Programming and Data Analysis for Scientists

C++ Workshop 4

Functions in C++ and additional topocs

Prof Stephen Clark

University of BRISTOL

SCIF20002

Functions in C++ and some extras

The purpose of this workshop is to introduce two key concepts in the C++ programming language. The *learning objectives* are:

- To understand how to declare and implement *functions*.
- To see how to overload functions and pass-by-reference arguments.

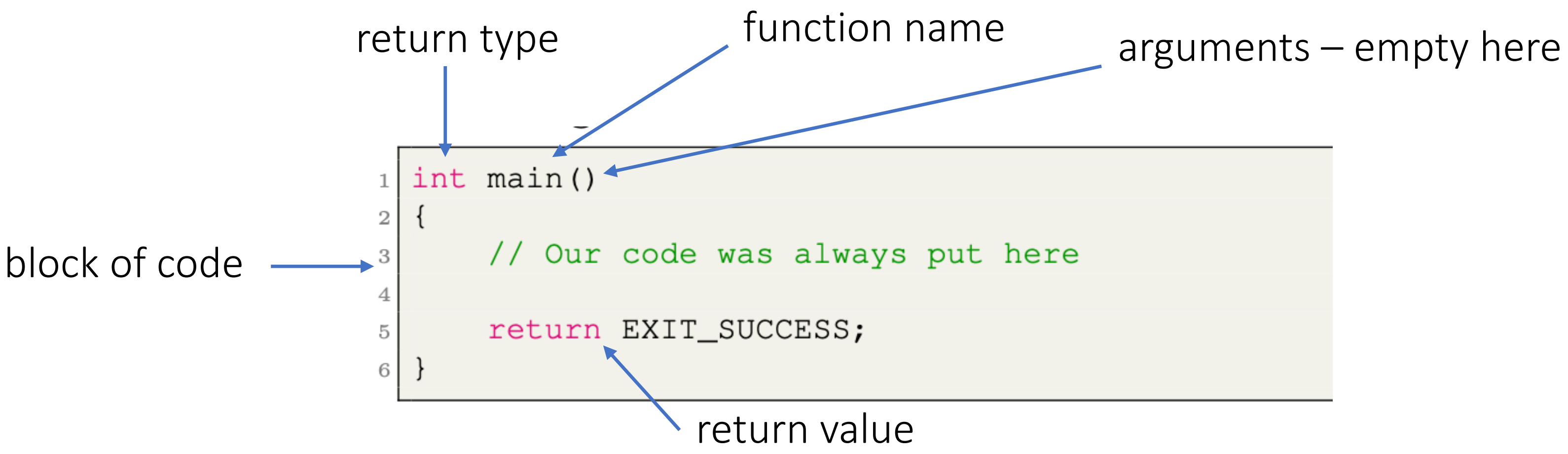
Additional useful topics …

- To be able to implement random number generation.
- To understand how implement multi-file compilations.

Functions in C++

Modern modular programming requires a sub-division of tasks into reusable units called *functions*. We have already been using the basic syntax:

All C++ programs start with a **mandatory** `main()` function:



return type

function name

arguments – empty here

block of code

```
int main()
{
    // Our code was always put here

    return EXIT_SUCCESS;
}
```

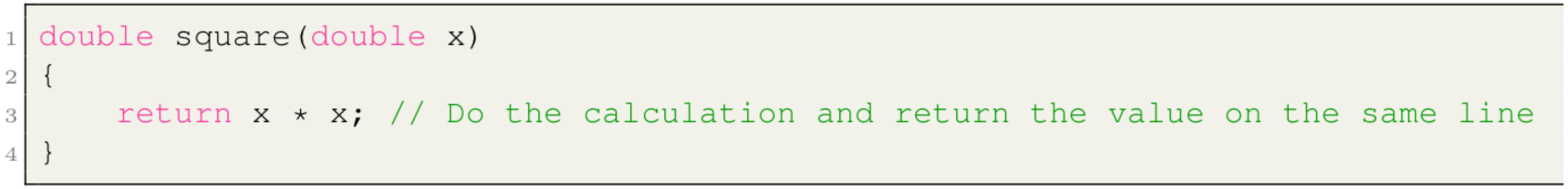
return value

Let’s look at some other examples …

Simple example functions

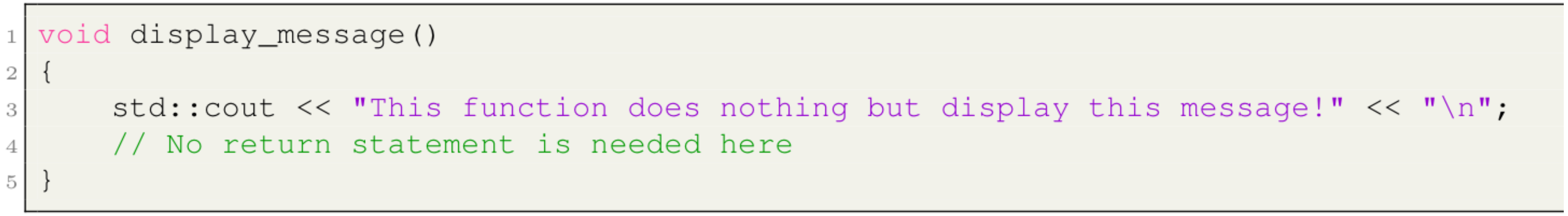
Here are three examples:

Just squares a `double`



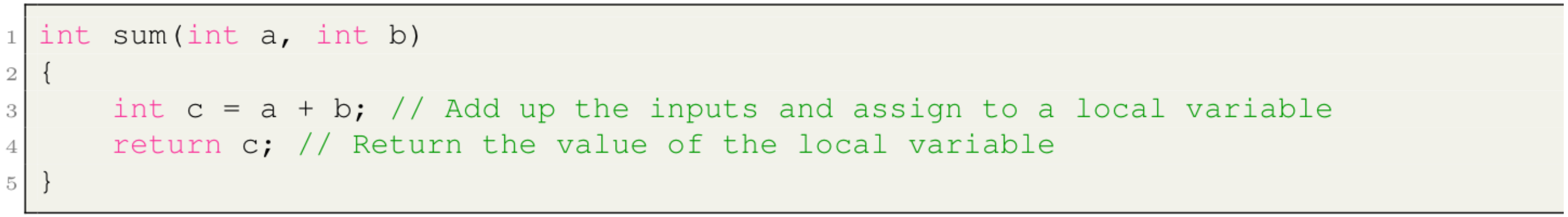
```cpp
double square(double x)
{
    return x * x; // Do the calculation and return the value on the same line
}
```

Just displays a message, has no return value



```cpp
void display_message()
{
    std::cout << "This function does nothing but display this message!" << "\n";
    // No return statement is needed here
}
```

Takes two inputs



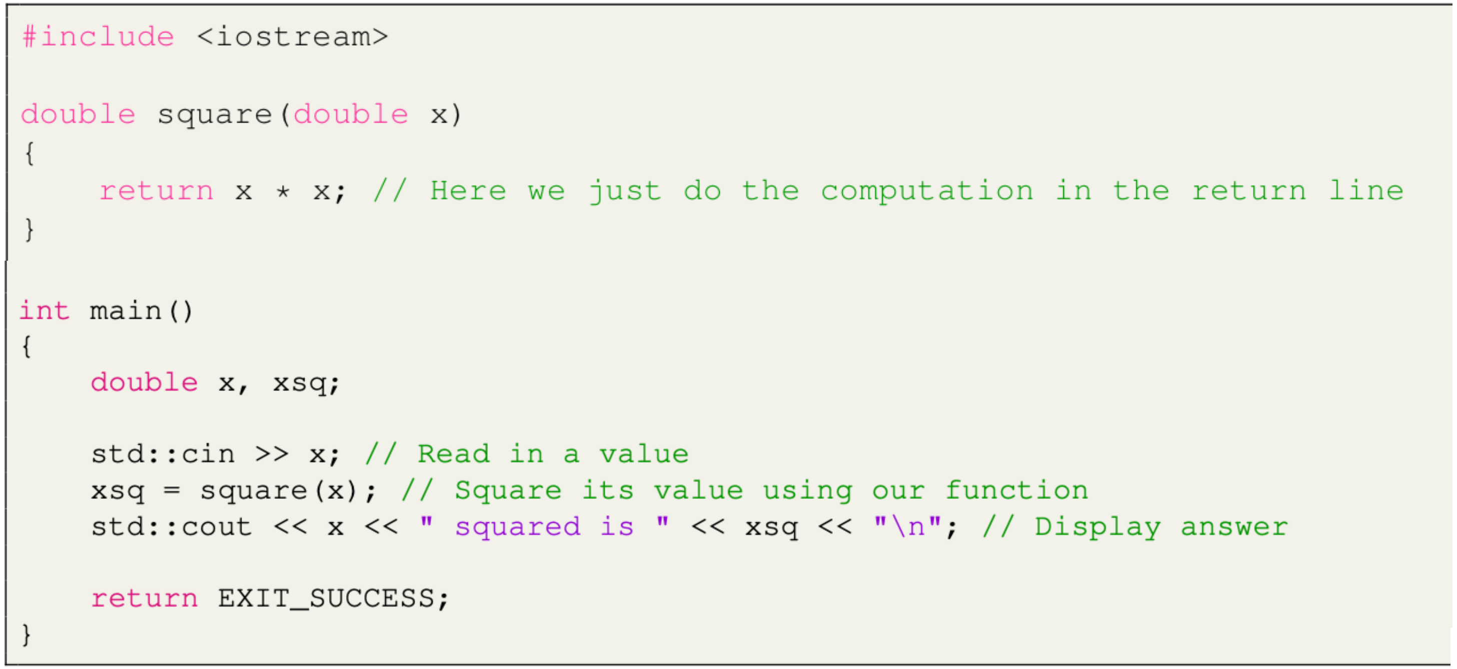
```cpp
int sum(int a, int b)
{
    int c = a + b; // Add up the inputs and assign to a local variable
    return c; // Return the value of the local variable
}
```

Calling a function

Declare and define the function outside and before `main()`:

Declare and define

Then use inside `main()`



```cpp
#include <iostream>

double square(double x)
{
    return x * x; // Here we just do the computation in the return line
}

int main()
{
    double x, xsq;

    std::cin >> x; // Read in a value
    xsq = square(x); // Square its value using our function
    std::cout << x << " squared is " << xsq << "\n"; // Display answer

    return EXIT_SUCCESS;
}
```

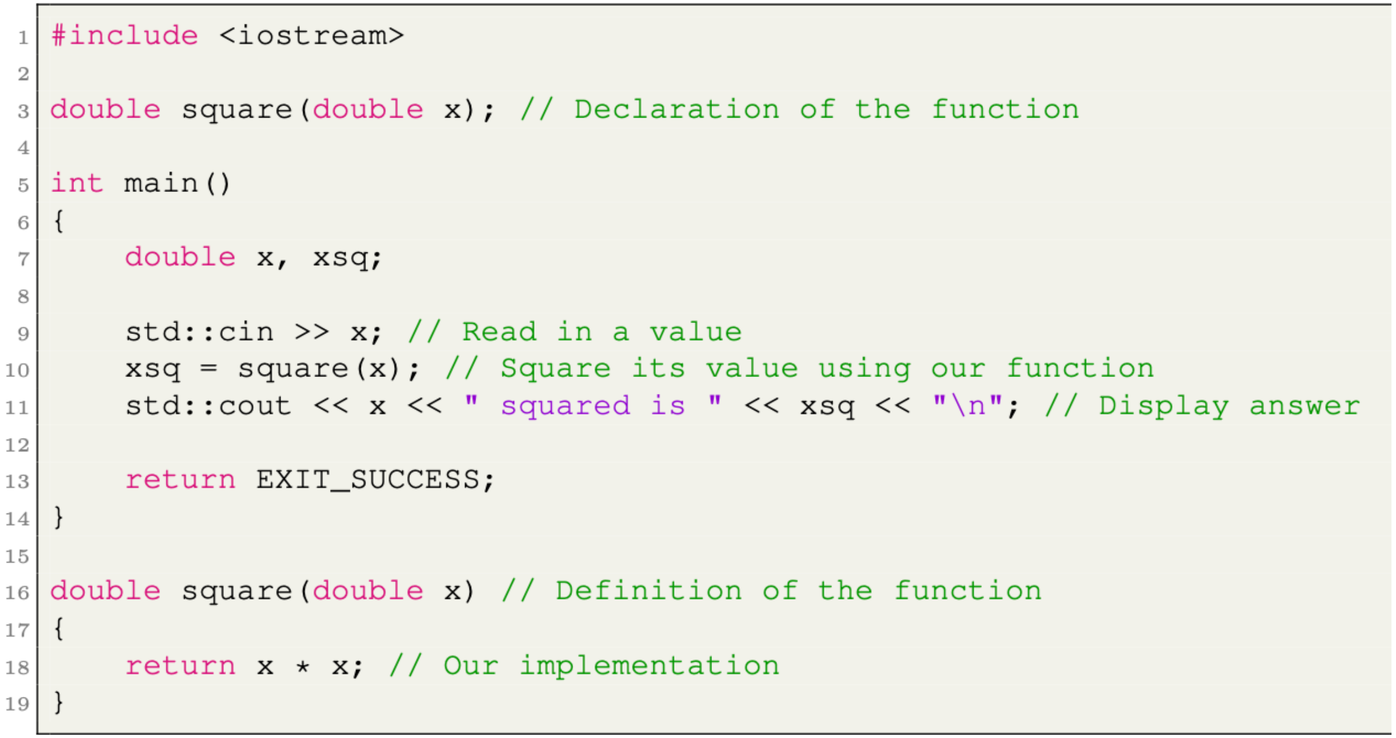
Calling a function

All functions must be declared before their use. Often this requires that we separate declarations from the definition (implementation):

Declare

Then use inside `main()`

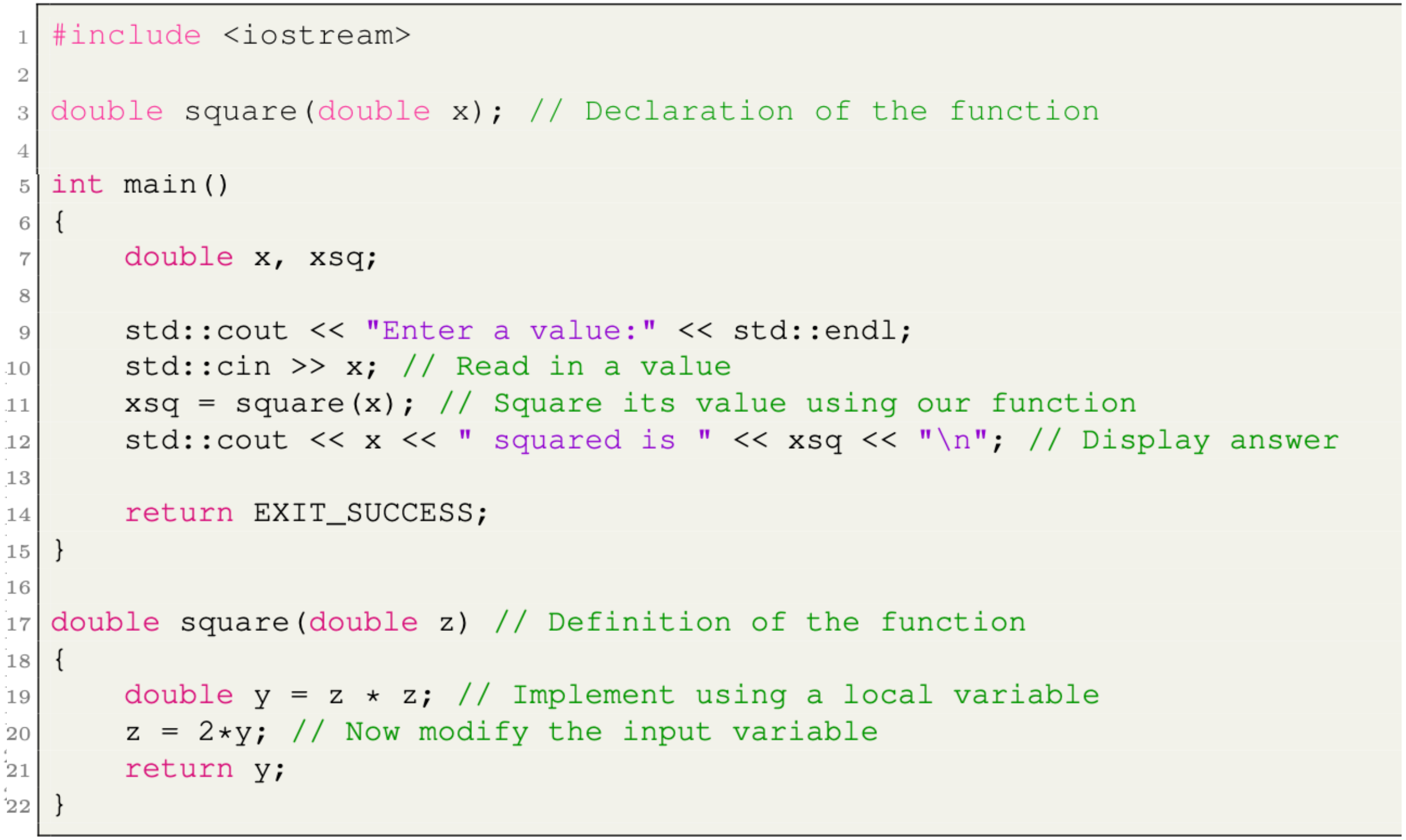
Define



```cpp
#include <iostream>

double square(double x); // Declaration of the function

int main()
{
    double x, xsq;

    std::cin >> x; // Read in a value
    xsq = square(x); // Square its value using our function
    std::cout << x << " squared is " << xsq << "\n"; // Display answer

    return EXIT_SUCCESS;
}

double square(double x) // Definition of the function
{
    return x * x; // Our implementation
}
```

Variable scope

Variables are passed-by-value in C/C++, so all variables are local in scope:



```
#include <iostream>

double square(double x); // Declaration of the function

int main()
{
    double x, xsq;

    std::cout << "Enter a value:" << std::endl;
    std::cin >> x; // Read in a value
    xsq = square(x); // Square its value using our function
    std::cout << x << " squared is " << xsq << "\n"; // Display answer

    return EXIT_SUCCESS;
}

double square(double z) // Definition of the function
{
    double y = z * z; // Implement using a local variable
    z = 2*y; // Now modify the input variable
    return y;
}
```

Declare

Deliberately called input a different name

Won’t change x passed to it!

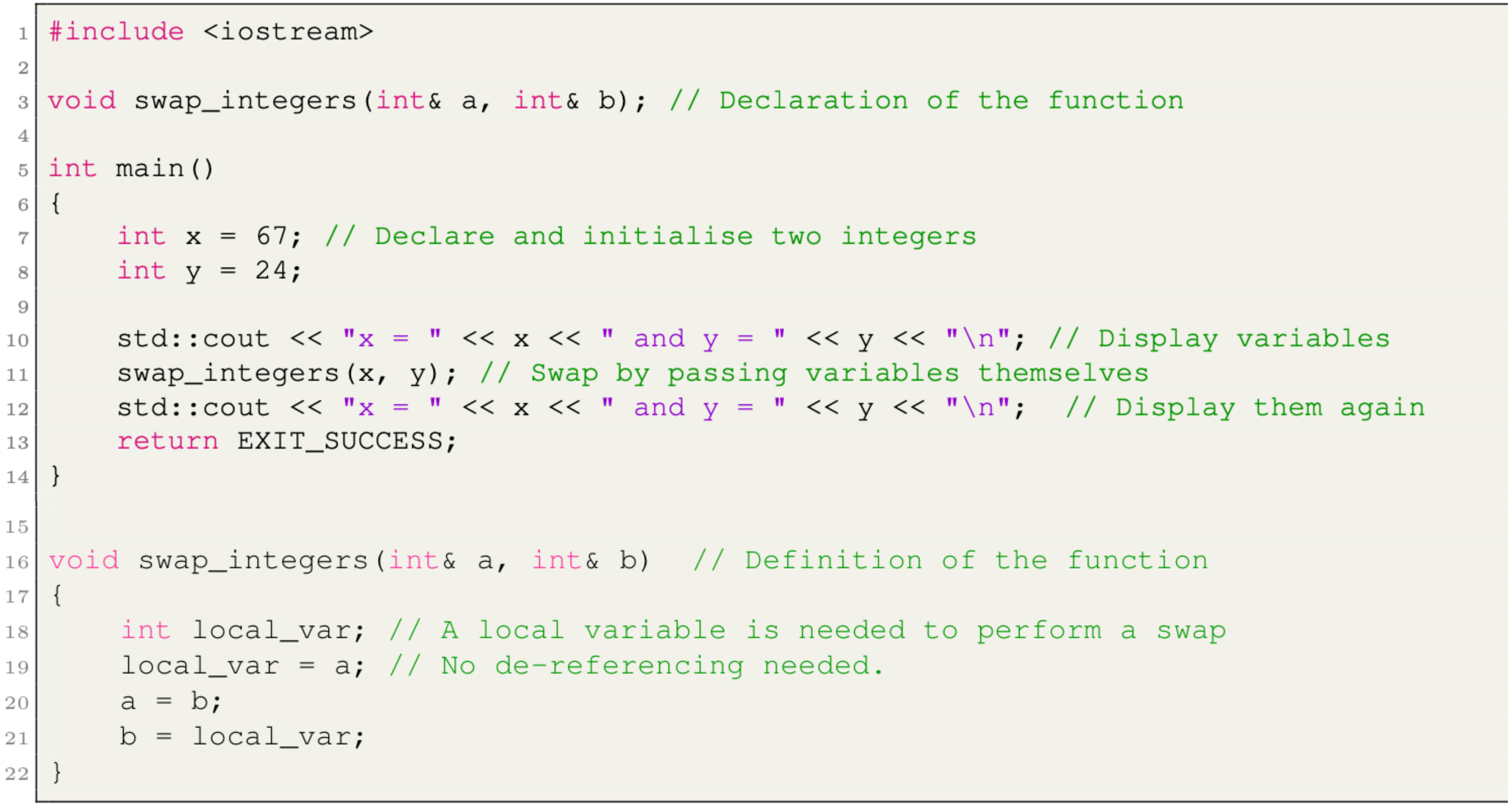
Explicit pass-by-reference

Can tell the compiler to make a variable pass by reference …

Declare

Call function

Define



```cpp
#include <iostream>

void swap_integers(int& a, int& b); // Declaration of the function

int main()
{
    int x = 67; // Declare and initialise two integers
    int y = 24;

    std::cout << "x = " << x << " and y = " << y << "\n"; // Display variables
    swap_integers(x, y); // Swap by passing variables themselves
    std::cout << "x = " << x << " and y = " << y << "\n";  // Display them again
    return EXIT_SUCCESS;
}

void swap_integers(int& a, int& b)  // Definition of the function
{
    int local_var; // A local variable is needed to perform a swap
    local_var = a; // No de-referencing needed.
    a = b;
    b = local_var;
}
```

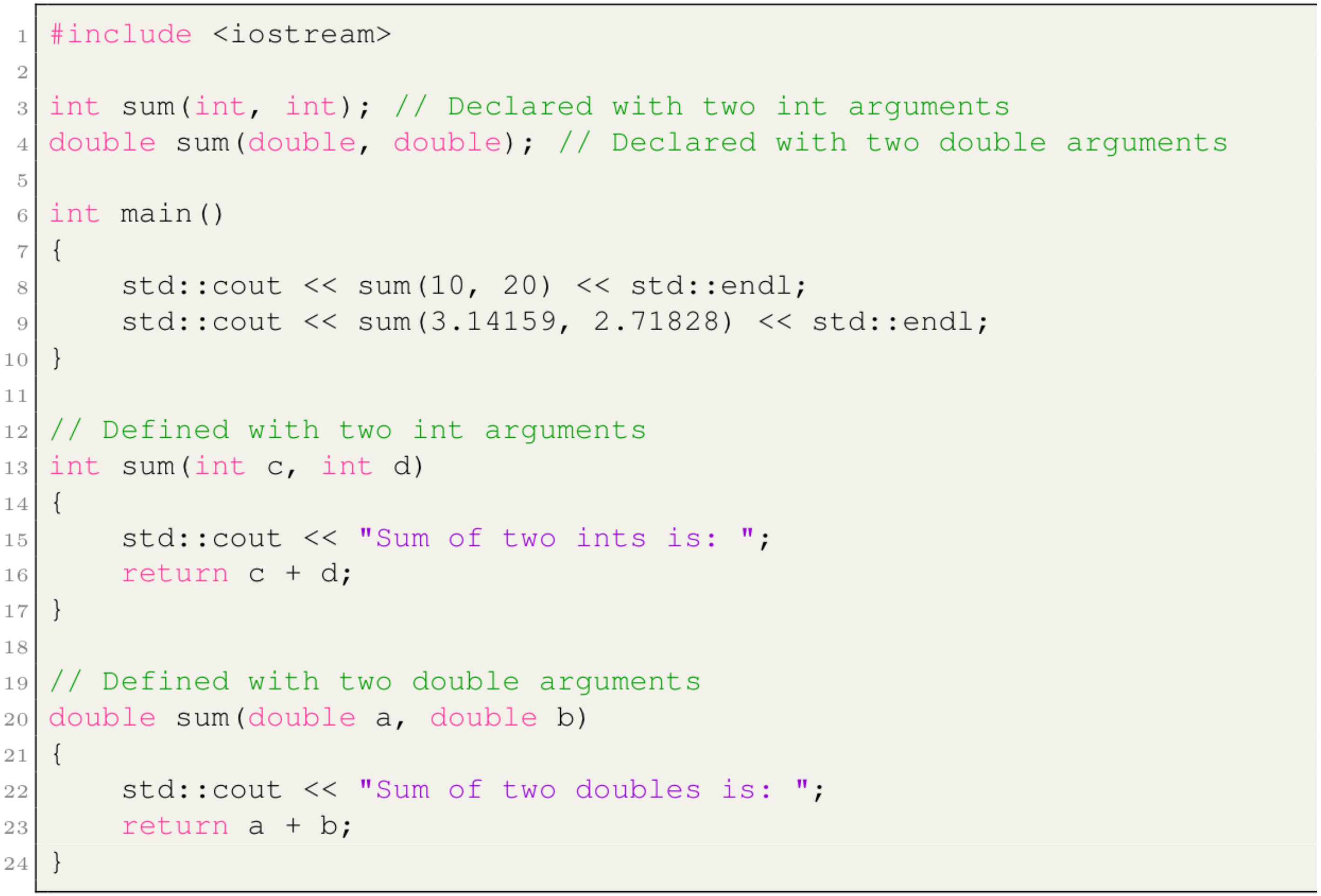
Function overloading

Declare two functions with the same name but different arguments … *overloading*

Call functions

Define `int` version

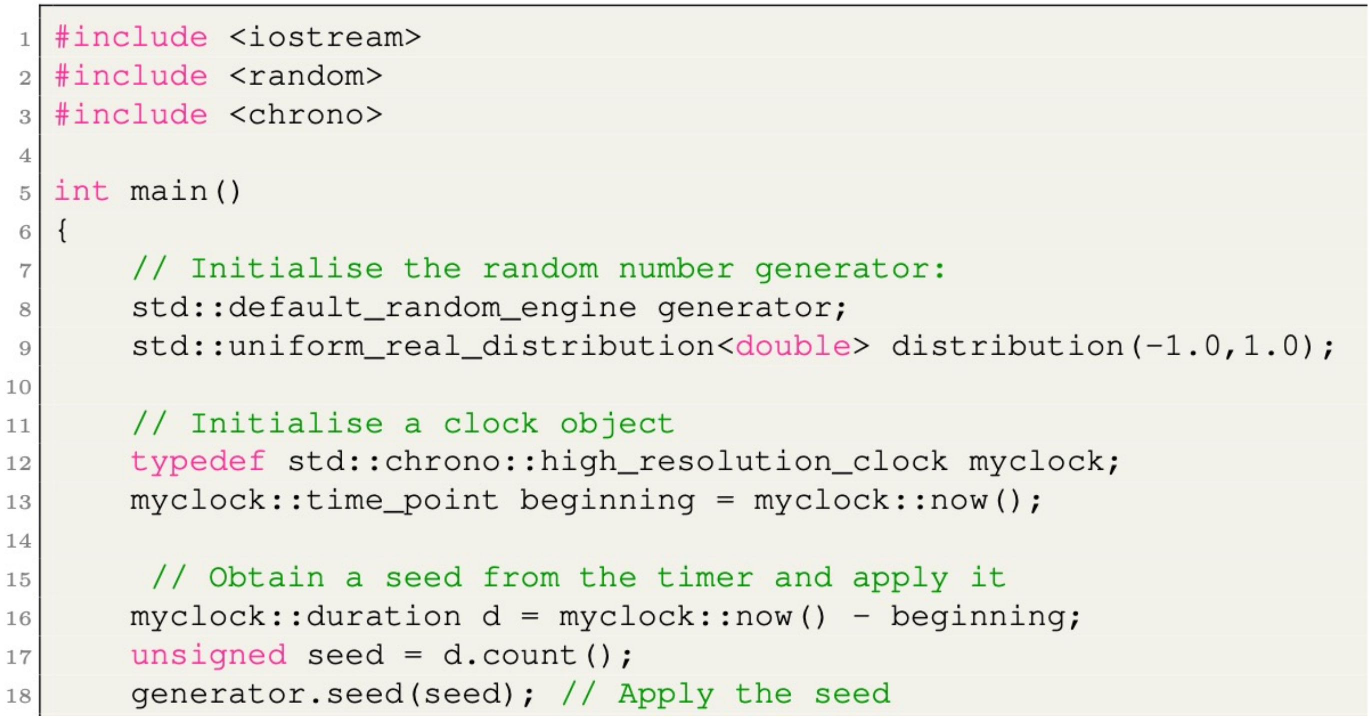
Define `double` version



```cpp
#include <iostream>

int sum(int, int); // Declared with two int arguments
double sum(double, double); // Declared with two double arguments

int main()
{
    std::cout << sum(10, 20) << std::endl;
    std::cout << sum(3.14159, 2.71828) << std::endl;
}

// Defined with two int arguments
int sum(int c, int d)
{
    std::cout << "Sum of two ints is: ";
    return c + d;
}

// Defined with two double arguments
double sum(double a, double b)
{
    std::cout << "Sum of two doubles is: ";
    return a + b;
}
```

Random number generation

Many numerical methods require a source of pseudo-random numbers. In C++ we use the following …



```
#include <iostream>
#include <random>
#include <chrono>

int main()
{
    // Initialise the random number generator:
    std::default_random_engine generator;
    std::uniform_real_distribution<double> distribution(-1.0,1.0);

    // Initialise a clock object
    typedef std::chrono::high_resolution_clock myclock;
    myclock::time_point beginning = myclock::now();

     // Obtain a seed from the timer and apply it
    myclock::duration d = myclock::now() - beginning;
    unsigned seed = d.count();
    generator.seed(seed); // Apply the seed
```

include libraries

declare generator and distribution objects

declare clock object and initialise timer

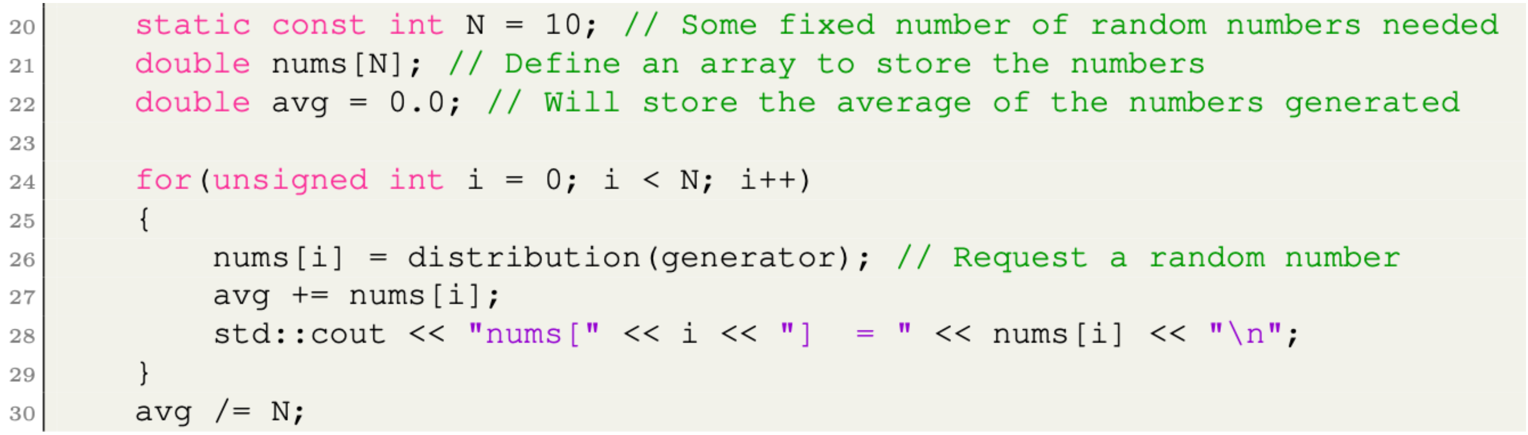
measure duration to here

use the number of “ticks” as a seed for generator

Random number generation

With this setup we then simply request numbers from the distribution:

generate a random number



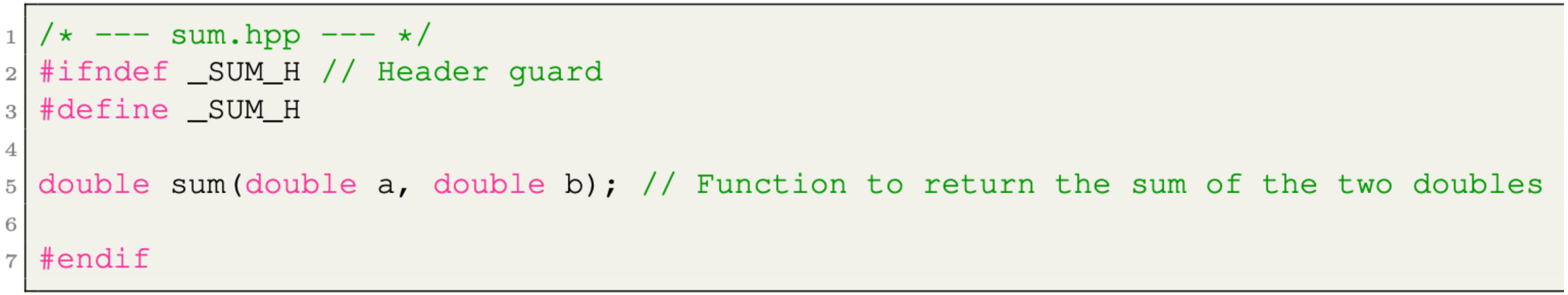
```
    static const int N = 10; // Some fixed number of random numbers needed
    double nums[N]; // Define an array to store the numbers
    double avg = 0.0; // Will store the average of the numbers generated

    for(unsigned int i = 0; i < N; i++)
    {
        nums[i] = distribution(generator); // Request a random number
        avg += nums[i];
        std::cout << "nums[" << i << "] = " << nums[i] << "\n";
    }
    avg /= N;
```

In Exercise 1 you will test this pseudo-random number generator.

Multi-file compilation

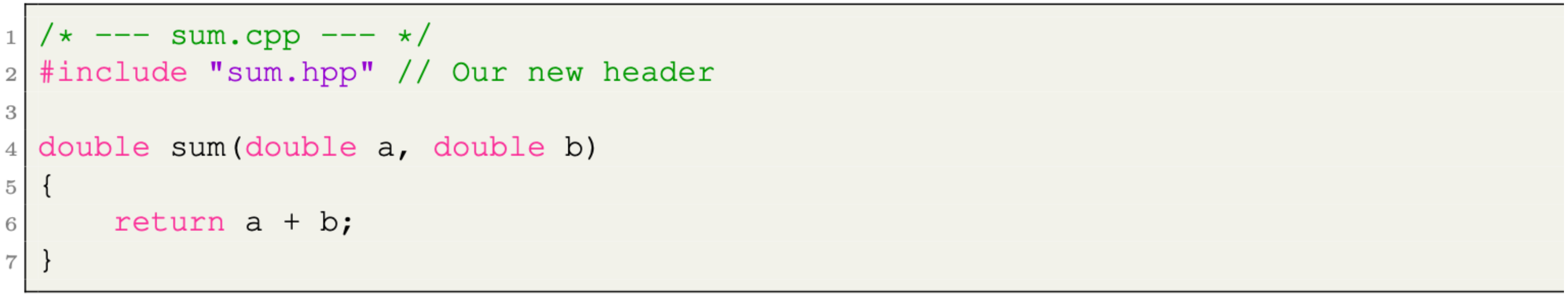
For larger projects it is useful to separate out function declarations, function implementations and the `main()` program into different files:

Header file for `sum.cpp` with declaration



```cpp
/* --- sum.hpp --- */
#ifndef _SUM_H // Header guard
#define _SUM_H

double sum(double a, double b); // Function to return the sum of the two doubles

#endif
```

Implementation



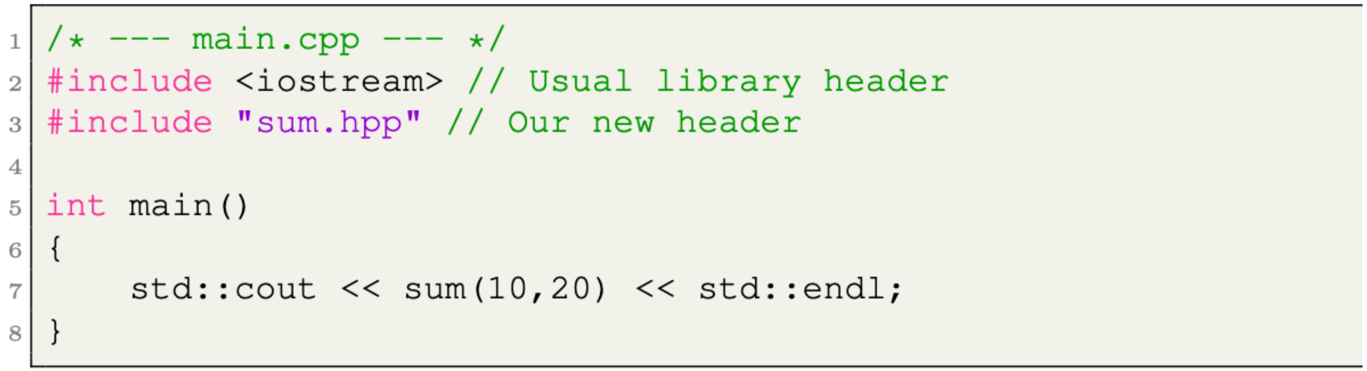
```cpp
/* --- sum.cpp --- */
#include "sum.hpp" // Our new header

double sum(double a, double b)
{
    return a + b;
}
```

Multi-file compilation

We then include the header file in our `main()` program:

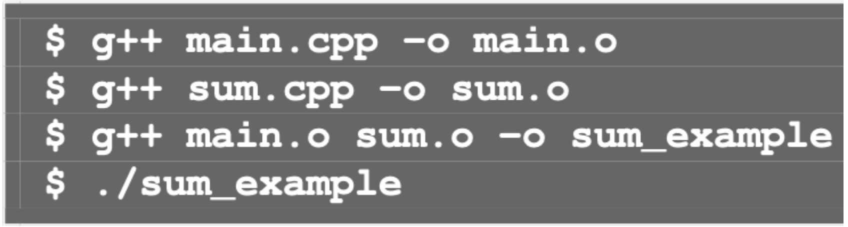
Preprocessor will insert contents of `sum.hpp` here

Call the function

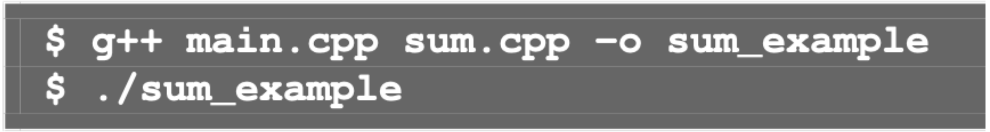


```cpp
/* --- main.cpp --- */
#include <iostream> // Usual library header
#include "sum.hpp" // Our new header

int main()
{
    std::cout << sum(10,20) << std::endl;
}
```

The header declares the function, but we still need to give the compiler the implementation. We can compile as:



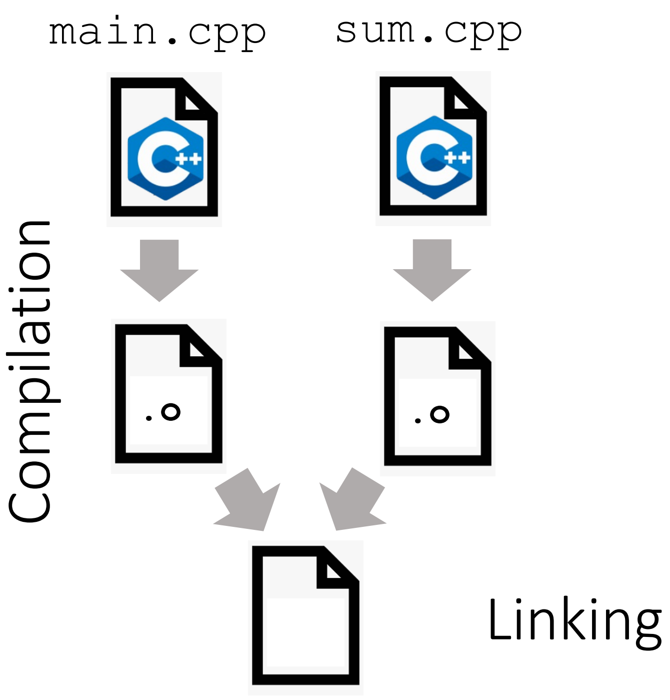
```
$ g++ main.cpp -o main.o
$ g++ sum.cpp -o sum.o
$ g++ main.o sum.o -o sum_example
$ ./sum_example
```

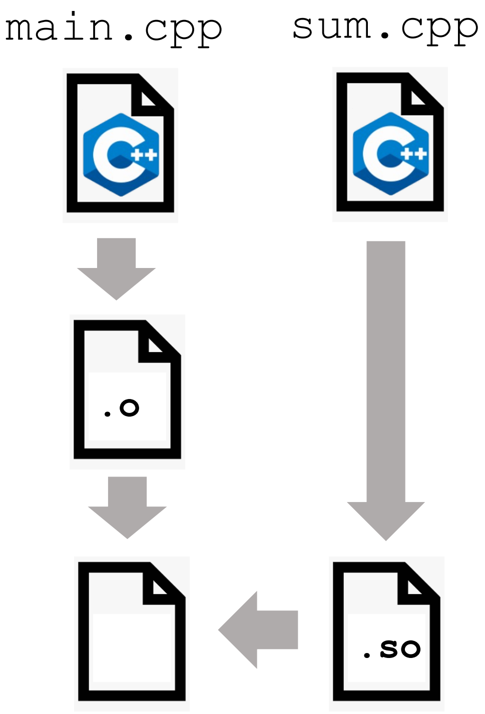


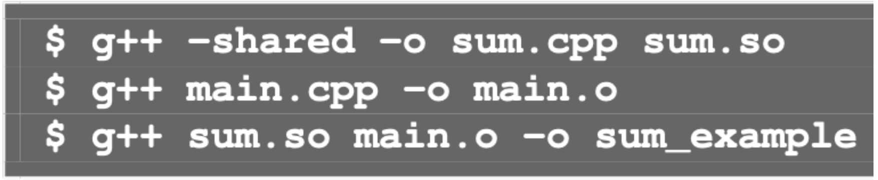
```
$ g++ main.cpp sum.cpp -o sum_example
$ ./sum_example
```

Static and dynamic linking

Above is *static linking*: We can also use *dynamic linking*:







```
$ g++ -shared -o sum.cpp sum.so
$ g++ main.cpp -o main.o
$ g++ sum.so main.o -o sum_example
```

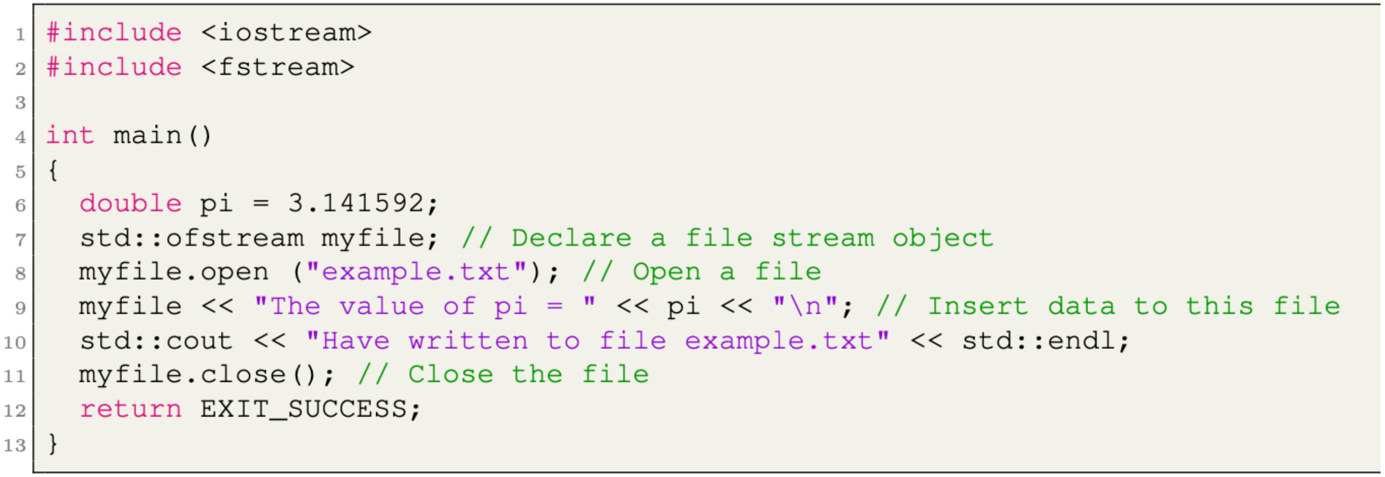
Produces smaller executables and an easily distributable “library” + header file that others could use in their code.

Writing to a file

Writing to files is analogous to writing to the console but we need an IO stream pointing at a file instead …

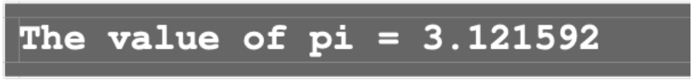
Include the file IO stream header

Write to file using the insertion << operation



```
#include <iostream>
#include <fstream>

int main()
{
  double pi = 3.141592;
  std::ofstream myfile; // Declare a file stream object
  myfile.open ("example.txt"); // Open a file
  myfile << "The value of pi = " << pi << "\n"; // Insert data to this file
  std::cout << "Have written to file example.txt" << std::endl;
  myfile.close(); // Close the file
  return EXIT_SUCCESS;
}
```

Creates `example.txt` file containing



```
The value of pi = 3.121592
```

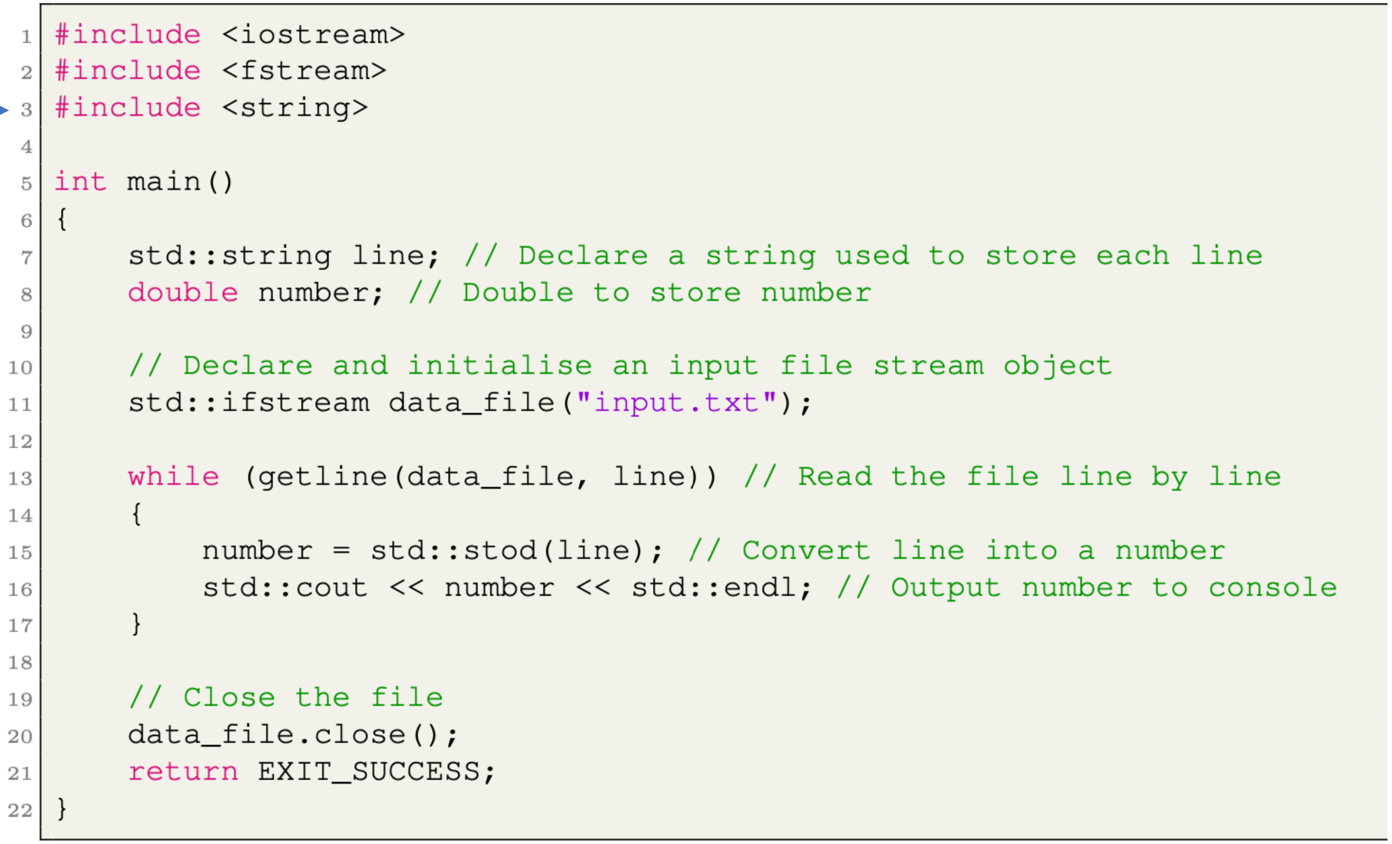
Reading a file

Reading files needs code that can parse strings. Here we have a simple example:

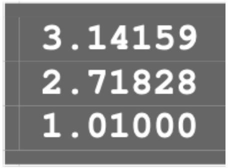
Include string library

Get line of the file

Code expects one number per line so converts entire line



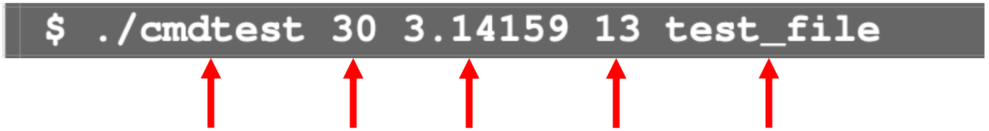
```cpp
#include <iostream>
#include <fstream>
#include <string>

int main()
{
    std::string line; // Declare a string used to store each line
    double number; // Double to store number

    // Declare and initialise an input file stream object
    std::ifstream data_file("input.txt");

    while (getline(data_file, line)) // Read the file line by line
    {
        number = std::stod(line); // Convert line into a number
        std::cout << number << std::endl; // Output number to console
    }

    // Close the file
    data_file.close();
    return EXIT_SUCCESS;
}
```



```
3.14159
2.71828
1.01000
```

Command line arguments

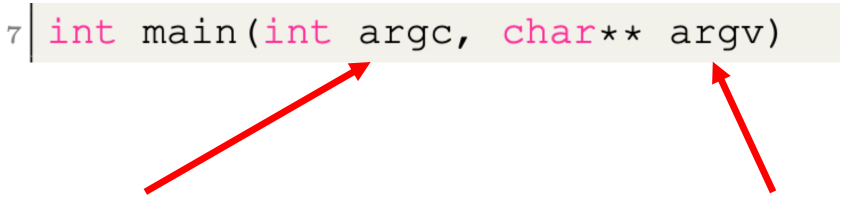
Suppose we have program `cmdtest` and run it from the command line as:



```
$ ./cmdtest 30 3.14159 13 test_file
```

command line arguments

The mandatory `main()` function has inputs which have a fixed format:



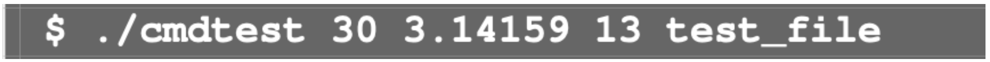
```
int main(int argc, char** argv)
```

number of command line arguments
(above = 5)

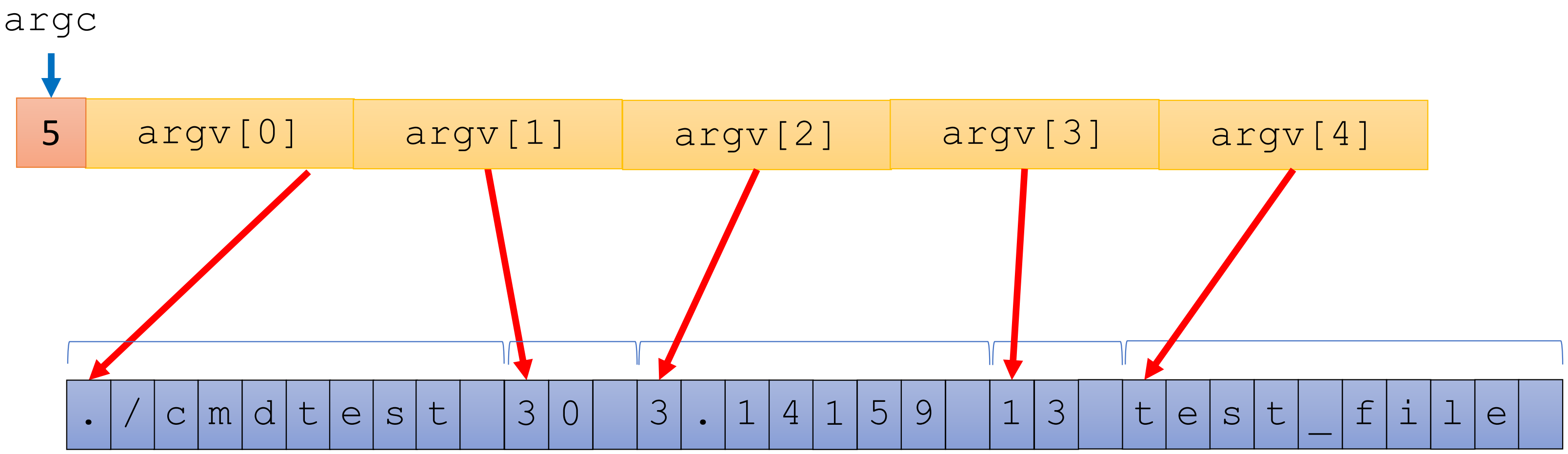
array of character arrays (string) containing the command line arguments

Command line arguments

For this example, the following would be passed …



```
$ ./cmdtest 30 3.14159 13 test_file
```

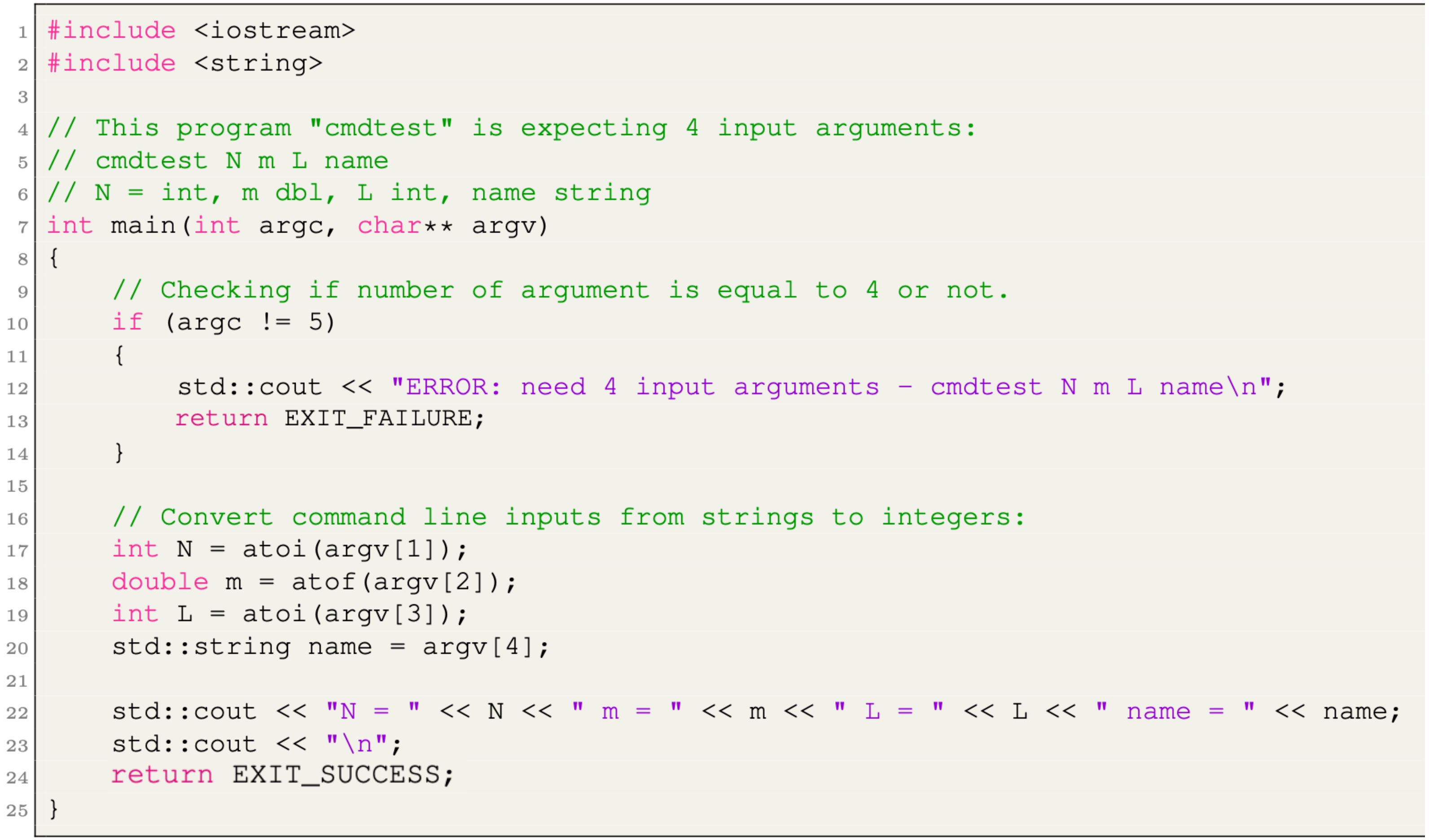


We are usually only interested in `argv[1]` onwards which pass information directly to our program.

Command line argument example:

check if enough argument have been passed

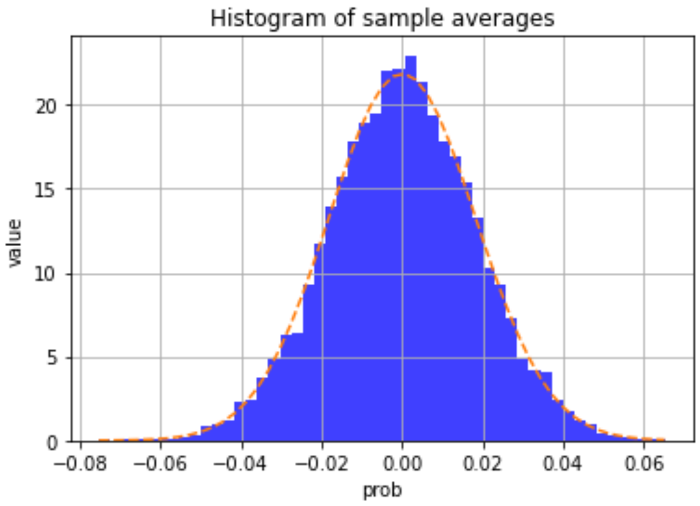
convert the character arrays into required types

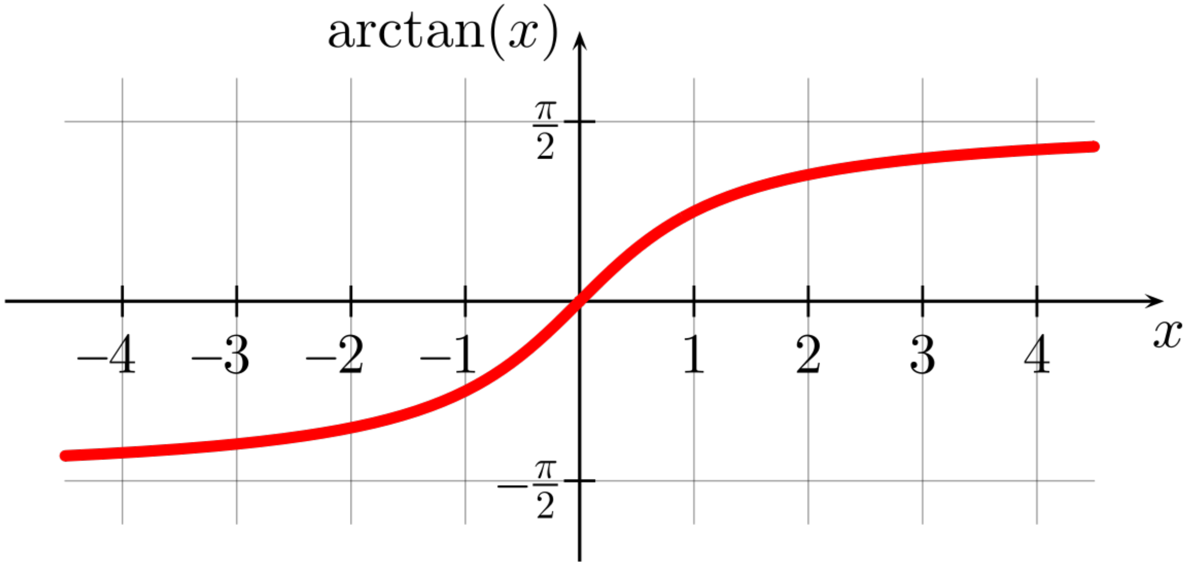


```
#include <iostream>
#include <string>

// This program "cmdtest" is expecting 4 input arguments:
// cmdtest N m L name
// N = int, m dbl, L int, name string
int main(int argc, char** argv)
{
    // Checking if number of argument is equal to 4 or not.
    if (argc != 5)
    {
        std::cout << "ERROR: need 4 input arguments - cmdtest N m L name\n";
        return EXIT_FAILURE;
    }

    // Convert command line inputs from strings to integers:
    int N = atoi(argv[1]);
    double m = atof(argv[2]);
    int L = atoi(argv[3]);
    std::string name = argv[4];

    std::cout << "N = " << N << " m = " << m << " L = " << L << " name = " << name;
    std::cout << "\n";
    return EXIT_SUCCESS;
}
```

Workshop exercises

The exercises this week give you some tasks using the methods introduced:


Histogram of sample averages
value
prob


$\arctan(x)$
$\frac{\pi}{2}$
$-\frac{\pi}{2}$
x