Good Practice

Data Processing Course,
I. Hrivnacova, IJCLab Orsay

- Writing Correct and Good Programs
- Formatting Source Code
- Choosing Names
- Coding Conventions
- Commenting Code
- Programming Errors
- Debugging Programs
- Testing

Writing Correct and Good Programs

- Good programs:
  - that produce the correct results
  - that others (and you yourself three days later) can understand, so that the programs can be maintained
- Good practice:
  - Format and layout of the source code with appropriate indents
  - Choose good names that are self-descriptive and meaningful
  - Follow established convention so that everyone has the same basis of understanding.
  - Provide comments to explain the important as well as salient concepts
  - Write your program documentation while writing your programs.
  - Avoid un-structured constructs, such as break and continue, which are hard to follow.

Format Source Code

- Use appropriate indents, white spaces and white lines.
- **Use consistently 2 or 3 spaces for indent**, and blank lines to separate sections of codes.(one at a time)

```
while(!stream.eof())
      {
      std::string word;
      std::getline (stream,word);
      if(s != "END_IMAGE")
      {
      std::cout << word << std::endl;
      }
      else
      {
      std::cout << std::endl;
                  return true;
            }
```

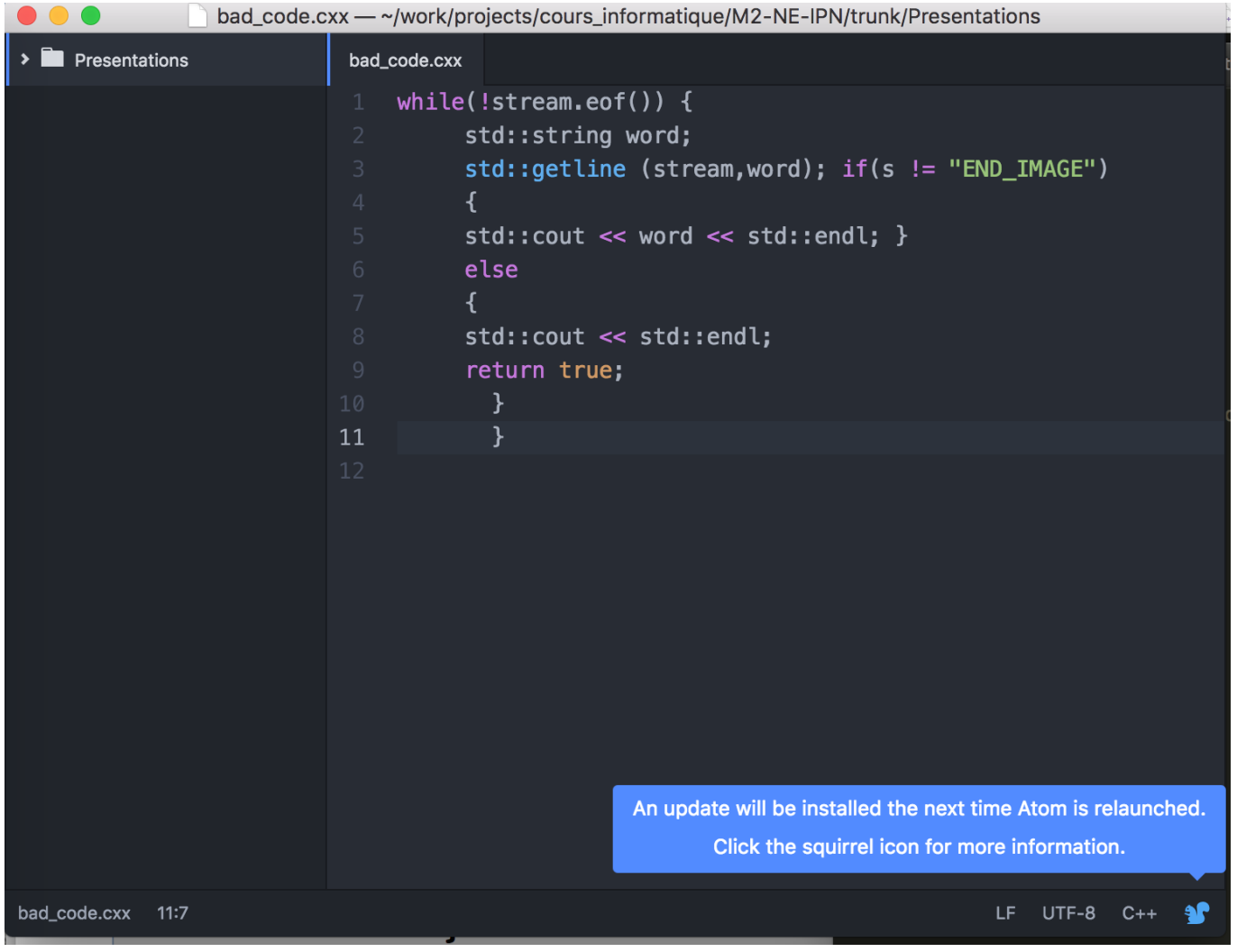
```
while(!stream.eof())
{
  std::string word;
  std::getline(stream, word);

  if(s != "END_IMAGE")
  {
    std::cout << word << std::endl;
  }
  else
  {
    std::cout << std::endl;
    return true;
  }
}
```

Format Source Code

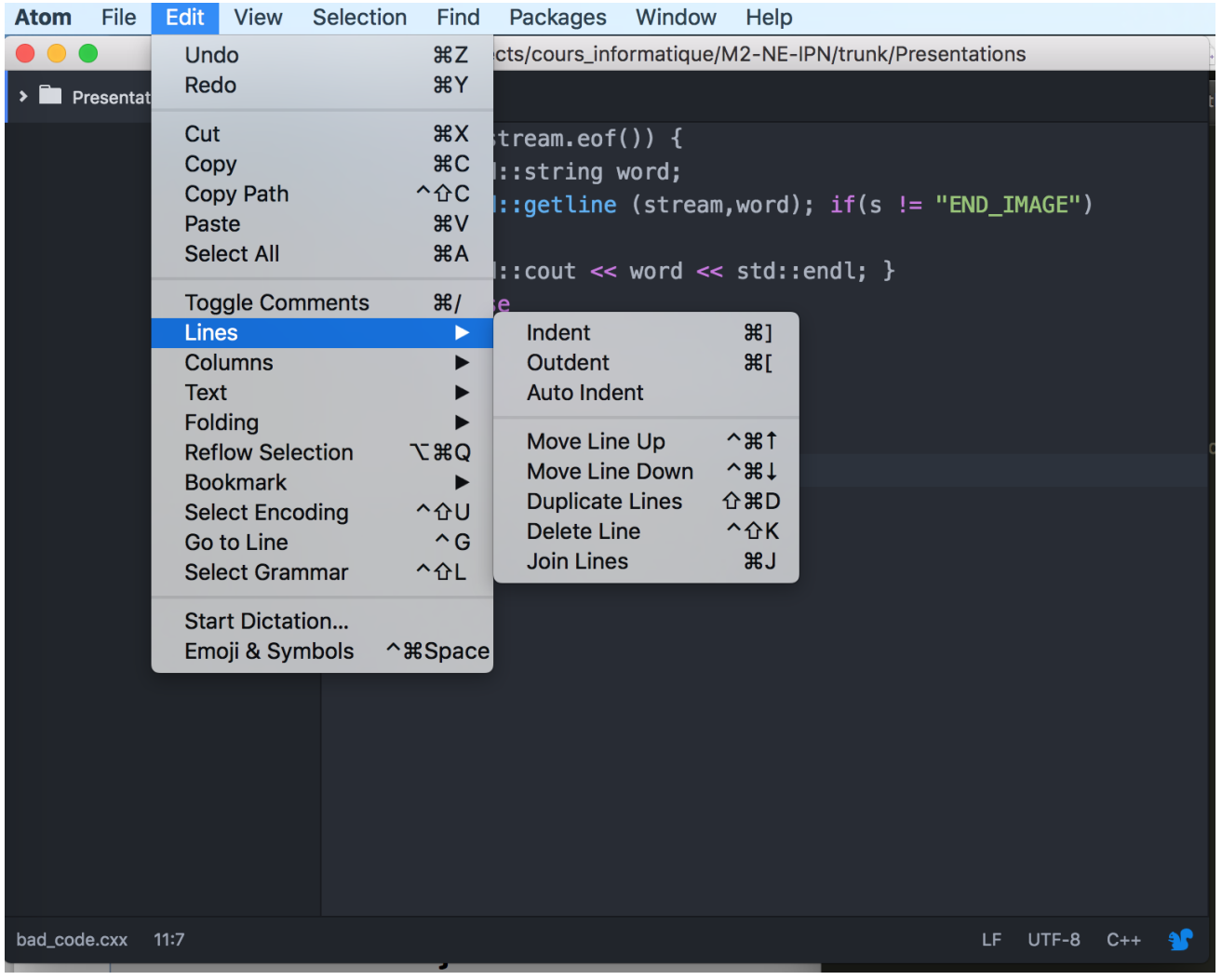
- Different styles are possible, we choose one and follow it in the whole project

```
while(!stream.eof())
     {
     std::string word;
     std::getline (stream,word);
     if(s != "END_IMAGE")
     {
     std::cout << word << std::endl;
     }
     else
     {
     std::cout << std::endl;
               return true;
          }
     }
```

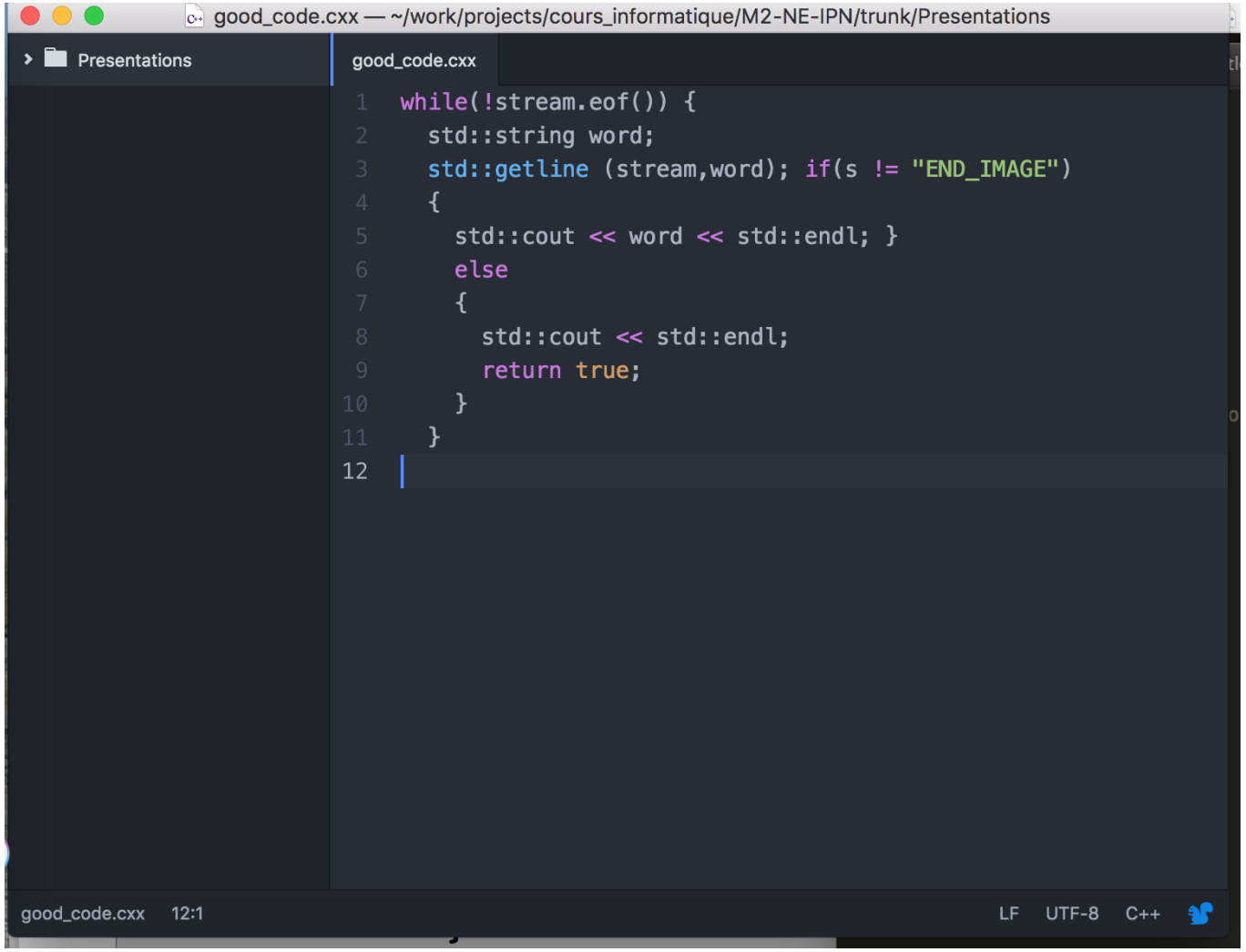
```
while(!stream.eof()) {
  std::string word;
  std::getline(stream, word);

  if(s != "END_IMAGE") {
    std::cout << word << std::endl;
  }
  else {
    std::cout << std::endl;
    return true;
  }
}
```



Code improperly indented



Auto Indent function available in Atom editor



Code well formatted

Choosing Names

- Choose good names that are self-descriptive and meaningful, e.g., row, col, size, xMax, numStudents.
- Do not use meaningless names, such as a, b, c, d.
- Avoid single-alphabet names (easier to type but often meaningless), except common names likes x, y, z for co-ordinates and i for index.

Coding Conventions

- The larger projects usually establish coding convention so that everyone has the same basis of understanding.
  - See for example the Google C++ Style Guide
- Conventions used in our course:
  - The names of local variables and function arguments start with a lower-case letter
    - `int length = 0;`
    - `bool read (std:: istream & stream)`
  - Camel case convention: in the names which consist of several words, words are written together and start with a capital letter (except the first one following general rules)
    - `readImage, myFile, myOutputFile`
  - Names and comments are in English

Comments & Documentation

- Provide comments to explain the important as well as salient concepts
  - A good comment goes beyond stating the obvious, or what is already clear from the proper naming of variables / function
- Write your program documentation while writing your programs
  - Documentation can be embedded in the source code or written separately (a user guide document)
- In our course:
  - **Keep the commented lines given with the start code**
  - You may eventually modify them to reflect what has been done, for example

```
// Implement a for loop over the data
```

  - Change to

```
// Loop over the data
```

Programming Errors

- **Compilation Error** (or Syntax Error): can be fixed easily.
- **Runtime Error**: program halts pre-maturely without producing the results - can also be fixed easily.
- **Logical Error**: program completes but produces incorrect results.
  - It is easy to detect if the program always produces wrong result.
  - It is extremely hard to fix if the program produces the correct result most of the times, but incorrect result sometimes.
  - A good testing strategy is needed

```
int numbers[6]
  = {11, 22, 33, 44, 55, 66};
int index = getIndex();
// Index can be out of bound!
cout << numbers[index] << endl;
```

Initialization of Variables

- A variable declaration should generally be followed by its initialization
- `int length = 0;`
- **Use of uninitialized variables may lead to unpredictable results**

Debugging Programs

Here are the common debugging techniques:

- **Study the error messages!**
  - **Always start from the first one** (the other ones may be be misleading due to the first one)
  - Do not close the console when error occurs and pretending that everything is fine. This helps most of the times.
- **Insert print statements** at appropriate locations to display the intermediate results.
  - It works for simple toy program, but it is neither effective nor efficient for complex program.
- Use a (graphic) debugger.
  - This is the most effective means. Trace program execution step-by-step and watch the value of variables and outputs.
- Advanced tools such as profiler (needed for checking memory leak and function usage).

Testing

- It is impossible to try out all the possible outcomes, even for a simple program.
- Program testing usually involves a set of representative test cases, which are designed to catch the major classes of errors.