



University
of Glasgow

SUPA Graduate C++ Course

Lecture 1

Sarah Boutle
University of Glasgow

Many thanks to Adrian Buzatu, S. Allwood-Spiers

Introduction

- There are 4 lectures
- This is a practical subject, you cannot learn C++ by attending these 4 lectures
- Starts for complete beginners and gets far in 4 hours
- You need to put some work in yourselves:
 - 4 assignments, hand in the assignments to get the credits for this course
 - Try things out yourself, it's the only way to learn to code
 - google is your friend
- Labs:
 - Monday afternoons, 2-5pm starting 24th November
 - Room 220a, Kelvin Building, University of Glasgow
 - Can be used to get help for the assignments, but not compulsory
 - Please indicate if you intend to come in person

Today's Lecture: Basic Elements

- Introduction
 - Foreword
 - Programming Methodology
- Basic C/C++ Syntax
 - Simple types and operators
 - Conditional Statements
 - Loops
 - Functions
 - Header files
 - File input

Programming Methodology

Form a plan of the program needed before writing any C++.

- Use a flowchart or pseudo-code
- Think through the implementation

http://en.wikipedia.org/wiki/Flow_chart
<http://en.wikipedia.org/wiki/Pseudocode>

A little planning at the beginning can save a lot of time later on.

- This is especially true of Object Orientated languages

1. Requirements
2. Design
3. Implementation
4. Testing
5. Documentation

First C++ program example

STEP1: Using an editor, e.g. emacs, create a file `HelloWorld.cc` containing the following:

```
// My first C++ program

#include <iostream>

int main(){

std::cout << "Hello World!" << std::endl;
return 0;

}
```

1. A C++ program has a `main()` function from which the application will start running.
2. `main()` must return an `int`
3. The return statement in `main` returns control to the operating system
4. Statements end with a semicolon

First C++ program example

STEP 2: we need to compile the file (creates machine-readable code)

```
g++ -o Helloworld HelloWorld.cc
```

invokes
compiler

name of output
executable file

source
code

STEP 3: run the program:

```
./Helloworld
```

Result..... computer shows: Hello World!

Notes on Compiling

```
g++ -o HelloWorld HelloWorld.cc
```

is actually an abbreviated way of saying

```
g++ -c HelloWorld.cc
```

→ Compiler (-c) produces HelloWorld.o ("object files")

then

```
g++ -o HelloWorld HelloWorld.o
```

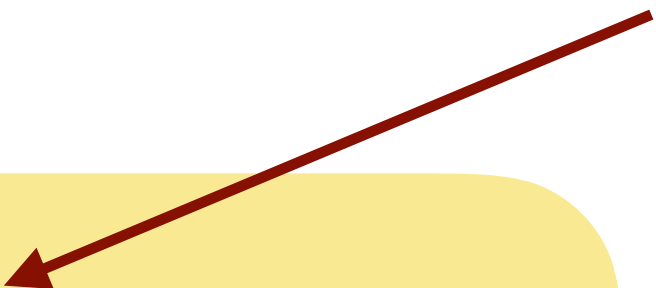
→ Links the object file(s)

if the program contains more than one source file:

```
g++ -o HelloWorld HelloWorld.cc HelloMars.cc \  
HelloUniverse.cc
```

Closer look at first program: comments

```
// My first C++ program
```



```
#include <iostream>
```

```
int main(){
```

```
std::cout << "Hello World!" << std::endl;  
return 0;
```

```
}
```

Comments: lines ignored
by the compiler

// inline comment

/* */ multiline comment

Closer look at first program: comments

```
/* S. Boutle  
** A very simple C++ program  
** to print one line  
*/  
  
#include <iostream>  
  
  
int main(){  
  
std::cout << "Hello World!" << std::endl;  
return 0;  
  
}
```

1. You should include comments in your code to make it understandable by others, and your future self!
2. Good practice: Start each file with comments indicating the authors name, purpose of the code, required input...

Closer look at first program: Include statements

```
/* S. Boutle  
** A very simple C++ program  
** to print one line  
*/  
  
#include <iostream>  
  
int main(){  
  
std::cout << "Hello World!" << std::endl;  
return 0;  
  
}
```

compiler directive

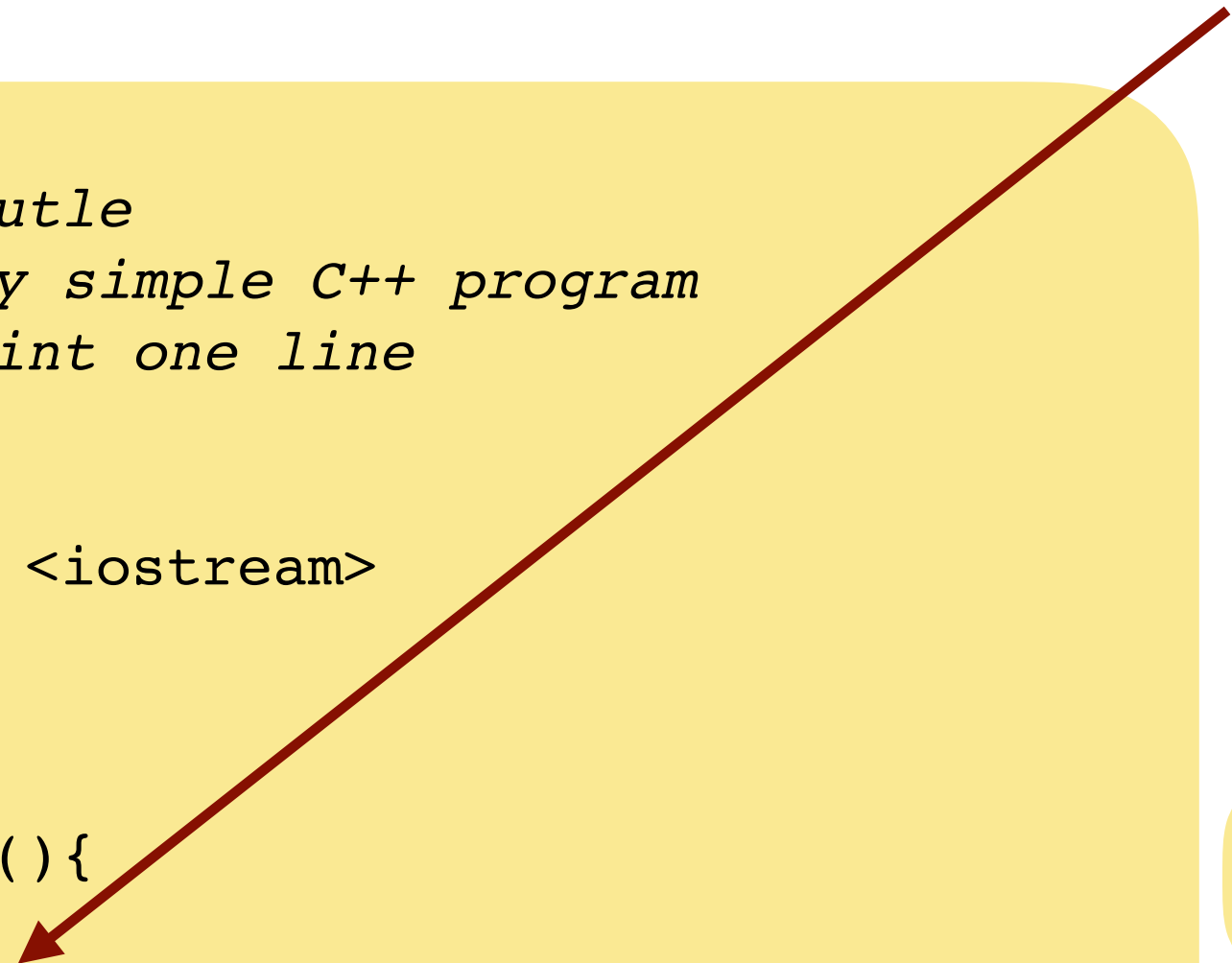
1. Compiler directive starts with a #
2. Not executed at run time but provide information to the compiler
3. `iostream` is a file containing definitions of library routines which the code will use
4. `iostream` contains functions that perform i/o operations to communicate with keyboard and monitor, e.g `cout`

Closer look at first program

```
/* S. Boutle
** A very simple C++ program
** to print one line
*/

#include <iostream>

int main(){
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```



standard output stream:
`cout` in namespace `std`

1. Send strings (set of characters) to `cout` with a `<<` shift operator
2. Automatic conversion to string for numerical quantities

```
cout << "x = " << x << endl;
```

3. Sending `endl` to `cout` indicates a new line, (try it without)

Closer look at first program

```
/* S. Boutle
** A very simple C++ program
** to print one line
**/

#include <iostream>

using namespace std;

int main(){

cout << "Hello World!" << endl;
return 0;

}
```

standard output stream:
`cout` in namespace `std`



C++ building blocks

All of the words in a C++ program are either:

Reserved words: cannot be changed, e.g.,

`if, else, int, double, for, while, class, ...`

Library identifiers: default meanings usually not changed,

e.g., `cout, sqrt` (square root), ...

Programmer-supplied identifiers:

e.g. variables created by the programmer,

`x, y, probeTemperature, photonEnergy, ...`

Variable declaration and assignment

Declaring a variable reserves some memory for it and establishes its name:

```
int i;
```

→ declares a variable of type `int` with identifier (name) `i`

Naming tips:

1. Variable names must start with a letter or a _
2. C++ is case sensitive
3. Try to use meaningful variable names
4. Some keywords are reserved: <http://en.cppreference.com/w/cpp/keyword>
5. Suggest lowerCamelCase

Variable declaration and assignment

All variables must be declared before use, usually just before 1st use

Then initialize the variable by assigning it a value

```
int i; //declaration  
i=3; //assignment
```

Can be done in one step

```
int j = 5; //declaration and initialisation
```



assignment operator

Data types

Data values can be stored in variables of several types

C++ has some predefined types:

Basic integer:

`int` (also `short`, `unsigned`, `long int`)

Basic floating point types: i.e. for real numbers,

`float` (32 bits), `double` (64 bits), ...

Boolean:

`bool` (equal to `true` or `false`)

Character:

`char` (single ASCII character only, can be blank), no native string

Some assignment examples

```
int main(){

    int numPhotons;           // Use int to count things
    double photonEnergy;      // Use double for real numbers
    bool goodEvent = true;    // Use bool for true or false
                                true, false are predefined constants

    double x, y, z;           // More than one on line
    x = 3.7;                   // Can initialize value
    y = 5.2;                   // when variable declared.
    z = x + y;                 // Value of char in ' '

    cout << "z = " << z << endl;

    z = z + 2.8;               // N.B. not like usual equation
    cout << "now z = " << z << endl;

}
```

Constant Variables

Ensure the value of a variable doesn't change

e.g. useful for keeping parameters of a problem in an easy to define place, where they are easier to modify

Use keyword `const` in declaration:

```
const int numChannels = 12;  
const double PI = 3.14159265;  
  
PI = 3.2; // ERROR will not compile
```

Mathematical operators/expressions

operation	symbol
addition	+
subtraction	-
multiplication	*
division	/
modulus	%

1. C++ has obvious notation for mathematical expressions
2. Other basic mathematical functions can be found in `<cmath>` and `<math.h>`
3. * and / have precedence over + and -, * and / have same precedence, carry out left to right. **If in doubt use parentheses!**

← division of int is truncated

```
int n,m;  
n = 5; m = 3;  
int ratio = n/m // ratio has value 1
```

modulus gives remainder of integer division

```
int nModM = n%m // nModM has value 2
```

Standard Maths Functions

Simple mathematical functions are available through the standard C library `cmath`, including:

`abs acos asin atan atan2 cos cosh exp fabs fmod
log log10 pow sin sinh sqrt tan tanh`

Most of these can be used with `float` or `double` arguments;

- return value is then of same type.

Raising to a power, $z = x^y$, with `z = pow(x, y)` involves log and exponentiation operations; not very efficient for `z = 2, 3`, etc.

Some advocate e.g. `double xSquared = x*x;`

To use these functions we need: `#include <cmath>`

Google for C++ `cmath` or see www.cplusplus.com for more info

Boolean Operators/Expressions

- Logic operators are used for comparing 2 expressions or variables
- Result is a boolean
- Can be combined with:

`&&` (and)

`||` (or)

`!` (not)

```
int n, m; n = 5; m = 3;
bool b = n < m;           // false
```

```
bool b1 = (n < m) && (n != 0)
```

```
bool b2 = (n%m >= 5) || !(n == m)
```

operator	meaning
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

not "=" (assignment operator)

true or false?

Shorthand assignment

full statement	shorthand equivalent
<code>n = n + m</code>	<code>n += m</code>
<code>n = n - m</code>	<code>n -= m</code>
<code>n = n * m</code>	<code>n *= m</code>
<code>n = n / m</code>	<code>n /= m</code>
<code>n = n % m</code>	<code>n %= m</code>
<code>n = n + 1</code>	<code>n++</code>
<code>n = n - 1</code>	<code>n--</code>
<code>n = n + 1</code>	<code>++n</code>
<code>n = n - 1</code>	<code>--n</code>

`j=i++;` means
`j=i` then `i=i+1`
`j=++i;` means
`i=i+1` then `j=i`

Example

```
#include <iostream>

using namespace std;

int main() {

    int myNum = 3;
    int j = myNum++;

    cout << "myNum = " << myNum << ", j= " << j << endl;

    int myNumCubed = myNum*myNum*myNum;
    cout << "The cube of " << myNum << " is " << myNumCubed << endl;

    return 0;

}
```

`int j = myNum++;` `myNum = 4 , j= 3`
cube of 4 is 64

`int j = ++myNum;` `myNum = 4 , j= 4`
cube of 4 is 64

Conditions

Simple flow control is done with `if` and `else`:

```
if ( boolean test expression ){  
    Statements executed if test expression true  
}
```

or

```
if (expression1 ){  
    Statements executed if expression1 true  
}  
else if ( expression2 ) {  
    Statements executed if expression1 false and expression2 true  
}  
else {  
    Statements executed if both expression1 and expression2 are false  
}
```

Note the indentation and the placement of curly brackets

For a single statement, no need for curly brackets but be careful...

```
if (expression1 ) Statements executed if expression1 true
```


Aside: `bool` vs `int`

C and earlier versions of C++ did not have the type `bool`. Instead, an `int` value of zero was interpreted as false, and any other value as true. This still works in C++:

```
int num = 1;
if(num) {
    .....
```

It is best to avoid this.

- if you want to check true or false, use a `bool`
- if you want to check if a number is zero or not, then use the corresponding boolean expression:

```
int num = 1;
if(num != 0) {
    .....
```

Loops

A `while` loop allows a set of statements to be repeated as long as a particular condition is true

```
while( boolean expression ){  
    // statements to be executed as long as  
    // boolean expression is true  
}
```

A `do-while` loop is similar but always executes at least once

```
do {  
    // statements to be executed first time  
    // through loop and then as long as  
    // boolean expression is true  
} while ( boolean expression )
```

A `for` loop is a set of statements to be repeated a fixed number of times

```
for ( initialization action ;  
    boolean expression ; update action ){  
    // statements to be executed  
}
```

Loops

For a `while` loop to be useful, the boolean expression must be updated each pass through the loop

```
while (x < xMax) {  
    x += y;  
    ...  
}
```

A `do-while` loop can be useful if first pass is needed to initialize boolean expression

```
int n; bool gotValidInput = false;  
do {  
    cout << "Enter a positive int" << endl;  
    cin >> n;  
    gotValidInput = n > 0;  
} while ( !gotValidInput );
```

A `for` loop note that `i` will only be defined inside the `{}`

```
int sum = 0;  
for (int i = 1; i<=n; i++){  
    sum += i;  
}
```

Loops: more control

continue causes a single iteration of loop to be skipped
(jumps back to start of loop).

break causes exit from entire loop
(only innermost one if inside nested loops).

```
while ( processEvent ) {  
  
    if ( eventSize > maxSize ) { continue; }  
  
    if ( numEventsDone > maxEventsDone ) {  
        break;  
    }  
    // rest of statements in loop ...  
}
```

Usually best to avoid these by use of **if** statements

Scope

The **scope** of a variable is that region of the program in which it can be used.

If a block of code is enclosed in braces { }, then this delimits the scope for variables declared inside the braces.

This includes braces used for loops and if structures:

```
int x = 5;
for (int i=0; i<n; i++){
    int y = i + 3;
    x = x + y;
}
cout << "x = " << x << endl; // OK
cout << "y = " << y << endl; // BUG -- y out of scope
cout << "i = " << i << endl; // BUG -- i out of scope
```

Variables declared outside any function, including main, have 'global scope'. They can be used anywhere in the program.

Functions

Until now we have seen the main function as well as mathematical functions. We, the user, can also define functions

return type

function name

type of parameter to be passed to it

```
int cubeNumber(int); // prototype
```

```
int main() {  
    int myNum = 3;  
    ...  
    int myNumCubed = cubeNumber(myNum);  
    cout << "The cube of " << myNum << " is " << myNumCubed << endl;  
    return 0;  
}
```

```
int cubeNumber(int i) { // definition  
    return (i*i*i);  
}
```

note that the scope of i is local to the function

Function Return types

The prototype must also indicate the return type of the function:

`int, float, double, char, bool....`

```
int cubeNumber(int);
```

The function definition must return a value of this type:

```
int cubeNumber(int i) { // definition
    return (i*i*i);
}
```

When calling a function, it must be used in the same manner as an expression of the corresponding type:

```
int myNumCubed = cubeNumber(myNum);
```

Function Return types: `void`

The return type may be `void` in which case no return statement:

```
void showProduct(double a, double b){  
    cout << "a*b = " << a*b << endl;  
}
```

To call a function with return type `void`, we simply write its name with any arguments followed by a semicolon:

```
showProduct(3, 7);
```


Functions

- Now we can ‘call’ `cubeNumber` whenever we need the area of an ellipse; this is modular programming.
- The user doesn’t need to know about the internal workings of the function, only that it returns the right result.
- ‘Procedural abstraction’ means that the implementation details of a function are hidden in its definition, and needn’t concern the user of the function.
- A well written function can be re-used in other parts of the program and in other programs.
- Functions allow large programs to be developed by teams (as is true for classes, which we will see soon).

Putting functions in separate files

Often we put functions in a separate files. The declaration of a function goes in a 'header file' called, e.g., `cubeNumber.h`, which contains the prototype:

```
#ifndef CUBE_NUMBER_H
#define CUBE_NUMBER_H

// function to compute cube of a number

int cubeNumber(int);

#endif
```

The directives `#ifndef` (if not defined), etc., serve to ensure that the prototype is not included multiple times. If `ELLIPSE_AREA_H` is already defined, the declaration is skipped.

Putting functions in separate files

Then to use the function, you need to include the header file in all files where the function is called

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

int main() {
    int myNum = 3;
    ...
    int myNumCubed = cubeNumber(myNum);
    cout << "The cube of " << myNum << " is " << myNumCubed << endl;
    return 0;
}
```

The directives `#ifndef` (if not defined), etc., serve to ensure that the prototype is not included multiple times. If `ELLIPSE_AREA_H` is already defined, the declaration is skipped.

Functions: passing arguments by value

Consider a function that tries to change the value of an argument:

```
void tryToChangeArg(int x) {  
    x = 2*x;  
}
```

It won't work:

```
int x = 1;  
tryToChangeArg(x);  
cout << "now x = " << x << endl; // x still = 1
```

This is because the argument is passed 'by value': only a copy of the value of `x` is passed to the function.

In general this is a **Good Thing**: don't want arguments of functions to have their values changed unexpectedly.

Functions: passing arguments by value

Sometimes, however, we want to return modified values of the arguments. But a function can only return a single value.

We can change the argument's value by passing it "by reference":

```
void tryToChangeArg(int&); // prototype

void tryToChangeArg(int& x){ // definition
    x = 2*x;
}

int main(){
    int x = 1;
    tryToChangeArg(x);
    cout << "now x = " << x << endl; // now x = 2
}
```

Argument passed by reference must be a variable,
e.g., `tryToChangeArg(7);` will not compile.

Functions: default arguments

Sometimes it is convenient to specify default arguments for functions in their declaration:

```
double line(double x, double slope=1, double offset=0);
```

The function is then defined as usual:

```
double line(double x, double slope, double offset){  
    return x*slope + offset;  
}
```

We can then call the function with or without the defaults:

```
y = line (x, 3.7, 5.2); // here slope=3.7, offset=5.2  
y = line (x, 3.7);      // uses offset=0;  
y = line (x);           // uses slope=1, offset=0
```

Functions: Overloading

We can define versions of a function with different numbers or types of arguments (signatures). This is called function overloading:

```
double cubeNumber(double);  
double cubeNumber (double x){  
    return x*x*x;  
}  
  
double cubeNumber(float);  
double cubeNumber (float x){  
    double xd = static_cast<double>(x);  
    return xd*xd*xd;  
}
```

Return type can be same or different; argument list must differ in number of arguments or in their types.

Functions: Overloading

When we call the function, the compiler looks at the signature of the arguments passed and figures out which version to use:

```
float x;  
double y;  
double z = cubeNumber(x); // calls cubeNumber(float) version  
double z = cubeNumber(y); // calls cubeNumber(double) version
```

This is done e.g. in the standard math library `cmath`:

There is a version of `sqrt` that takes a `float` (and returns `float`), and another that takes a `double` (and returns `double`).

Note it is not sufficient if functions differ only by return type -- they must differ in their argument list to be overloaded.

Streams

A stream is an object that characters can be inserted to (e.g. `cout`) or extracted from (e.g. `cin`).

1. Streams provide a uniform basis for input and output independent of device
2. Streams allow access to i/o devices, e.g.:
 - files stored on a hard drive
 - the terminal or console
 - a printer
 - a database

Read/Write to File

Simple example that opens an existing file to read data from it:

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

using namespace std;

int main(){
    // create an ifstream object (name arbitrary)...

    ifstream myInput;
    // Now open an existing file...

    myInput.open("myDataFile.txt");
    // check that operation worked...
    if ( myInput.fail() ) {
        cout << "Sorry, couldn't open file" << endl;
        exit(1); // from cstdlib
    }
    ...
}
```

Read from input stream

Suppose the file contains columns of numbers like

1.0	7.38	0.43
2.0	8.59	0.52
3.0	9.01	0.55

We can read in these numbers from the file:

```
double x, y, z;
for(int i=1; i<=numLines; i++){
    myInput >> x >> y >> z;
    cout << "Read " << x << " " << y << " " << z << endl;
}
```

This loop requires that we know the number of lines in the file

Read to the end of the file:

Often we don't know the number of lines in a file ahead of time. We can use the “end of file” (`eof`) function:

```
double x, y, z;
int line = 0;

while ( !myInput.eof() ){
    myInput >> x >> y >> z;
    if ( !myInput.eof() ) {
        line++;
        cout << x << " " << y << " " << z << endl;
    }
}
cout << line << " lines read from file" << endl;
...
myInput.close(); // close when finished
```

Review

- Introduction
 - Foreword
 - Programming Methodology
- Basic C/C++ Syntax
 - Simple types and operators
 - Conditional Statements
 - Loops
 - Functions
 - Header files
 - File input

Assignment 1

1. Exercise: Read the file `input2D_float.txt` and for each line print the value from the first column (x value) and second column (y value). Assuming these are x and y components of a 2-D vector, compute the absolute value of the vector. Write to a new file called `output2D_float.txt` for each entry of the initial file three columns, the first two being the already had x and y, while the third one is the newly computed absolute value.

Notions you will learn: reading from a file, writing to a file, building a function called absolute value taking as inputs two values, each of the type float; how to use mathematical functions as square root and power two.

2. The same as in one, except use the input file `input3D_float.txt`, which has three values, the coordinates x, y, z.

Notions you will learn: Though you can build another function with another name, less you can and it is best practice to do so build another function with the same name (something like `absolute_value`), which takes this time three arguments. The fact that you are allowed to have functions with the same name while the compiler knows they are different because they have different input arguments is called overloading. Here we are doing function overloading.

Assignment 1

3. The same as in 1, except instead of using decimal numbers of type float, use integer numbers of type unsigned. Use input2D_int.txt and input3D_int.txt.

Notions you will learn: Though you all you need is to copy paste the code from above, and replace float with int, it is both tedious and a bad software practice. What happens if you found a bug in one function? Then you would have to modify in the other one, too. But maybe you forget to do so. You can and it is a good practice to do so, to build a function that can take as arguments either one type of variable (float), or the other (int). This is called templating. Here we are doing function templating.

4. Now we want to do the same thing for one of the input files, but instead of looping over all the four lines in the file, we want the user to give it from a command line argument the choice to use only a certain amount of lines, for example 2. If the user gives a number of lines smaller than 0, then the program should print an error text and then close. If the user gives a number larger than the number of lines, it should print a warning and not crash, but simply go through all the lines and end and just before the end print a warning.

Notions you learn from this: how to pass command line arguments.