

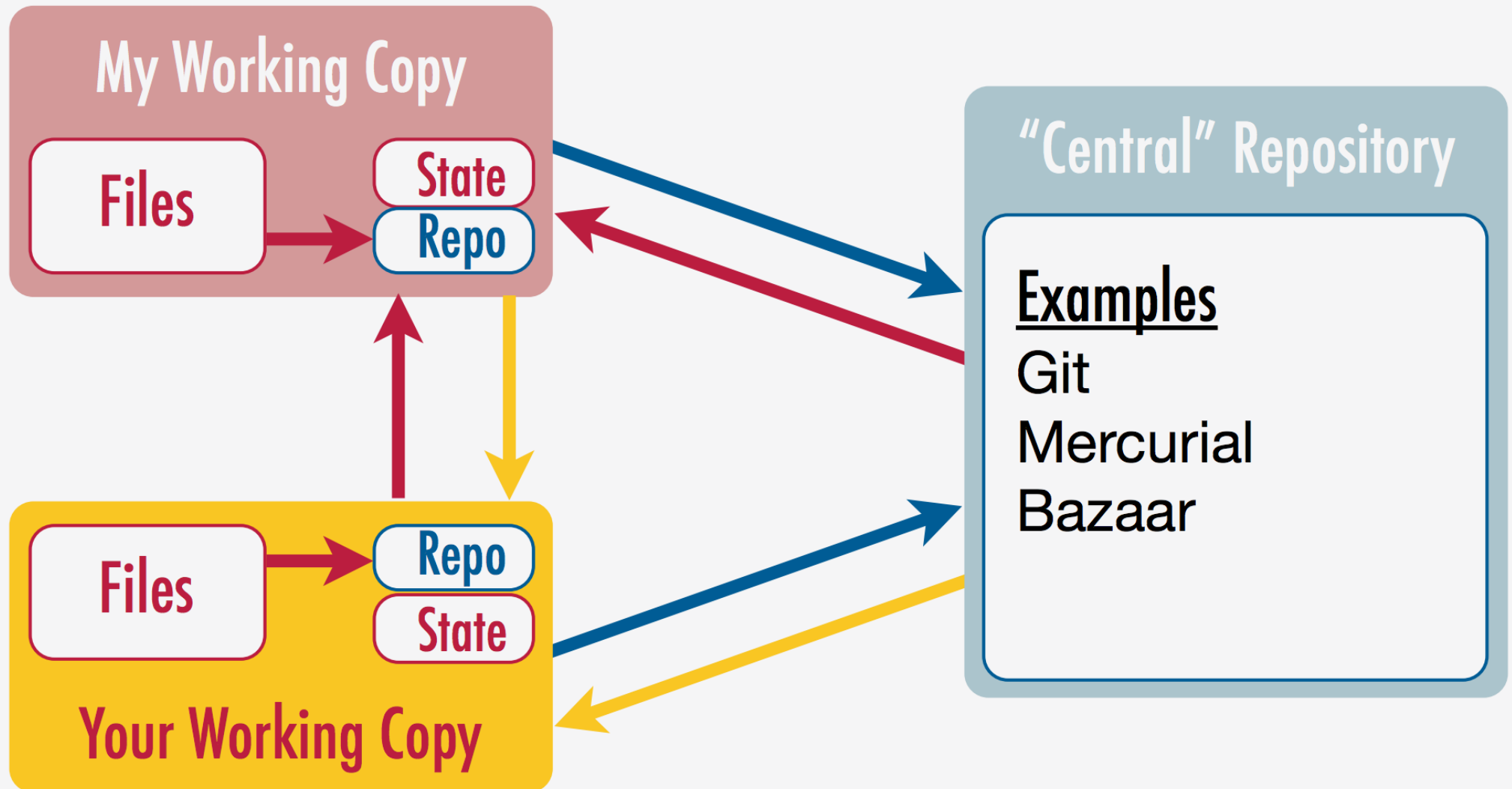
# Quick Recap

---

# Recap – Git Version Control

---

We covered how to use Git for Version Control:



# Recap - Writing and Compiling Code

C++ code can be written using any text editor, but to create the actual programs requires a compiler that creates the machine-readable code

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    // Read and print three
```

```
    // floating point numbers
```

```
    std::cout << "Give 3 nums" << std::endl;
```

```
    float a{0}, b{0}, c{0};
```

```
    std::cin >> a >> b >> c;
```

```
    std::cout << "You gave... ";
```

```
    std::cout << a << ", " << b << ", "
```

```
        << c << std::endl;
```

```
}
```

Compiled Code



Additional Libs



Raw Code

Executable



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

We covered the basic syntax of C++

Preprocessor directive to include other code - see later!

A function definition - see later

```
#include <iostream>
int main()
{
    // This is a comment
    /* This is a
       Multiline comment */

    std::cout << "Hello World!\n";
}
```

It is good practise to add comments to your code - these are ignored by the compiler but help you explain what you're trying to do, both to other people and yourself a few months on!

The braces indicate blocks ('scope') of code, in this case a function

Every statement in C/C++ must be ended with a semi-colon. This is a frequent cause of compiler errors so watch out!

# Recap - Types, Objects, Values and Variables

---

We covered the basics of how data is stored in C++:

- Types – How to interpret data in a memory location ('object') and what operations can be performed by it
- Object – Defined area of memory that holds the data ('values') associated with a type
- Value – Actual data/bits in memory interpreted by the 'type'
- Variable – A flag or name of an area of memory ('object')

As well as how they are manipulated using operators:

- Assignment: `a = b`
- Dec/Increment: `a--`, `a++`
- Bitwise shift/stream: `a << b`, `a >> b`
- Modulus: `%`
- Array: `[]`

# Recap – Program Flow

---

We looked at conditionals and loops:

```
switch (flag)
{
    case 0:
        // Do something for this value
        break;

    case 1:
        // Do something else for this value
        break;

    default:
        // Do something for all other values
        break;
}
```

```
if (a == b)
{
    // Do something...
}
else
{
    // Do something else instead
}
```

```
int i{0};
while (i < 10)
{
    // Do something 10 times
    ++i;
}
```

```
for (int i{0}; i < 10; ++i)
{
    // Do something 10 times
}
```

# Recap – Vectors

---

We also looked at the first basic container – a vector:

```
#include <vector>
#include <string>
#include <iostream>

int main()
{
    // Construct a vector
    std::vector<double> vec = {1.2, 3.4, 5.6};

    // print out the vector size (3)
    std::cout << vec.size() << std::endl;

    // add a few elements
    vec.push_back(7.8);
    vec.push_back(9.1);

    // vector size (5)
    std::cout << vec.size() << std::endl;

    // remove an element
    vec.pop_back();

    // vector size (4)
    std::cout << vec.size() << std::endl;

    // loop over the vector using an index counter
    for (size_t i{0}; i < vec.size(); i++)
    {
        std::cout << "Index: " << i << " " << vec[i] << std::endl;
    }
}
```

# Recap – Command Line Arguments

---

Finally, we covered getting information into your program from the user using command line arguments:

## Reading arguments in C++

---

- Due to backward compatibility with C, the way that these appear in **main()** are as two function arguments:
  - **argc** is an integer - the number of arguments
  - **argv** is a C-style array of C-style strings - the arguments themselves
- These are rather fiddly to work with, so best to immediately convert them into a more usable form, a `std::vector` of `std::string` objects:

```
int main(int argc, char* argv[])  
{  
    std::vector<std::string> cmdLineArgs { argv, argv+argc };  
}
```

- We can then loop over and/or access the individual arguments as with any `std::vector`



# New Github Classroom for Day 2

---

- As mentioned last week, there will be a new 'base' repo for each day
- This will contain the 'ideal solution' to the previous day's exercises
- Clone the created repository from the link on the course pages just as with last week and you're good to go!

# C++ Functions

---

Mark Slater

UNIVERSITY OF  
BIRMINGHAM

# Program Flow – Functions (1)

---

- As mentioned before, it's good coding practice to reuse as much code as possible. The main way of doing this in C++ is through the use of functions that can then be called in other code blocks. Just like a variable, a function must be declared before it can be used:

*<return\_type> <function\_name> ( <arguments> ) { <code\_block> }*

- After declaration, the function is called by just giving the function name and the required arguments in brackets. At this point, the program flow jumps to this function until it hits a 'return' statement or the end of its scope
- An important point to remember is that variables are passed '**by value**' – i.e. the object value passed is copied to the new variable, the object itself is not sent to the function
- This is important to remember when dealing with 'large' objects like strings or vectors – a copy will be made which can be slow!

# Program Flow – Functions (2)

Declare and define a function that multiplies two numbers together and returns the result

Note that you have already encountered a function: the 'main' function. This is a special function that is where the program starts, but it behaves in the same way

```
#include <iostream>

double multiply( const double first, const double second )
{
    return first * second;
}

void print( const double value )
{
    std::cout << "Result: " << value << std::endl;
}

int main()
{
    double a{43.0},
    double b{21.0},

    double c{ multiply(a, b) };
    print(c);

    print( multiply(a, c) );
}
```

This function prints the given number with an additional message

Note that, the variables here **WILL NOT BE CHANGED** as the values are **COPIED** to the function where new objects are created

# Documenting Function Behaviour

---

- Though we have already highlighted the importance of documenting your code in comments, with functions there are more things to consider
- We will be covering one popular style of comment documentation in detail later, but for the moment, make sure you are detail:
  - A brief, one line description of the function
  - If necessary, more info about how to use the function and what it does
  - The arguments to the function
  - What it returns

```
double multiply( double a, double b)
{
    /* multiply two values together and return the result

    double a: First number to multiply
    double a: Second number to multiply

    return: The value of the product of a and b
    */

    return a * b;
}
```

# Exercise - Start Using Functions

---

- Now you have learnt what functions are and why they are used, we can now start using them in your cipher code
- For the next exercise, move your transliteration code from within the 'while' loop into a function called 'transformChar' that takes a char and returns a string after applying the transliteration:

```
std::string transformChar( const char in_char )
```

- After removing code from the 'while' loop, in its place you will need to add a call to the 'transformChar' function and add the return value to the 'inputText' string
- Your new function will have to be put before your main function where it's referenced