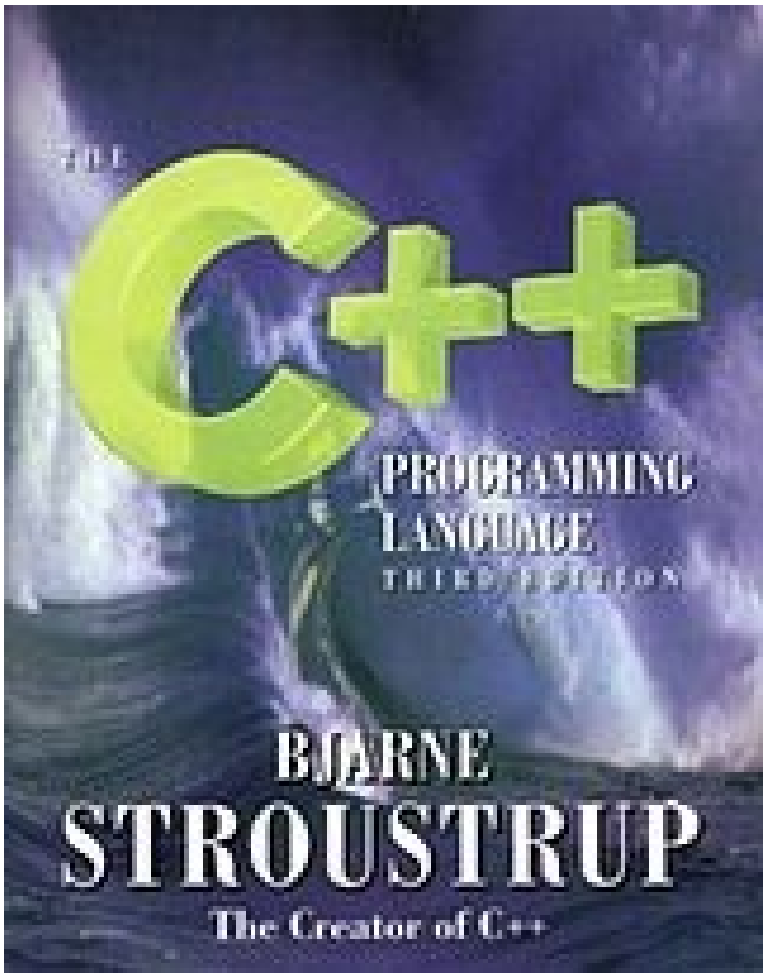
Introduction to C/C++ Day 2

Mark Slater

UNIVERSITY OF BIRMINGHAM

Overview

1. Quick Review
2. Pointers and References
3. Arrays



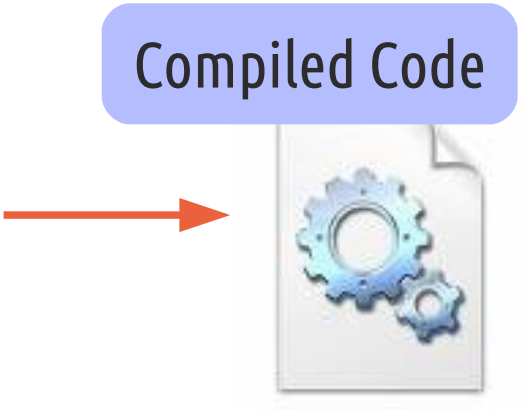
1. Quick Review

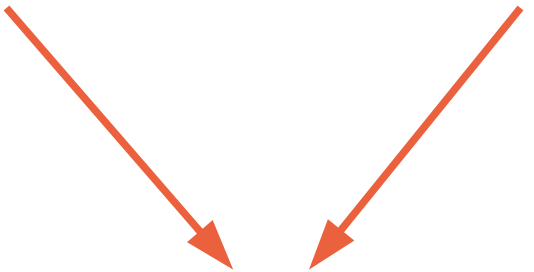
Recap - Writing and Compiling Code

It would probably be beneficial to cover the main points from the last session. First, building your program from your source:

```
#include <iostream>
int main()
{
// Read and print three
// floating point numbers
std::cout << "Give 3 nums" << std::endl;
float a, b, c;
std::cin >> a >> b >> c;
std::cout << "Your gave... ";
std::cout << a << ", " << b << ", " << c
<< std::endl;
return 0;
}
```

Raw Code





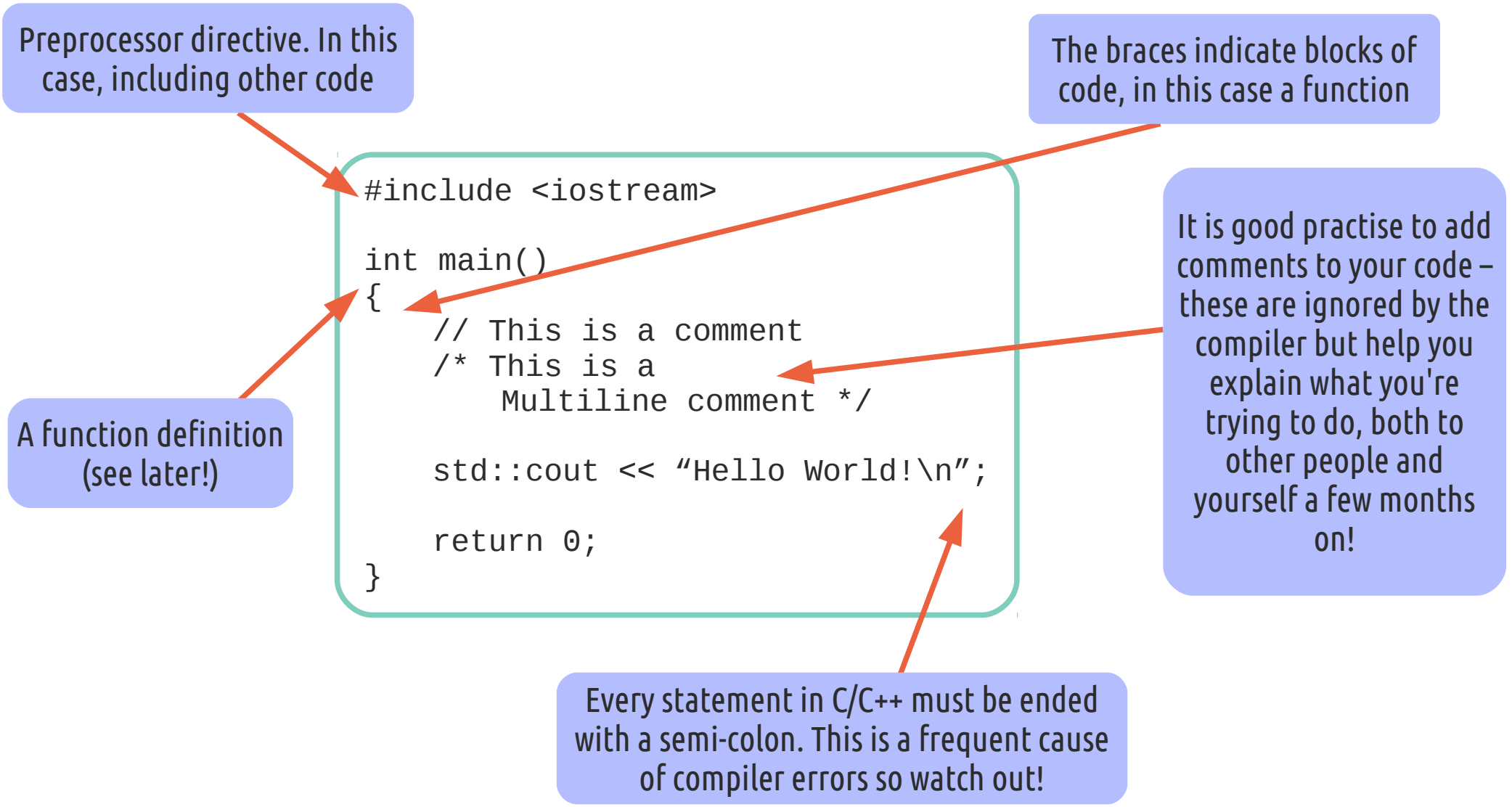
Executable

In this session you will extend on the basics of using a single code file

You will start to use header files and create library files that must be linked to produce the final program

Recap - Basic Syntax of a C/C++ Program

We covered the basic syntax of C++, including many keywords and operators



Recap – Objects and Operators

Variable declaration and the consequence for memory was covered as well as the use of operators

- A boolean (true/false) – 'bool'
- Integer number – 'int'
- Floating point number – 'float'
- Double precision number – 'double'
- Single Character/0-255 number – 'char'

Note that numerical variables can also be 'signed' (default) or 'unsigned'

We will start covering significantly more complicated objects, but remember – THEY ARE JUST LIKE THE BASIC VARIABLES!

- Multiplication: a * b
- Addition: a + b
- Increment: a++
- Bitwise shift/stream: a << b
- Modulus: %
- Array: []

Operators have precedence, just like in maths, as well as associativity (left ↔ right) and arity (# of operands)

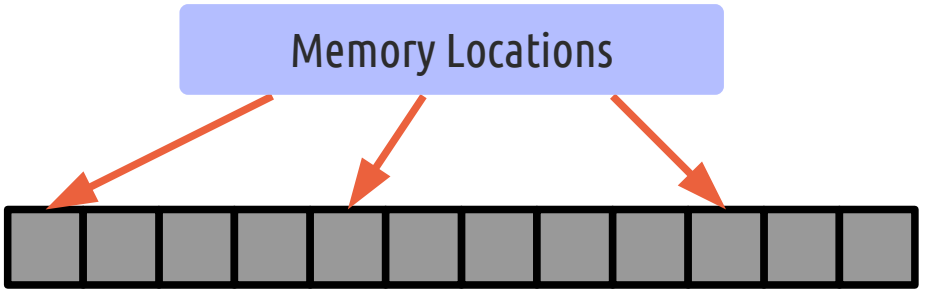
2. Pointers and References

Introducing Pointers

- We will now move on to what can be the most troublesome, but is also one of the most powerful aspects of C++: Pointers. They get a lot of bad press, mostly due to their misuse, but they are quite simple to understand
- Essentially, a pointer is an object that 'points' to another memory location. Therefore, the variable itself doesn't contain the data but only a reference to memory
- You can create pointers to any object available, but to create a persistent object that doesn't go 'out-of-scope', use the 'new' operator. However, you must remember to call the 'delete' operator on it after you've finished!

Pointers in Action (1)

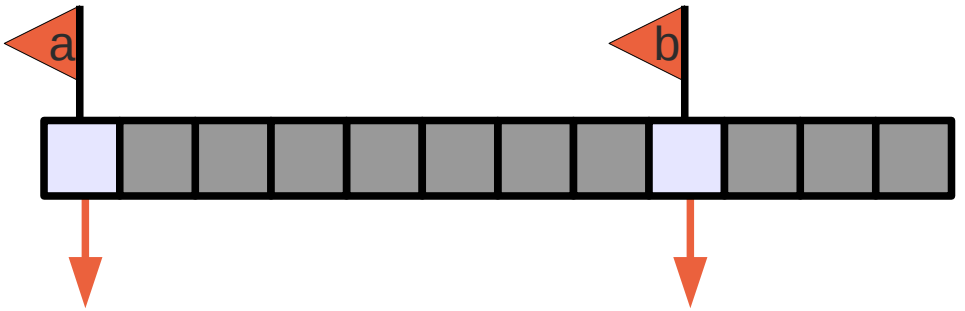
```
#include <iostream>

int main()
{
    int *a, *b;

    a = new int;
    b = new int;

    *a = 5;
    *b = 7;

    int c = (*a) + (*b);
    int *d = &c;

    delete a;
    ++a;

    std::cout << d << ":  " <<
        *d << std::endl;

    return 0;
}
```



To demonstrate how pointers work, we will return to our basic memory picture shown before

Pointers in Action (2)

```
#include <iostream>

int main()
{
    int *a, *b;

    a = new int;
    b = new int;

    *a = 5;
    *b = 7;

    int c = (*a) + (*b);
    int *d = &c;

    delete a;
    ++a;

    std::cout << d << ":   " <<
        *d << std::endl;

    return 0;
}
```

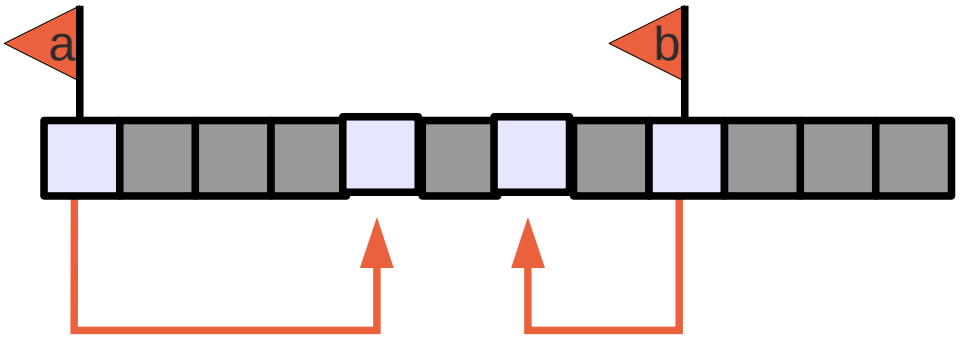


First, two pointers to ints (a and b) are declared

Note that, at present, they point to "nothing" and would (hopefully!) produce a crash if accessed at this point – *this is the primary cause of problems!*

Pointers in Action (3)

```
#include <iostream>

int main()
{
    int *a, *b;

    a = new int;
    b = new int;

    *a = 5;
    *b = 7;

    int c = (*a) + (*b);
    int *d = &c;

    delete a;
    ++a;

    std::cout << d << ":   " <<
        *d << std::endl;

    return 0;
}
```

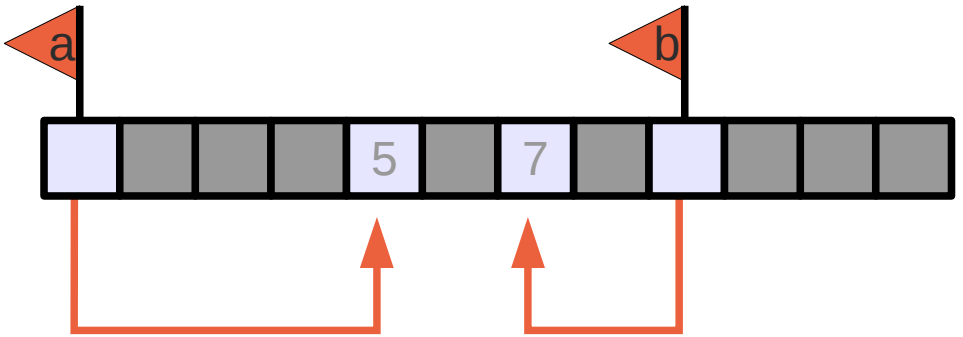


Next, *the memory* for two integers are allocated and the addresses of these allocations are assigned to the pointers

At this point, a and b now point to useful memory locations, but the values are still not initialised (i.e. junk!)

Pointers in Action (4)

```
#include <iostream>

int main()
{
    int *a, *b;

    a = new int;
    b = new int;

    *a = 5;
    *b = 7;

    int c = (*a) + (*b);
    int *d = &c;

    delete a;
    ++a;

    std::cout << d << ":  " <<
        *d << std::endl;

    return 0;
}
```

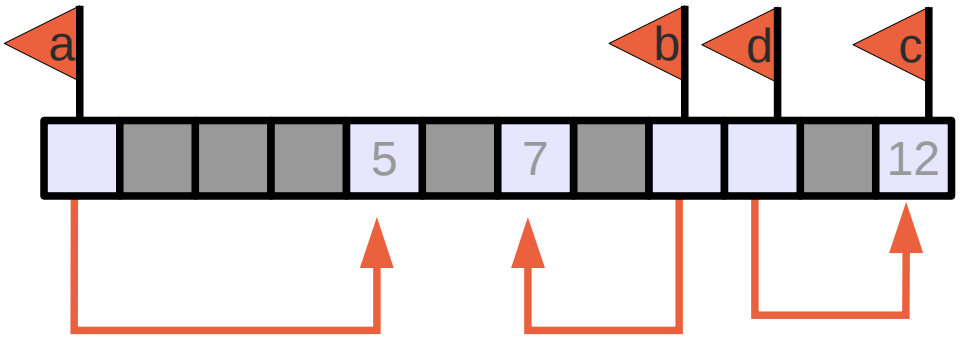


To actually assign values to the integers, we *dereference* the pointers

This essentially means 'assign this value to the object that is pointed to'

Pointers in Action (5)

```
#include <iostream>

int main()
{
    int *a, *b;

    a = new int;
    b = new int;

    *a = 5;
    *b = 7;

    int c = (*a) + (*b);
    int *d = &c;

    delete a;
    ++a;

    std::cout << d << ":   " <<
        *d << std::endl;

    return 0;
}
```

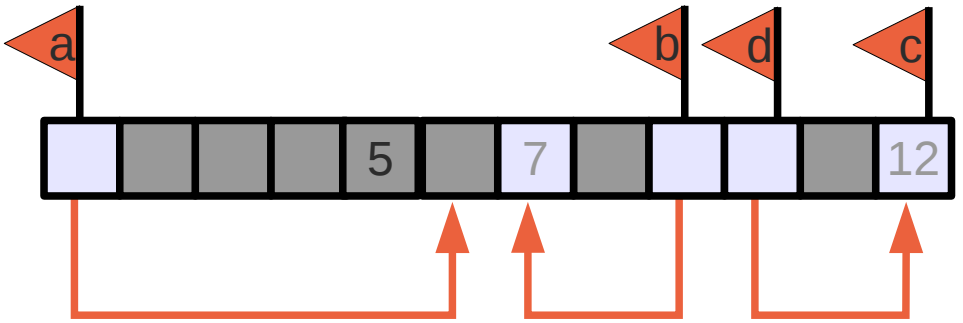


Next, we create a normal integer variable and assign the sum of the two integers pointed to by a and b

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*

Pointers in Action (6)

```
#include <iostream>

int main()
{
    int *a, *b;

    a = new int;
    b = new int;

    *a = 5;
    *b = 7;

    int c = (*a) + (*b);
    int *d = &c;

    delete a;
    ++a;

    std::cout << d << ":   " <<
        *d << std::endl;

    return 0;
}
```



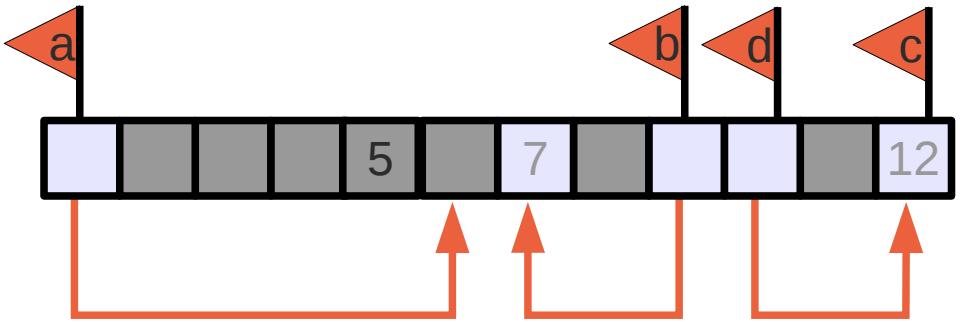
Just to show what happens, we now delete the memory allocated to the pointer a

To show that this doesn't affect the variable itself, we increment it

This has the affect of incrementing the address, not what was pointed to

Pointers in Action (7)

```
#include <iostream>

int main()
{
    int *a, *b;

    a = new int;
    b = new int;

    *a = 5;
    *b = 7;

    int c = (*a) + (*b);
    int *d = &c;

    delete a;
    ++a;

    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, *b;

    a = new int;
    b = new int;

    *a = 5;
    *b = 7;

    int c = (*a) + (*b);
    int *d = &c;

    delete a;
    ++a;

    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!

References

- We saw last week that functions could not change objects passed to them as they only had *copies* of the object. This was called *'passing by value'.*
- Pointers enable us to pass the object itself but this is not entirely trivial. In C++, the concept of references was introduced which was simply applying a new label to an object. This is termed *'passing by reference'.*

```
#include <iostream>

void change( double& a, double& b )
{
    a = 10;
    b = 2;
}

int main()
{
    int a = 43, b = 21;

    change(a, b);

    return 0;
}
```

The function definition has the *references* of two doubles, NOT just doubles

After calling the function, the values of a and b are changed

Note that this is different to the 'address-of' operator – double& is a different object type!

Sorting an Array (Ex. 1)

- In preparation for the homework, we will be adding a 'sort' function to your code

- The first step in this process is to provide a function that swaps two given numbers

By 'passing by reference', add a function to your code that takes two doubles and swaps them

Some Improvements (Ex. 2)

- In addition to the 'swap' function, we can also use 'passing by reference' to improve the maths function code you already have. Depending on how you coded it, to get both roots from the quadratic function, you will probably have had to call it twice as it only returns one number
- To get around this, we can pass pointers to the function that get filled with both positive and negative solutions. This also allows the return value of the function to be a boolean and therefore show if a successful solution was found rather than checking a set 'error value'

Try to alter the maths functions you currently have in your code to pass variables by reference for the answer. Also, where necessary make the return value indicate a valid answer has been returned

3. Arrays

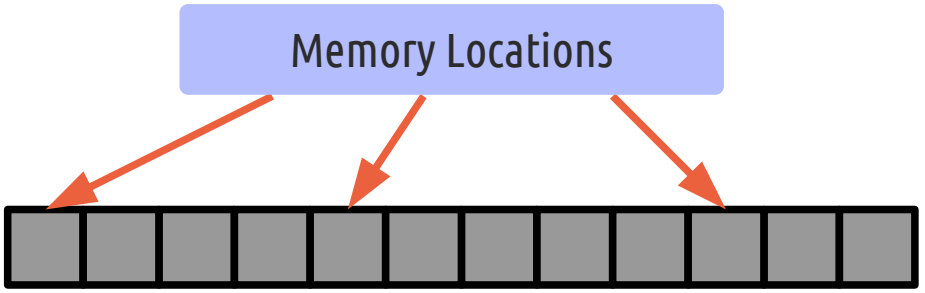
Introducing C-Style Arrays

- Arrays that hold multiple values are intimately linked to pointers due to the way they are handled

- 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

Arrays in Action (1)

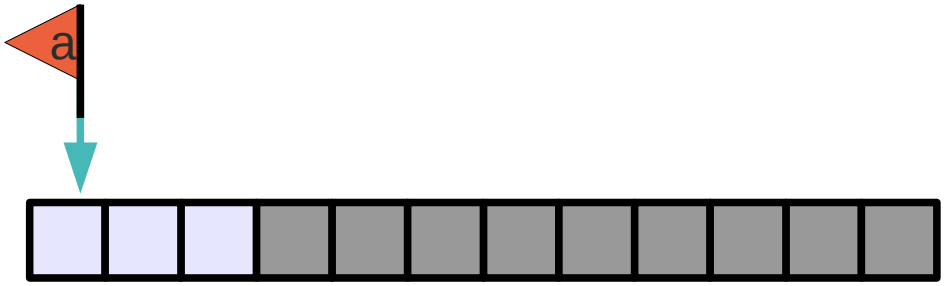
```
#include <iostream>

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;
}
```



And (for the last time!) here is an example of the memory allocation going on behind the scenes

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;
}
```

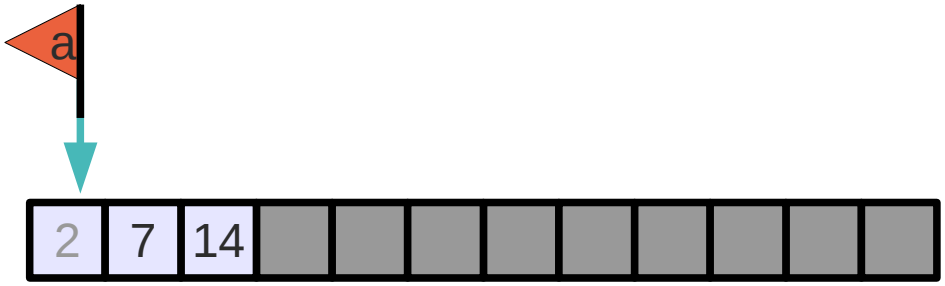


First, we declare an array

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

Arrays in Action (3)

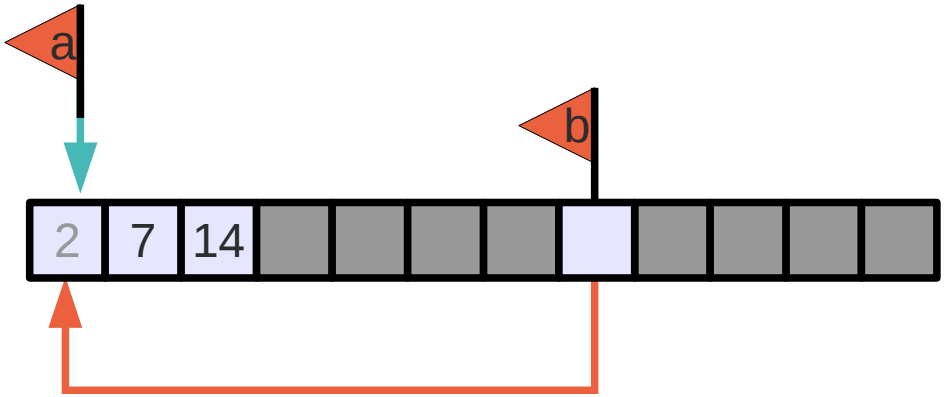
```
#include <iostream>

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)

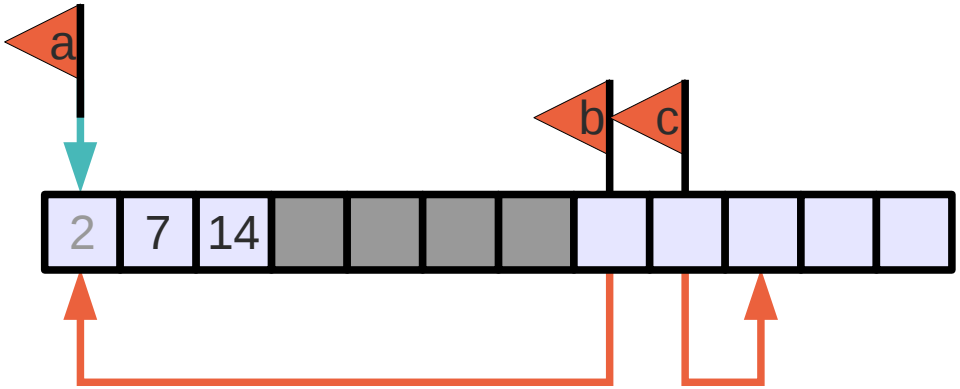
```
#include <iostream>

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)

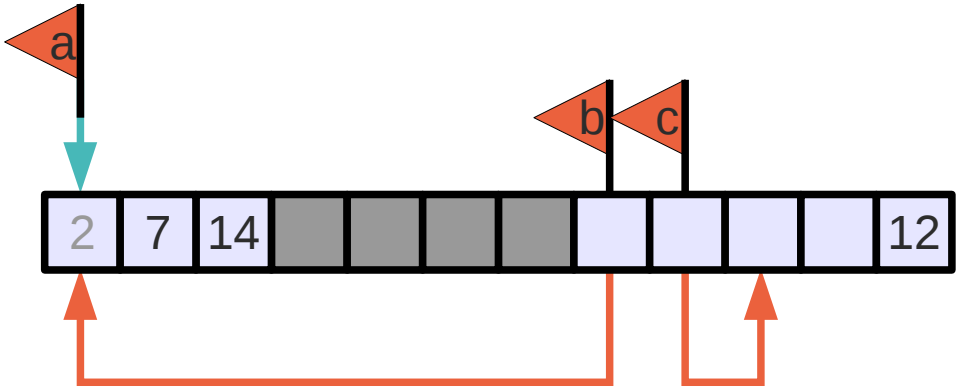
```
#include <iostream>

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)

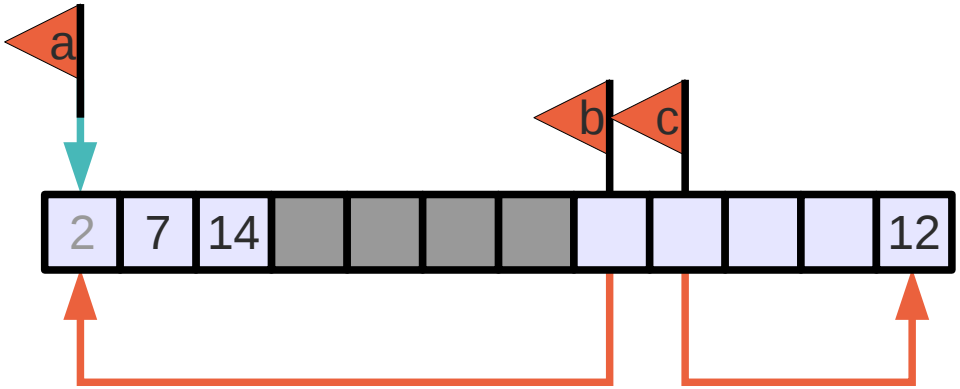
```
#include <iostream>

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 with pointers, if we increment the variable, we are incrementing the pointer, not the value pointed to

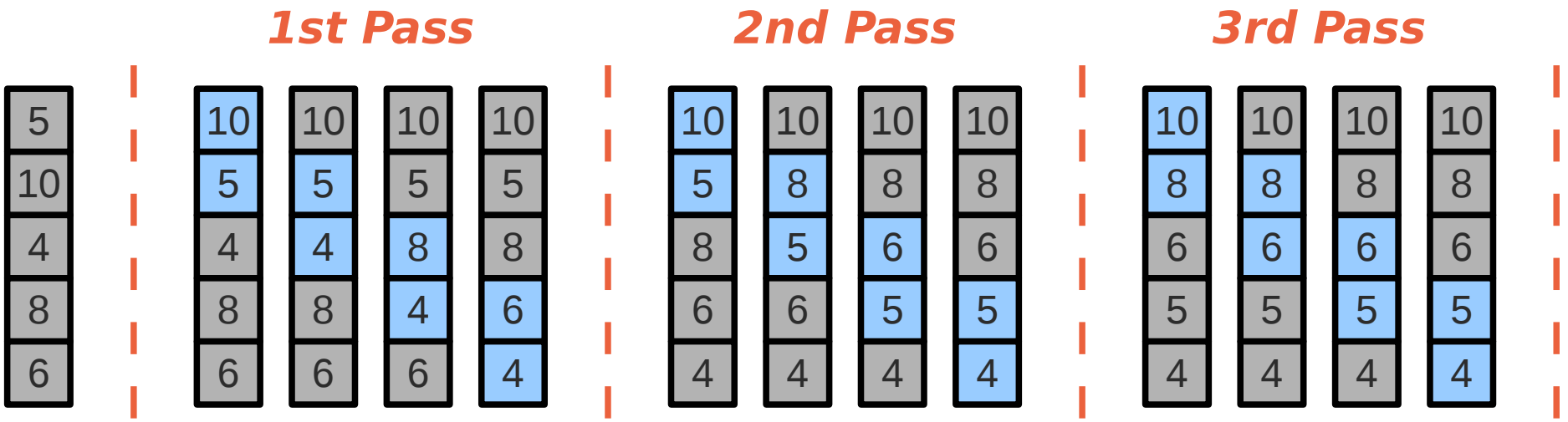
Arrays in Action (8)

```
#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

Sorting an Array (Ex. 3)

- Before we move on to adding some statistics functions to the code you got already, we can now return to our sorting function. To do this, we will implement a useful (if not overly efficient!) sorting algorithm called a 'Bubble Sort':



Create a function that takes an array and sorts it using the above algorithm and your 'swap' function previously

A Few Points

- Before we move on to makefiles and using libraries and header files, there are just a couple of more advanced points you may want to be aware of:

1. Heap vs. Stack

   When local objects are created, they are on the 'stack' part of memory. If the 'new' operator is used, these are placed on the 'heap'

2. Addresses are not quite the same as References

   Addresses (e.g. &a) are different to passing-by-reference. The latter involves the symbol table of the compiler rather than an actual address

3. Pointers and Arrays are different

   Though they can mostly be used interchangeably, there is a difference between them. See http://www.lysator.liu.se/c/c-faq/c-2.html

4. '\n' and std::endl are different

   std::endl also causes a stream to be 'flushed'

Don't worry about most of these – as long as you can do the basics, you'll be OK!