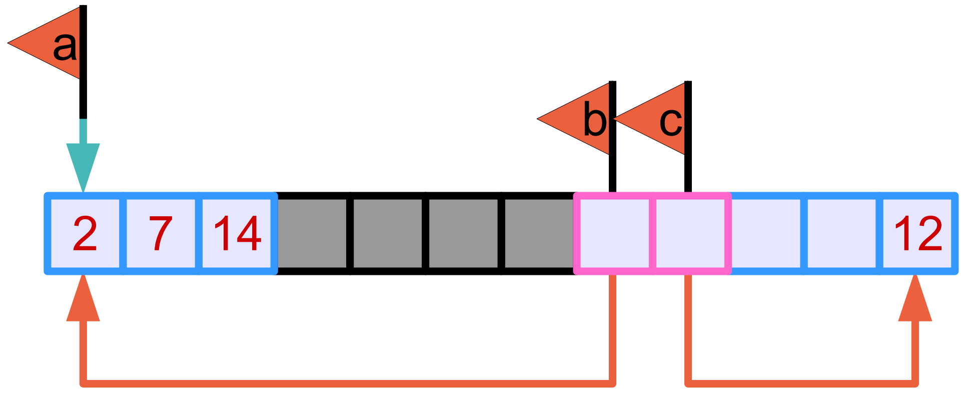
```
#include <iostream>

int main()
{
    int a[3];

    a[0] = 2;
    a[1] = 7;
    a[2] = a[0] * a[1];

    int *b = a;

    int *c = new int[3];

    c[2] = b[2] - a[0];

    c += 2;
    std::cout << *c << std::endl;

    // should have done:
    // c -= 2
    // delete [] c;
    return 0;
}
```



Again we fill the values in this new array

Arrays in Action (7)

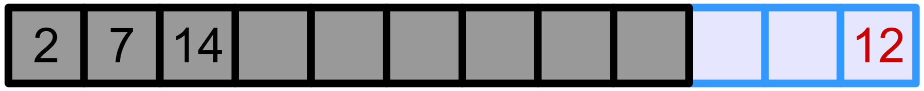
```
#include <iostream>

int main()
{
    int a[3];

    a[0] = 2;
    a[1] = 7;
    a[2] = a[0] * a[1];

    int *b = a;

    int *c = new int[3];

    c[2] = b[2] - a[0];

    c += 2;
    std::cout << *c << std::endl;

    // should have done:
    // c -= 2
    // delete [] c;
    return 0;
}
```



As can be done with pointers, if we increment the variable, we are incrementing the pointer, not the value pointed to

Arrays in Action (8)

```
#include <iostream>

int main()
{
    int a[3];

    a[0] = 2;
    a[1] = 7;
    a[2] = a[0] * a[1];

    int *b = a;

    int *c = new int[3];

    c[2] = b[2] - a[0];

    c += 2;
    std::cout << *c << std::endl;

    // should have done:
    // c -= 2
    // delete [] c;
    return 0;
}
```



Finally, after we go out of scope, we can see that because we didn't delete the 'new'd array, it's still present but the hard-coded array has been deleted.

Allocating Memory in Functions

```
#include <iostream>

void myfunc(int sz, int flag)
{
    // create an array
    int *arr = new int[sz];

    // do things depending on the flag
    if (flag == 0)
    {
        std:cout << "flag was 0" << std::endl;
        delete [] arr;
        return;
    }
    else if (flag == 1)
    {
        std:cout << "flag was 1 " << std::endl;
        delete [] arr;
        return;
    }

    delete [] arr;
}
```

Initialise an array based on an integer

Must remember to delete the array before each return statement

Must also delete the array before going out of scope

Allocating Memory in Classes

```
class DynamicArray {

public:
    DynamicArray( int sz )
    {
        // Initialise the arr_ member
        arr_ = new int[sz];
    }

    ~DynamicArray()
    {
        // Must remember to delete everything
        // that has been initialised
        delete [] arr_;
    }

private:
    int *arr_ = nullptr;
};
```

Best to allocate any memory in the constructor and delete it in the destructor

Drawbacks of Using C-Style Arrays

- As with raw pointers, there are several reasons why you shouldn't use C-Style arrays:
    - They are of fixed size unless being allocated with 'new'
    - If allocated with 'new', must remember to 'delete' them
    - Not as flexible or powerful as containers
    - No protection for the array pointer changing
    - No protection for going 'out of bounds' of the allocated memory
    - You can't easily determine the size of an array
- And so, as before:

Only use C-Style arrays when you absolutely have to – C++ containers are better in almost every practical way!