C++ - Pre-lecture 10

More advanced topics

Caterina Doglioni, PHYS30762 - OOP in C++ 2023

Part 0.1: in this lecture

In this lecture

- **Part 1:** exploration of the standard library [video 1]
    - This is the main library you can/should be using in your project
- **Part 2:** exceptions [video 2]
- **Part 3:** lambda functions [video 3]
- Together with pre-lecture 9, those are all **advanced features** of C++
    - Practicing these in the project will leads to extra marks
    - Be explicit in your report why you're using them / what design problem they solve
- **Part 4:** closing words, overall recap [very short video 4]
- We will also see in class: C++ in research

Part 1 / Video 1:
exploration of the standard library

The standard library (std:: & others)

- C++ is supported by a rich library of functions and classes: the **C++ standard library**
  - Library == list of headers and implementations we can use
- We have already made use of the C++ standard library:
  - I/O: standard (keyboard and screen) <iostream>; file <fstream>; manipulations <iomanip>, data structures: strings <string> and variable length arrays <vector>
- Standard library routines are declared in the std:: namespace
- Can split library into 4 sections
  1. C standard library
  2. Standard Template Library 3
  3. I/O stream library
  4. Miscellaneous
- On top of this, we can also use the **boost** library (not in project
  - truly enormous, and very useful in many cases! (http://www.boost.org)

The standard library (`std::` & others)

1. **C standard library**
    - C++ is a superset of C, no need to rewrite useful functions (e.g. `cmath`, `string`)
2. **Standard Template Library (STL)**
    - Useful set of template classes, e.g.
        - Container classes (`vector`, `list`, `map`, `set`, …)
        - Algorithms (`copy`, `find`, `sort`, `reverse`…)
        - Iterators
        - Numeric (including `complex` numbers)
3. **I/O stream library**
    - Streams, including `ostream` and `fstream`
4. **Miscellaneous**
    - Includes `string` class and `utility` library

- We won't have time to cover them all - see : http://www.cplusplus.com/reference for full details
    - Instead, we will see some interesting examples

Complex numbers

- One of the assignments in 2023 was to write a class that could handle complex numbers…then we found out that ChatGPT would do that for you rather quickly
  - …but STL already has one! (still, it can be good to practice)
- It's a template class, specialised for float, double, long double
  `template <class T> class complex;`
  - can do all the usual operations - try them out:

```
complex<double> z1(1,1),z2(1,-1); // two new complex numbers
cout<<"z1 = "<<z1<<endl; // (1,1)
cout<<"z2 = "<<z2<<endl; // (1,-1)
cout<<"z1 + z2 = "<<z1+z2<<endl; // (2,0)
cout<<"z1 * z2 = "<<z2*z2<<endl; // (0,-2)
cout<<"Real part of z1 = "<<z1.real()<<endl; // 1
cout<<"Imag part of z2 = "<<z2.imag()<<endl; // -1
cout<<"Modulus of z1 = "<<abs(z1)<<endl; // 1.41421
cout<<"Argument of z1 = "<<arg(z1)<<endl; // 0.
```

Code on GitHub at: Prelecture9/staticdata.cpp

Pairs (std::pair)

- A useful class when have close association of two object types (e.g. double and string)
- Defined in utility library `#include<utility>`
- Example:

```
std::pair<double,string> obj1; // Declare pair object
obj1.first = 1.; // Define first part of pair
obj1.second = "Object1"; // and second
cout<<"First part: "<<obj1.first<<endl;
cout<<"Second part: "<<obj1.second<<endl;
```

- Output:

```
First part: 1
Second part: Object1
```

- Used for std::map class (see later slide)

MANCHESTER 1824 The University of Manchester

Container classes in STL

- Containers = a set of class templates within STL
- Designed to hold many objects within a single container. Three main types:

   1. **Sequence containers**: elements ordered in strict linear sequence: vector, list, deque (access time linear)

   2. **Associative containers**: elements accessed using a **key** (not position in sequence): set, multiset, map, multimap, bitset

   3. **Container adapters**: provide specific interface for some of the above (e.g., queue; not covered here)

  - Also note many more in latest C++ standard!

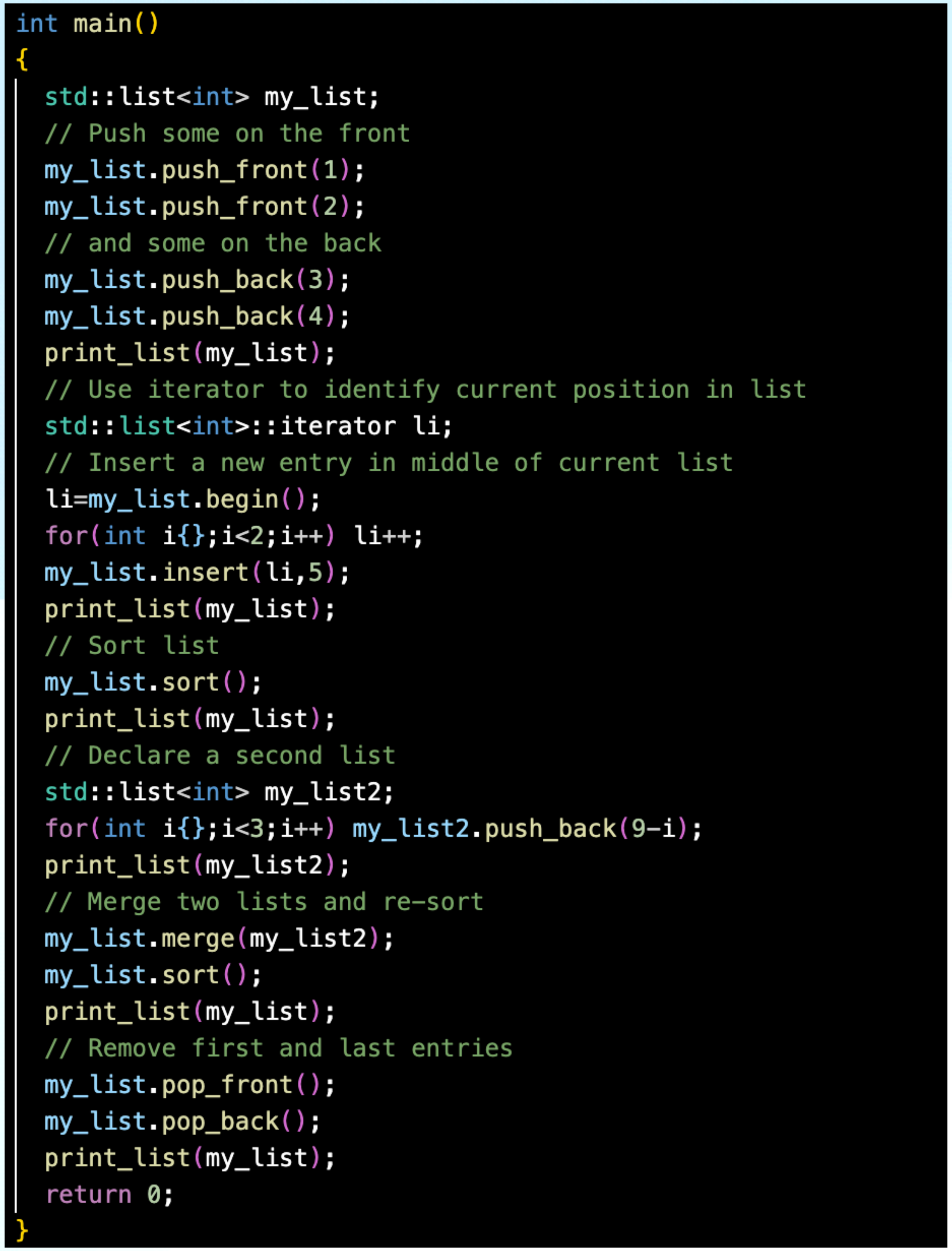
- Elements of container classes accessed using **iterators**
- Let us look at a few examples…

Sequence containers

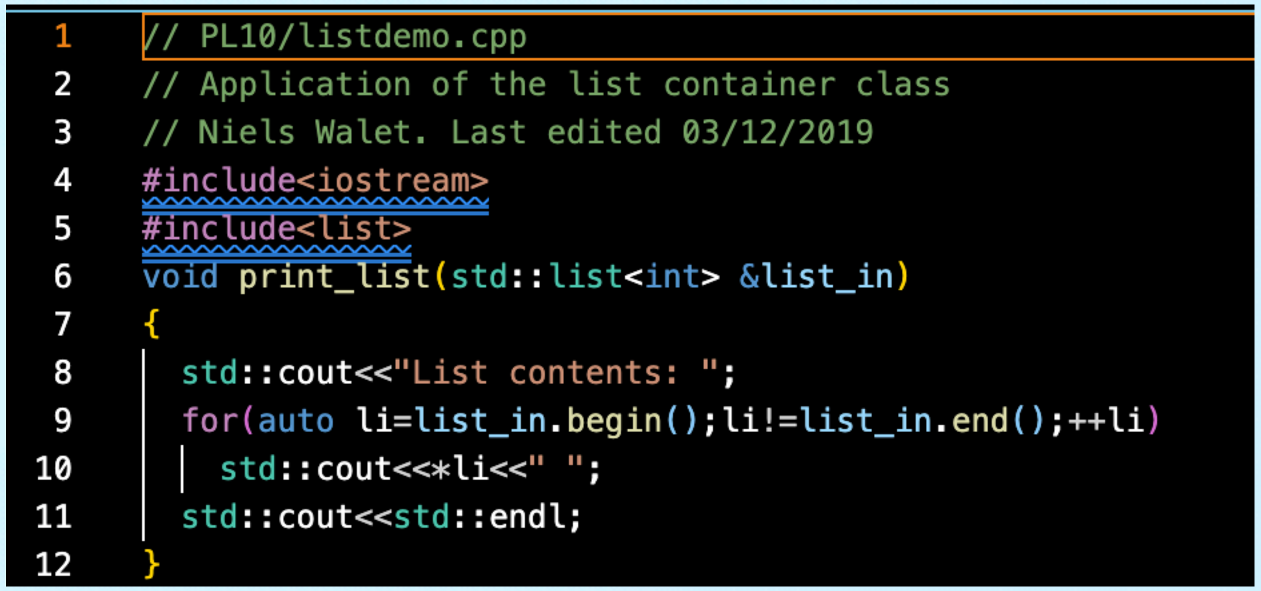
- Already met **`vector`** (a dynamic array)
  - Vectors allocate contiguous memory allowing random access ⇒strict linear relation between element and its memory address
- Another sequence container is the **`list`**, where each element instead uses two pointers (to previous and next element)
  - Faster than vectors for adding, subtracting and organising elements (e.g. sorting)
  - More awkward to find individual element (must use iterator or `search()` algorithm)

Code on GitHub at: Prelecture10/listdemo.cpp

Example of a list



```cpp
int main()
{
  std::list<int> my_list;
  // Push some on the front
  my_list.push_front(1);
  my_list.push_front(2);
  // and some on the back
  my_list.push_back(3);
  my_list.push_back(4);
  print_list(my_list);
  // Use iterator to identify current position in list
  std::list<int>::iterator li;
  // Insert a new entry in middle of current list
  li=my_list.begin();
  for(int i{};i<2;i++) li++;
  my_list.insert(li,5);
  print_list(my_list);
  // Sort list
  my_list.sort();
  print_list(my_list);
  // Declare a second list
  std::list<int> my_list2;
  for(int i{};i<3;i++) my_list2.push_back(9-i);
  print_list(my_list2);
  // Merge two lists and re-sort
  my_list.merge(my_list2);
  my_list.sort();
  print_list(my_list);
  // Remove first and last entries
  my_list.pop_front();
  my_list.pop_back();
  print_list(my_list);
  return 0;
}
```



```cpp
// PL10/listdemo.cpp
// Application of the list container class
// Niels Walet. Last edited 03/12/2019
#include<iostream>
#include<list>
void print_list(std::list<int> &list_in)
{
  std::cout<<"List contents: ";
  for(auto li=list_in.begin();li!=list_in.end();++li)
    std::cout<<*li<<" ";
  std::cout<<std::endl;
}
```

- **Output:**
  - Original list:
    List contents: 2 1 3 4
  - After insert: 2 1 5 3 4
  - After sorting: 1 2 3 4 5
  - The second list: 9 8 7
  - Merged and sorted lists: 1 2 3 4 5 6 7 8 9
  - Earlier list, deleting first and last elements : 2 3 4 5 6 7 8

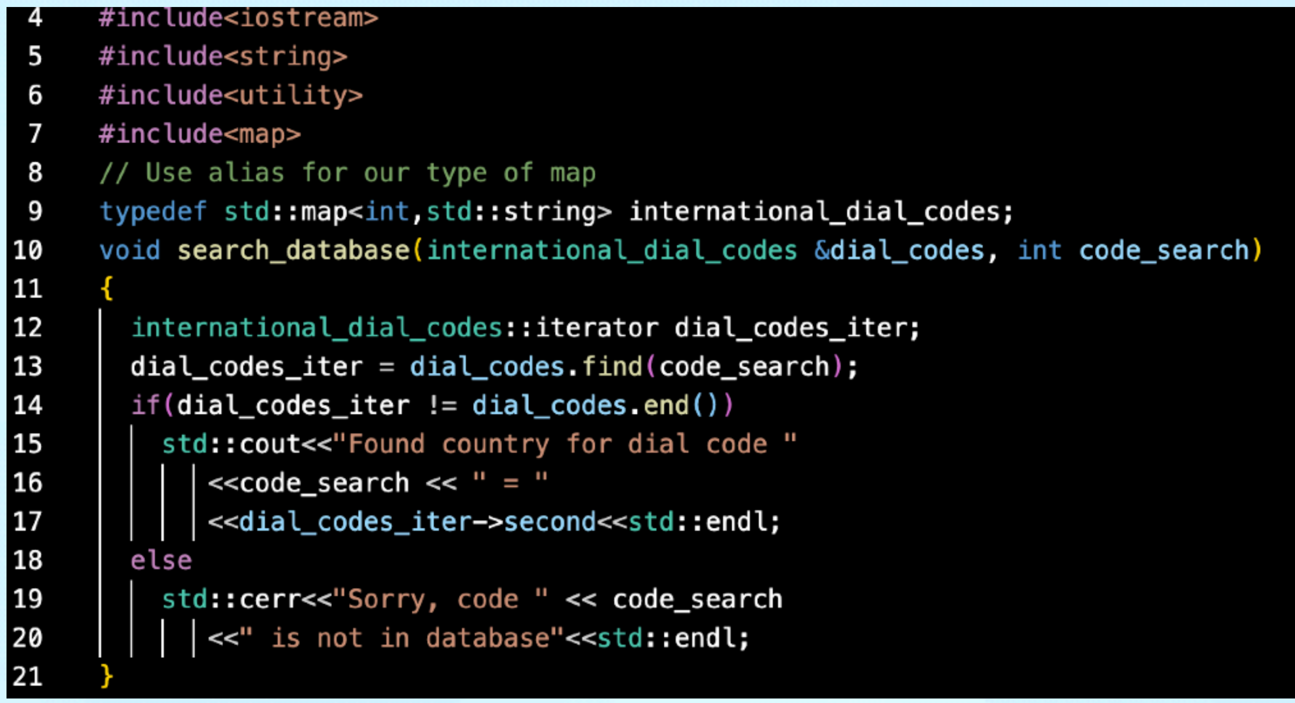
MANCHESTER 1824 The University of Manchester

Associative containers

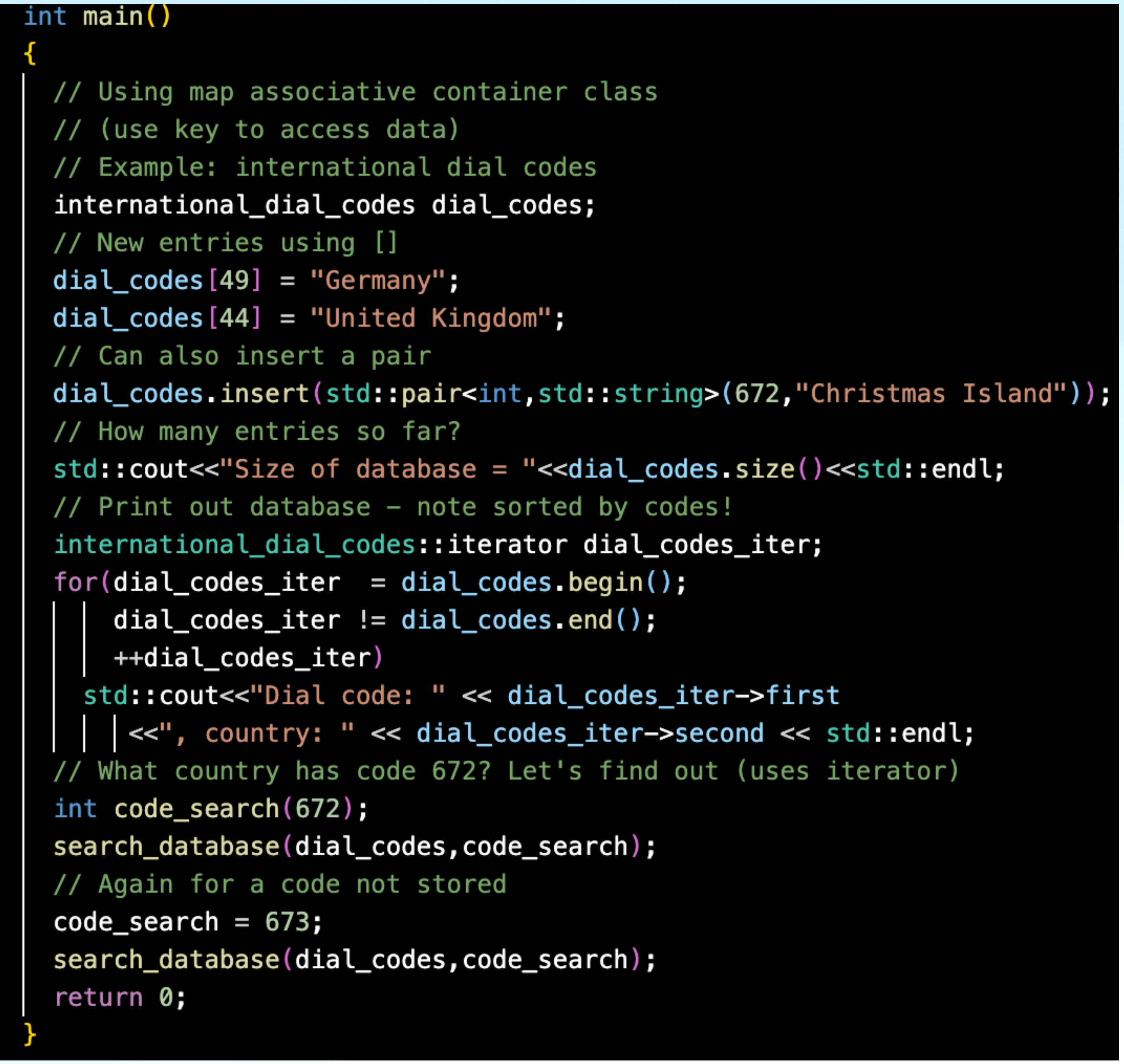
- Associative containers allow one to find entries by **association**
  - Example: the **`map`** class template - takes a pair of object types (*key* and *data*)
  - Overloads `operator[]` to use *key* instead of position in array (memory)
  - Useful method for accessing textual information stored by key
    - a map is essentially a simplified database
- Other example of associative container:
  - `multimap`: map where multiple keys are possible

Example of a map

Code on GitHub at: Prelecture10/mapdemo.cpp



```cpp
#include<iostream>
#include<string>
#include<utility>
#include<map>
// Use alias for our type of map
typedef std::map<int,std::string> international_dial_codes;
void search_database(international_dial_codes &dial_codes, int code_search)
{
  international_dial_codes::iterator dial_codes_iter;
  dial_codes_iter = dial_codes.find(code_search);
  if(dial_codes_iter != dial_codes.end())
    std::cout<<"Found country for dial code "
        <<code_search << " = "
        <<dial_codes_iter->second<<std::endl;
  else
    std::cerr<<"Sorry, code " << code_search
        <<" is not in database"<<std::endl;
}
```



```cpp
int main()
{
  // Using map associative container class
  // (use key to access data)
  // Example: international dial codes
  international_dial_codes dial_codes;
  // New entries using []
  dial_codes[49] = "Germany";
  dial_codes[44] = "United Kingdom";
  // Can also insert a pair
  dial_codes.insert(std::pair<int,std::string>(672,"Christmas Island"));
  // How many entries so far?
  std::cout<<"Size of database = "<<dial_codes.size()<<std::endl;
  // Print out database - note sorted by codes!
  international_dial_codes::iterator dial_codes_iter;
  for(dial_codes_iter  = dial_codes.begin();
      dial_codes_iter != dial_codes.end();
      ++dial_codes_iter)
    std::cout<<"Dial code: " << dial_codes_iter->first
        <<", country: " << dial_codes_iter->second << std::endl;
  // What country has code 672? Let's find out (uses iterator)
  int code_search(672);
  search_database(dial_codes,code_search);
  // Again for a code not stored
  code_search = 673;
  search_database(dial_codes,code_search);
  return 0;
}
```

- Note the **typedef** - saves some typing…

- Output:
```
Size of database = 3
Dial code: 44, country: United Kingdom
Dial code: 49, country: Germany
Dial code: 672, country: Christmas Island
Found country for dial code 672 = Christmas Island
```

- Note order of output - map sorts data (incremental order) based on key

- As such, the **< operator** must be already defined for key datatype (otherwise must define for the datatype you're using)

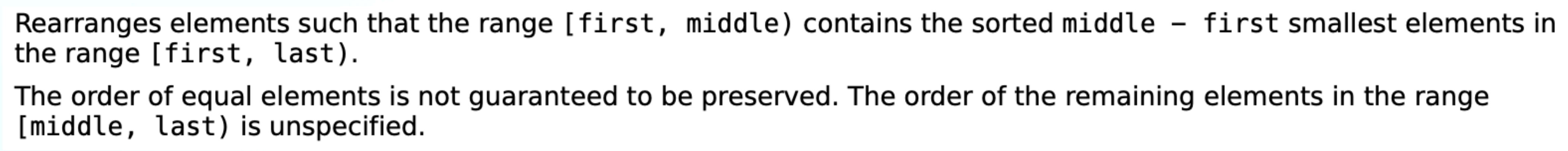
Algorithms

- Containers are often well used with the `<algorithm>` header
    - Most optimised way to perform operations on container
    - Examples: `find()` (already seen)
- Other examples that may be useful for your project:
    - **sort**
    - **partial_sort**
    - **find**
- There are many more… http://www.cplusplus.com/reference/algorithm/

MANCHESTER 1824 The University of Manchester

Why using STL algorithms?

- In general, if there is an algorithm in STL for the thing you want to do, that’s the most efficient way it can be written → good to use it!
    - This is a big advantage of open source code!
- How we use an efficient STL algorithm in a high energy physics example:
    - We need to sort the content of interesting collision events (in this case, *content = a vector of “jets” of particles*) by some property (in this case, *property = energy*)
    - We need to do this in the most efficient way possible, because the code needs to run up to 20000 times per second in the ATLAS real-time decision making system, and consume the least CPU resources
- **Example of `partial_sort`:** only keep & sort the first N elements of the vector (where N is constant)



Rearranges elements such that the range `[first, middle)` contains the sorted `middle - first` smallest elements in the range `[first, last)`.

The order of equal elements is not guaranteed to be preserved. The order of the remaining elements in the range `[middle, last)` is unspecified.

- the content of the vector can be anything - in this case it’s a complex class
- we need to have a *helper function* defining what the operator **>** will do when we use it for sorting
- The implementation of this kind of algorithm in STL is much more efficient than anything we could intuitively do!

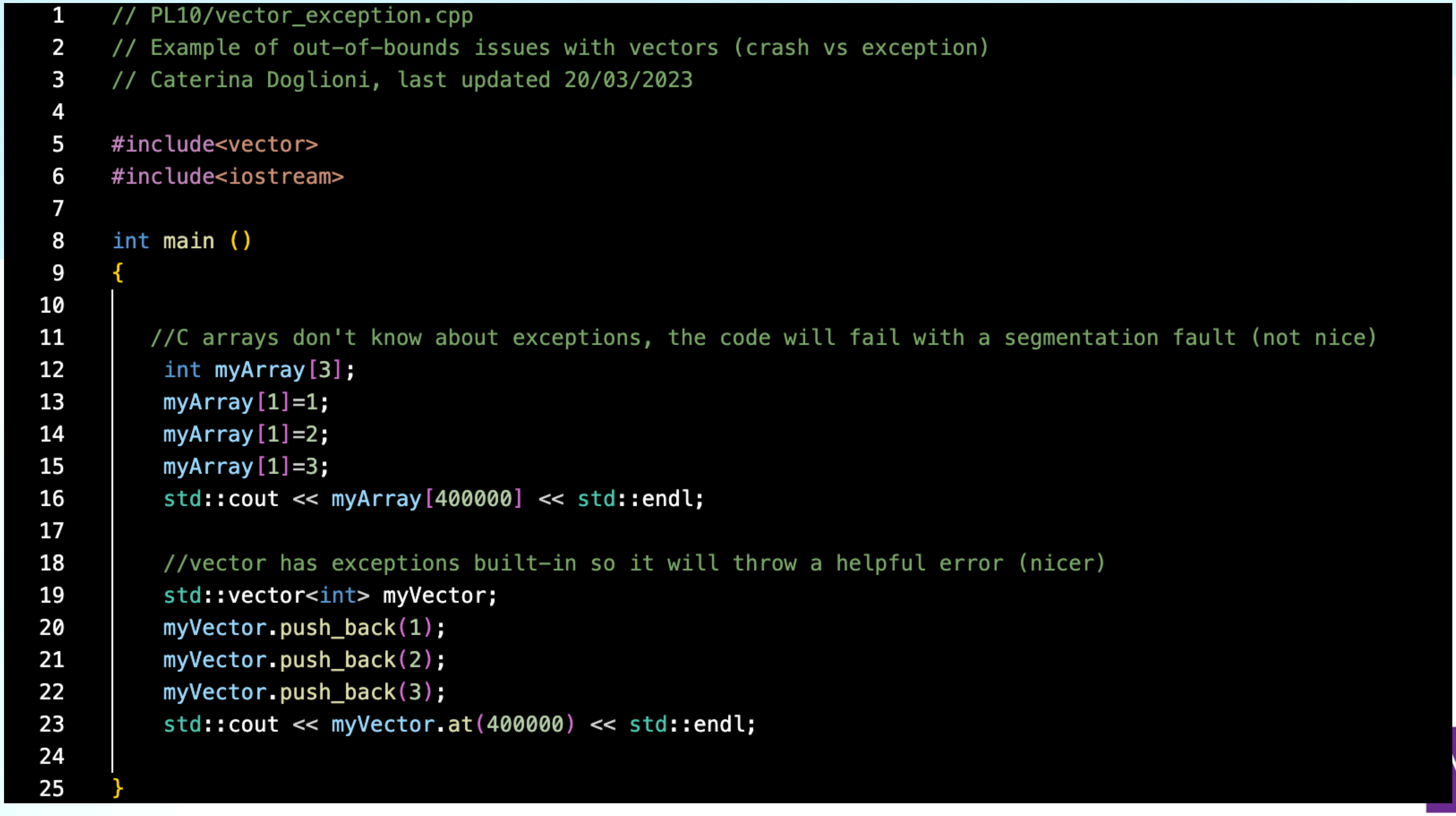
Part 2 / Video 2:
exceptions

Why exceptions?

- You're doing everything you can to have a **robust** OOP C++ code
  - Keeping interface and implementation separate
  - Only letting users modify data members via class methods
    - And doing proper input checking
  - But there may still be something in your code that ☠️ you really never ever want to happen ☠️
    - Example: segmentation fault, division by zero…but also more customised things for your code
    - Often you can only find out if something is going wrong **at run-time**
- Note: Python works with exceptions a lot, where user *asks for forgiveness, not permission*
- Exceptions are a tidy way to get your code to **quit with a helpful error message** if something unwanted (and/or unplanned) is about to happen

Let’s raise an exception

- STL containers all have exceptions built-in
- Example: ask for elements outside a `std::vector` vs `array`

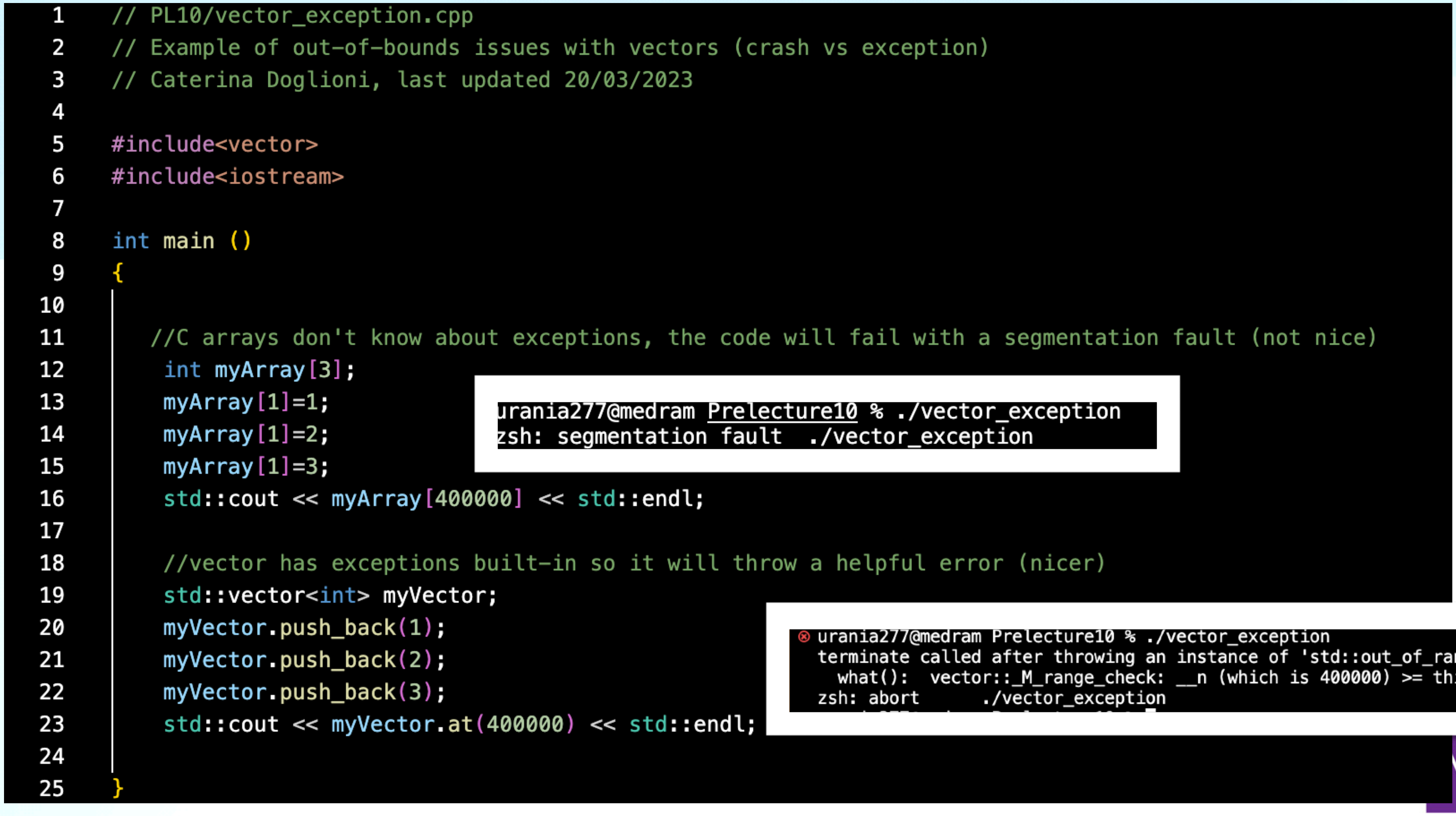


```cpp
// PL10/vector_exception.cpp
// Example of out-of-bounds issues with vectors (crash vs exception)
// Caterina Doglioni, last updated 20/03/2023

#include<vector>
#include<iostream>

int main ()
{

   //C arrays don't know about exceptions, the code will fail with a segmentation fault (not nice)
    int myArray[3];
    myArray[1]=1;
    myArray[1]=2;
    myArray[1]=3;
    std::cout << myArray[400000] << std::endl;

    //vector has exceptions built-in so it will throw a helpful error (nicer)
    std::vector<int> myVector;
    myVector.push_back(1);
    myVector.push_back(2);
    myVector.push_back(3);
    std::cout << myVector.at(400000) << std::endl;

}
```

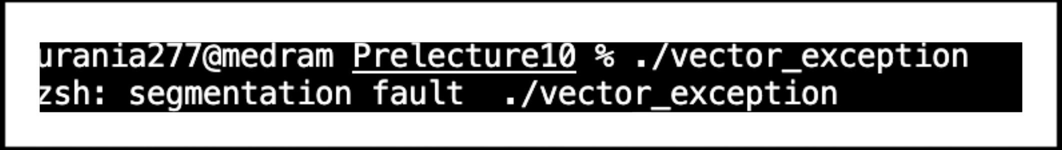
Code on GitHub at: Prelecture10/vector_exception.cpp

Let's raise an exception

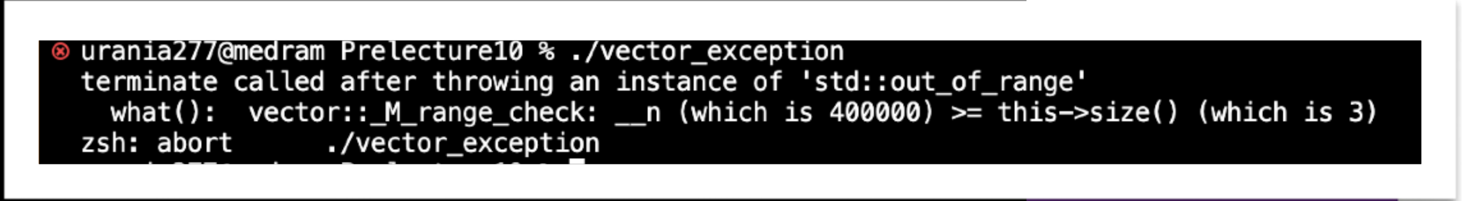
- STL containers all have many exceptions built-in
- Example: ask for elements outside a std::vector vs array



```cpp
// PL10/vector_exception.cpp
// Example of out-of-bounds issues with vectors (crash vs exception)
// Caterina Doglioni, last updated 20/03/2023

#include<vector>
#include<iostream>

int main ()
{

  //C arrays don't know about exceptions, the code will fail with a segmentation fault (not nice)
   int myArray[3];
   myArray[1]=1;
   myArray[1]=2;
   myArray[1]=3;
   std::cout << myArray[400000] << std::endl;

   //vector has exceptions built-in so it will throw a helpful error (nicer)
   std::vector<int> myVector;
   myVector.push_back(1);
   myVector.push_back(2);
   myVector.push_back(3);
   std::cout << myVector.at(400000) << std::endl;

}
```



```
urania277@medram Prelecture10 % ./vector_exception
zsh: segmentation fault  ./vector_exception
```



```
urania277@medram Prelecture10 % ./vector_exception
terminate called after throwing an instance of 'std::out_of_range'
  what():  vector::_M_range_check: __n (which is 400000) >= this->size() (which is 3)
zsh: abort      ./vector_exception
```

Code on GitHub at: Prelecture10/vector_exception.cpp

How to use exceptions

- Detecting and handling exceptions (run-time errors) is a key part of writing any program
- Especially true when using dynamic memory management
  - Could even be vital for mission-critical software
- C++ provides a neat method for this: `try`, `throw`, `catch`
  - The `try` keyword is used to look for exceptions
  - `catch` is used to decide what to do with them depending on what is happening
  - `throw` is used in an if statement that checks whether something may be going wrong
    - and transfers the code execution to go directly to `catch`

A pseudo-code exception example

//below are the variables indicating the kinds of things that can go wrong = the exceptions that will be thrown

```
const int divide_by_zero(-1)
const int bad_input(-1)
```

//helper function doing something, e.g. dividing two numbers

```
double divide(double numerator, double denominator) {
if (x==0) throw (divide_by_zero);
else return numerator/denominator; }

int main() {
```

    //get user input for two numbers that you want divided by each other

```
    try {
```

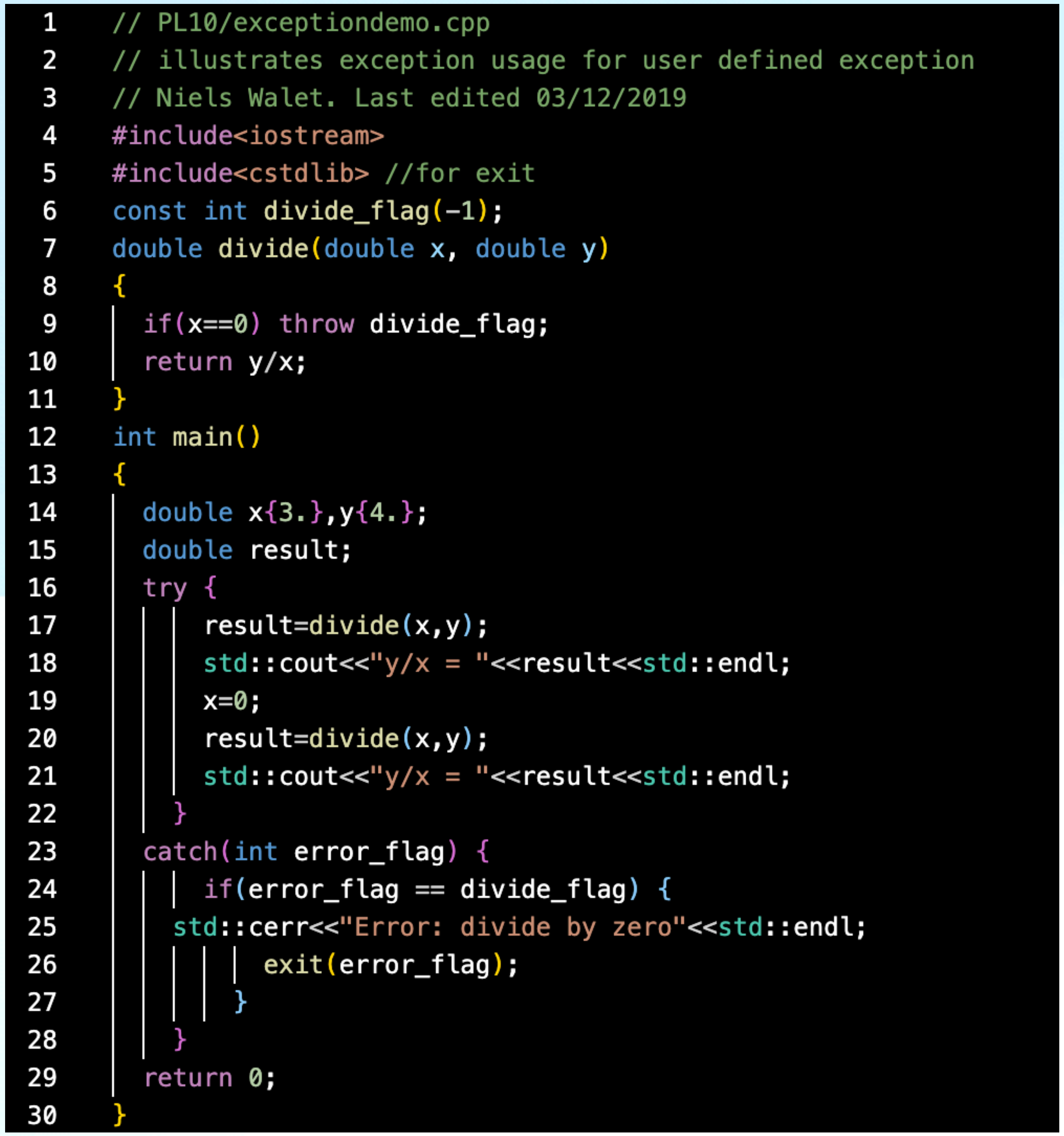
    // this is the block of code where something may go wrong
    // e.g. you are dividing two numbers, but these numbers may not be numbers, or the denominator may be zero…

```
    }

    catch(int errorFlag) {
```

    //this is the block of code where you check what happened and decide what to do
    //e.g. if `errorFlag` is `divide_by_zero`, write something helpful on screen and quit

```
    } …
```

A code exception example



```cpp
// PL10/exceptiondemo.cpp
// illustrates exception usage for user defined exception
// Niels Walet. Last edited 03/12/2019
#include<iostream>
#include<cstdlib> //for exit
const int divide_flag(-1);
double divide(double x, double y)
{
  if(x==0) throw divide_flag;
  return y/x;
}
int main()
{
  double x{3.},y{4.};
  double result;
  try {
      result=divide(x,y);
      std::cout<<"y/x = "<<result<<std::endl;
      x=0;
      result=divide(x,y);
      std::cout<<"y/x = "<<result<<std::endl;
    }
  catch(int error_flag) {
      if(error_flag == divide_flag) {
    std::cerr<<"Error: divide by zero"<<std::endl;
          exit(error_flag);
        }
    }
  return 0;
}
```

Code on GitHub at: Prelecture10/exceptiondemo.cpp

- Your `main()` is now also using a return code that can be used to identify in the terminal if your executable exited well
- Let's try to compare with just a division by zero - this is a lot more helpful!

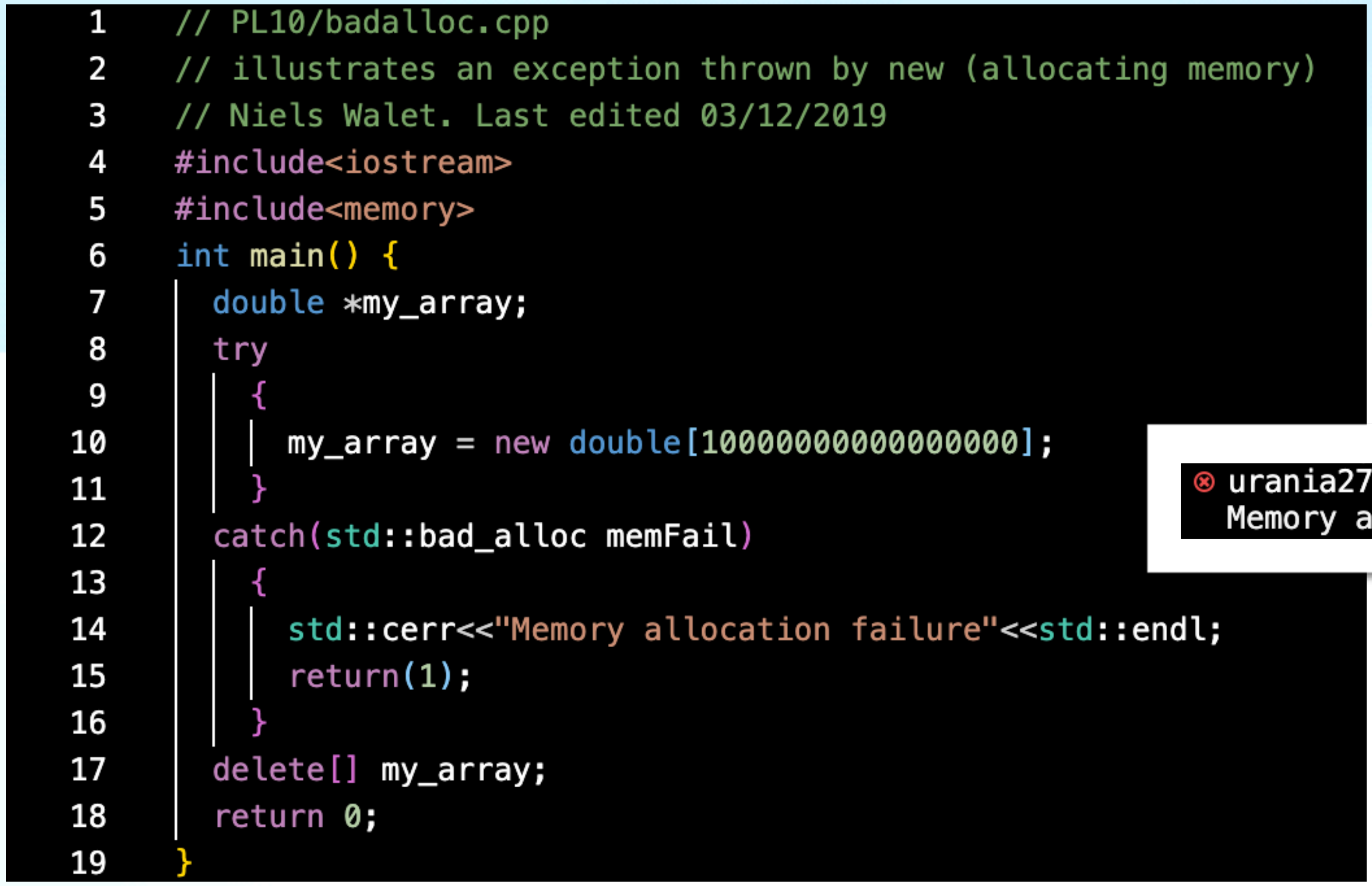
Good practices in exceptions

- You can have multiple catch statements for different datatypes where the appropriate one will be called (based on the type that is thrown), e.g.
  ```
  catch(int errorFlag) { ... }
  catch(double exceptionDouble) { ... }
  ```
  - Note that you can throw anything, not only int/double, and do something with it!
- Make sure you implement an appropriate catch for every throw
- When exception is thrown, program exits the try construct, and everything within that block is reset
- If throwing an object instantiated from derived class, catch it first (before base class objects) otherwise it will be caught by base class catch statement

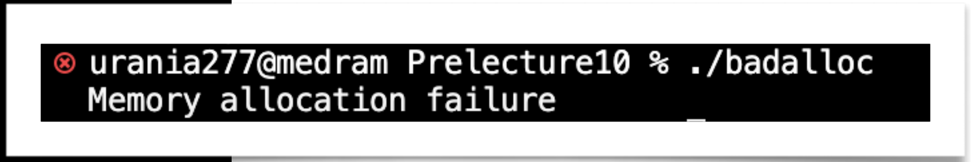
  ```
  // Wrong - object from derivedClass will be caught by first catch

  catch (baseClass B) { ... }
  catch (derivedClass D) { ... }
  ```

When allocating memory, use exceptions

- You are advised to use exception handling when allocating memory
    - If memory allocation fails (e.g. you've requested too much memory), an exception will be thrown of type `bad_alloc`



```cpp
// PL10/badalloc.cpp
// illustrates an exception thrown by new (allocating memory)
// Niels Walet. Last edited 03/12/2019
#include<iostream>
#include<memory>
int main() {
  double *my_array;
  try
    {
      my_array = new double[10000000000000000];
    }
  catch(std::bad_alloc memFail)
    {
      std::cerr<<"Memory allocation failure"<<std::endl;
      return(1);
    }
  delete[] my_array;
  return 0;
}
```



```
⊗ urania277@medram Prelecture10 % ./badalloc
  Memory allocation failure
```

Code on GitHub at: Prelecture10/badalloc.cpp

- A list of a number of possible exceptions in STL is here

In destructors, don't use exceptions

- Using exceptions in destructors isn't allowed in C++
  - Reason: see ISO C++ FAQ (who is aunt Tilda?)
  - TLDR: C++ destroys objects in a particular order, exceptions disrupt that flow. Also, it goes against the RAII principle.
- Exceptions in other functions as well are discouraged by some constructors at compilation time, e.g. by the Microsoft compiler
- This explains why some of you encountered a warning about this
- To make it go away, reassure the compiler that you're not throwing exceptions using the noexcept syntax
  - and if you are, read up why you shouldn't!

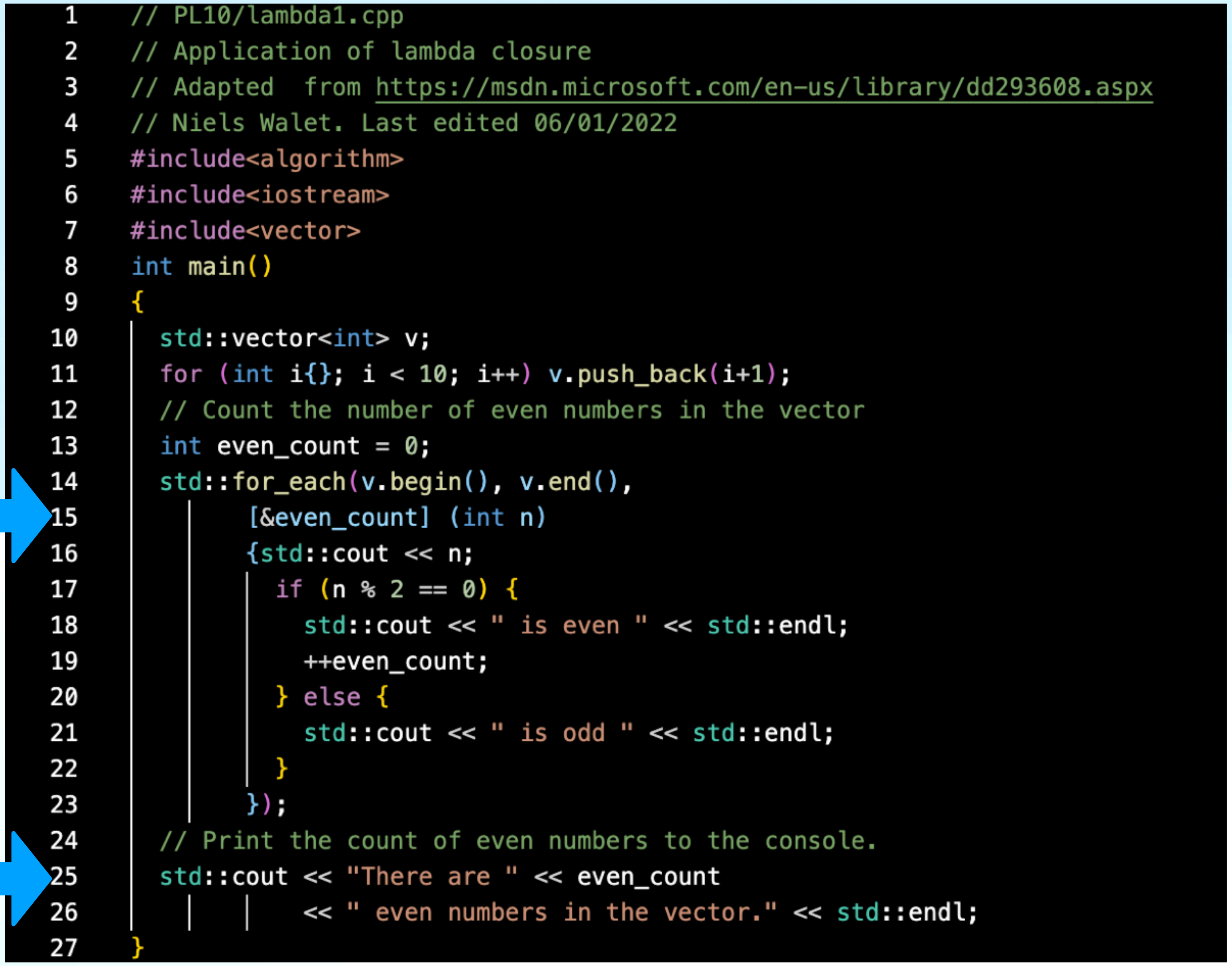
Part 3 / Video 3:
lambda functions + recap

MANCHESTER 1824
The University of Manchester

A very short introduction to `lambdas`

- Lambda function = anonymous unnamed short function in C++
    - You may have seen these in Python…
- You can use them when you want to write a simple and fast function that only using simple expressions
    - Also for readability: **simple function** that can be defined as near as possible to where it's called
- We will only give limited details on lambda functions (technically: lambda closures)
- A more complex alternative: functors (not covered in this course, see here for info)

A lambda function by example



```
1   // PL10/lambda1.cpp
2   // Application of lambda closure
3   // Adapted  from https://msdn.microsoft.com/en-us/library/dd293608.aspx
4   // Niels Walet. Last edited 06/01/2022
5   #include<algorithm>
6   #include<iostream>
7   #include<vector>
8   int main()
9   {
10    std::vector<int> v;
11    for (int i{}; i < 10; i++) v.push_back(i+1);
12    // Count the number of even numbers in the vector
13    int even_count = 0;
14    std::for_each(v.begin(), v.end(),
15          [&even_count] (int n)
16          {std::cout << n;
17            if (n % 2 == 0) {
18              std::cout << " is even " << std::endl;
19              ++even_count;
20            } else {
21              std::cout << " is odd " << std::endl;
22            }
23          });
24    // Print the count of even numbers to the console.
25    std::cout << "There are " << even_count
26              << " even numbers in the vector." << std::endl;
27  }
```

Code on GitHub at: Prelecture10/lambda1.cpp

- The code uses the `for_each` syntax as found in the `<algorithm>` header: it applies a function to each argument between begin and end.
- The lambda function definition is on line 15
- The [] part says it has access (by reference) to the `even_count` variable
- The () part shows that it takes an int as input
- The { } part is the definition of the function
- Then it's called via `even_count` in L25

MANCHESTER
1824
The University of Manchester

More lambda function details

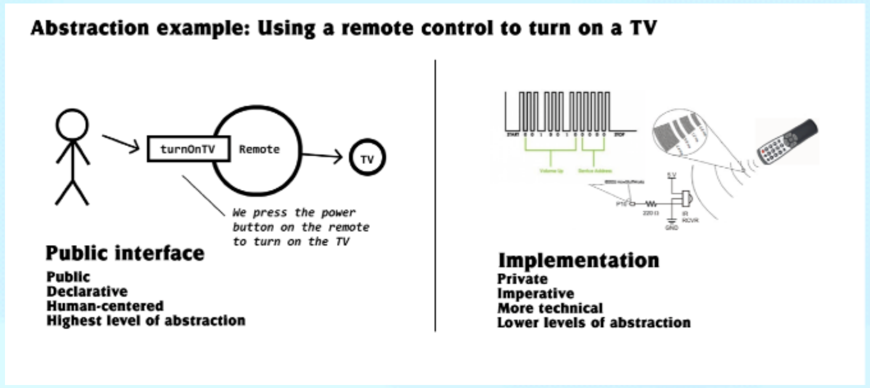
- The important part is what and how variables are captured in the lambda (made available later on for use with the lambda) using the square brackets:
  - [] Capture nothing
  - [&] Capture any variable used by reference (so we don't have to specify!)
  - [=] Capture any variable used by value
  - [=,&x] Capture any variable used by value, but x by reference
  - [x] Capture x by value, don't capture anything else
- The return type of a lambda is **void** by default;
  - for a simple return the compiler will work out what the return type is
    ```
    [] () { return 1; }// an int
    ```
  - you can also specify it with the **return value syntax (->)**
    ```
    [] () -> double { return 1.0; }// a double
    ```

MANCHESTER 1824 The University of Manchester

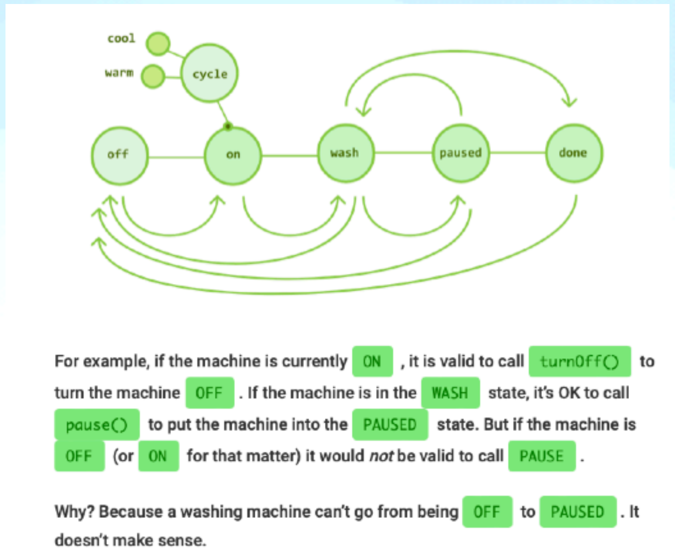
Closing words…

Overall course recap: OOP concepts

Most describe OOP in terms of these 4 principles:


Abstraction example: Using a remote control to turn on a TV
turnOnTV
Remote
TV
We press the power button on the remote to turn on the TV
Public interface
Public
Declarative
Human-centered
Highest level of abstraction
Implementation
Private
Imperative
More technical
Lower levels of abstraction

- Abstraction
  separate interface
  and implementation
  (remote control example)



For example, if the machine is currently `ON`, it is valid to call `turnOff()` to turn the machine `OFF`. If the machine is in the `WASH` state, it's OK to call `pause()` to put the machine into the `PAUSED` state. But if the machine is `OFF` (or `ON` for that matter) it would *not* be valid to call `PAUSE`.

Why? Because a washing machine can't go from being `OFF` to `PAUSED`. It doesn't make sense.

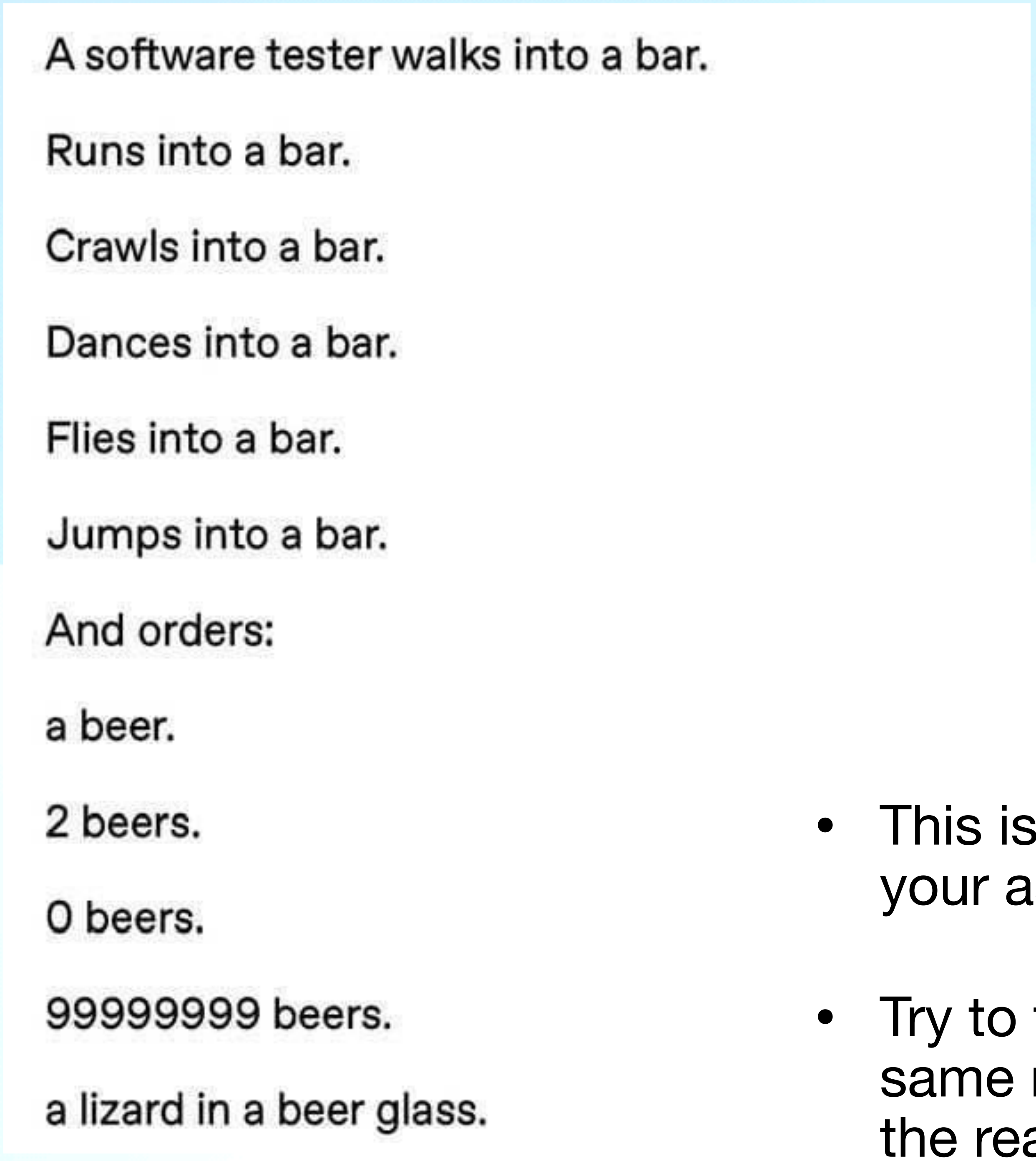
- Encapsulation:
  keep data private, alter properties
  via methods only (washing machine example)

- Inheritance:
  classes can be based on other classes to avoid code duplication

- Polymorphism:
  can decide **at run-time** what methods to invoke for a certain class, based on the object itself

Functionality/input checks: they are important!

…from the internet (couldn't find the source):



A software tester walks into a bar.

Runs into a bar.

Crawls into a bar.

Dances into a bar.

Flies into a bar.

Jumps into a bar.

And orders:

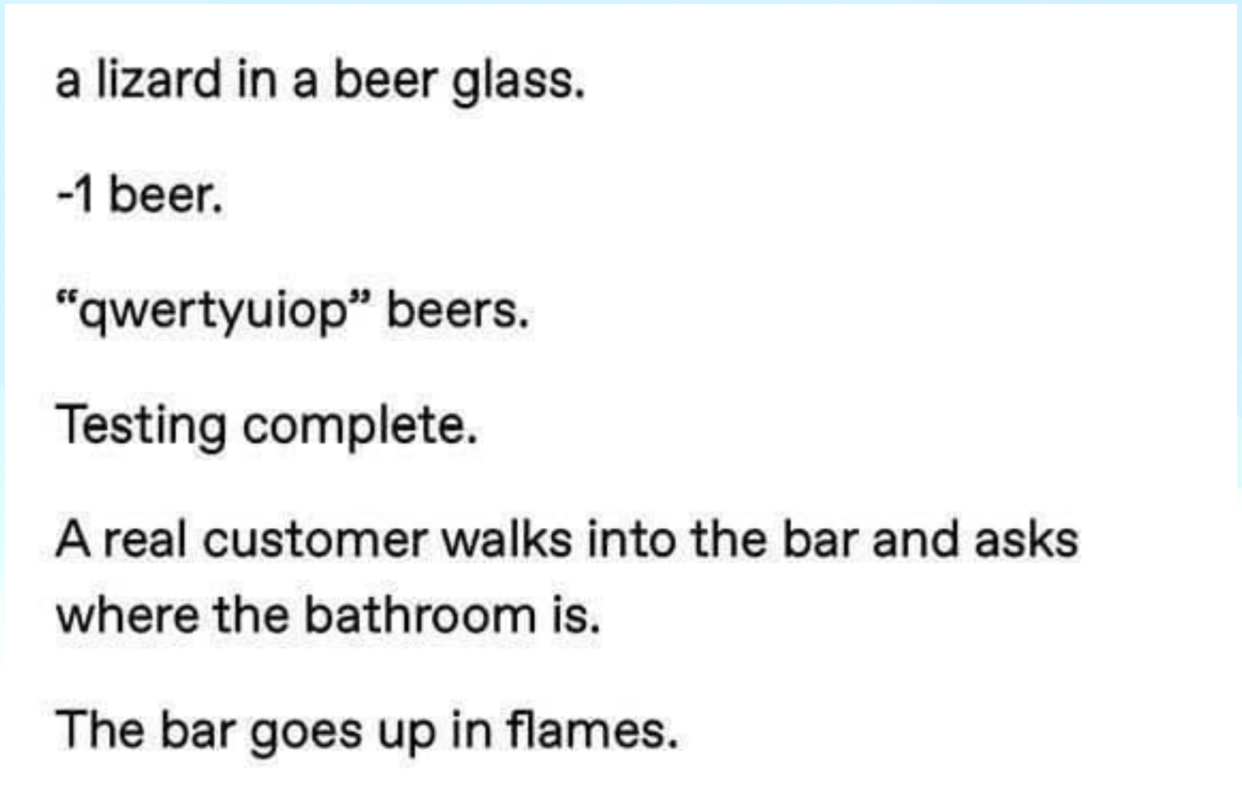
a beer.

2 beers.

0 beers.

99999999 beers.

a lizard in a beer glass.



a lizard in a beer glass.

-1 beer.

"qwertyuiop" beers.

Testing complete.

A real customer walks into the bar and asks where the bathroom is.

The bar goes up in flames.

- This is what I do when I mark your assignments
- Try to test your code with the same mindset (and make sure the real customer doesn't make the bar go up in flames either)

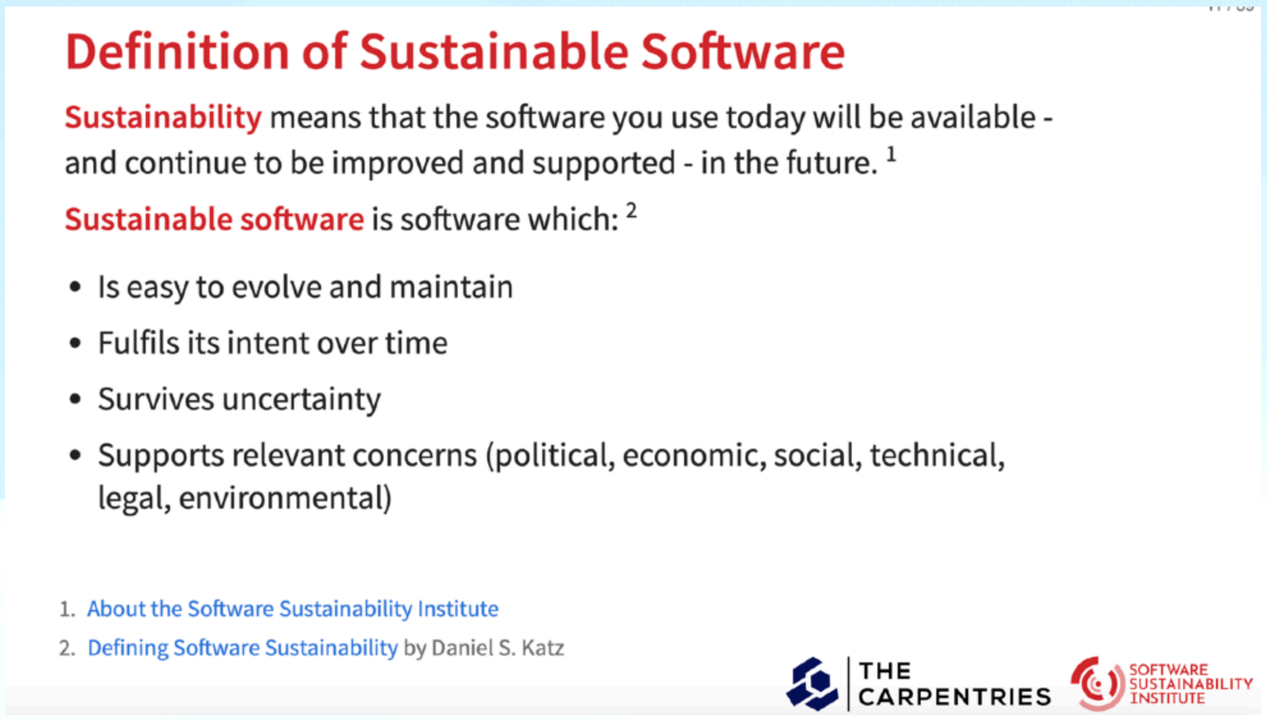
Overall course recap: key concepts

- In this course you were introduced to the main concepts of Object Oriented Programming
- Key concepts to understand:
  - **Classes** (the rules) and objects (the instances)
  - **Encapsulation** - we control how data are used
  - **Inheritance** - creating class super-structures
  - **Polymorphism** - one interface, multiple methods
  - **Class and function templates** - structures with generic types
  - **Organising code** - multiple files, headers and namespaces
  - **Good practice** - comments; handling exceptions
- Standardised C++17 has been around for 6 years (most if not all of its new features already in latest compilers), currently at C++21
- C++ is only one of the languages that use OOP (Java, C#, Python, Ruby, ...)
  - and some are based on C++ (Rust, Go(lang), . . .)
  - An interesting paper: which is the **most energy-efficient language**? See https://haslab.github.io/SAFER/scp21.pdf (I'm working on this question with many other great researchers & students , if you're interested in joining let me know!)
- **Remember:** secret to good programming is practice, and having fun with it!

MANCHESTER 1824
The University of Manchester

Closing with sustainable software

- Eli Chadwick's talk and recording:



Definition of Sustainable Software

Sustainability means that the software you use today will be available - and continue to be improved and supported - in the future. [1]

Sustainable software is software which: [2]

- Is easy to evolve and maintain
- Fulfils its intent over time
- Survives uncertainty
- Supports relevant concerns (political, economic, social, technical, legal, environmental)

1. About the Software Sustainability Institute
2. Defining Software Sustainability by Daniel S. Katz

(my summary & interpretation of)

- Eli hands-on suggestions

1. Don't assume your code is perfect (out of pride or shame)
2. Introduce a version control system
3. Make changes little and often
4. Get your code to compile, build and run on a machine that isn't yours
5. Invest some time in automating and formalising your tests and your code
    1. This is also why we tell you to test with bad input
6. Make your code modular, build it up from simple interacting components
7. Share your code: when it's not ready (to get feedback) and when it's ready (make it Open Source)
8. Join a community of practice of people who discuss similar questions/challenges

- Website for the Software Sustainability Institute
- An interesting paper: which is the **most energy-efficient language**? See https://haslab.github.io/SAFER/scp21.pdf (I'm working on this question with many other great researchers & students, if you're interested in joining us for research experience let me know!)