CM 10227/50258: Lecture 1

Dr. Rachid Hourizi and Dr. Michael Wright

October 2, 2017

Table of Contents

Introduction

Aims

- This course provides an introduction to programming
  - problem analysis
  - requirements elicitation
  - design
  - code writing
  - evaluation
  - problem identification and repair (debugging)
  - maintenance
  - extension

Outcomes

- On completion of the course, you will be able to
  - Design, implement, test and evaluate programs using
    - ★ a procedural paradigm (using a small subset of the C programming language)
    - ★ an object oriented paradigm (using the Java programming language)
  - Understand and use key data structures

Labs, Lectures and Other Sessions

- In order to help you to learn the concepts at the heart of the course, we provide the following sessions each week:
  - 2 hour lecture
  - 1 hour tutorial
  - Lab sessions

If you need additional help, we also provide

-
  - 15 minute drop in sessions: Sign up in advance from week 4
  - Peer assisted learning sessions: No sign up needed
- Each of these sessions should be on your timetables
- Details can also be found on Moodle

- If, on the other hand, you find the first few lab exercises easy, we would like to give you the opportunity to stretch yourselves!
- With that in mind, we will also be running advanced programming labs from weeks 4-11
- Those labs are
  - mainly aimed at those with stronger programming experience
  - Focused on "competitive programming" i.e. programming competitions such as UPIEPC and NWERC
- We will circulate details of the Advanced programming exercise once the main labs are underway

Assessment (CM10227)

- The course is assessed in two parts
  - 50 percent exam
    - ★ 1 exam in January 2018
  - 50 percent coursework
    - ★ 6 smaller lab sheets (or an equivalent 'advanced programming' exercise)
    - ★ 2 larger courseworks
- I will go into more detail about each part of this assessment in the second lecture

PLEASE NOTE THAT YOU MUST ACHIEVE A PASS - I.E. 40 PERCENT OR ABOVE - IN BOTH COURSEWORK AND EXAM IN ORDER TO PASS THIS COURSE

Assessment (CM50258)

- The course is assessed in two parts
  - 50 percent exam
    - ★ 1 exam in January 2018
  - 50 percent coursework
    - ★ 3 smaller lab sheets
    - ★ 1 larger programming coursework
    - ★ 1 code analysis and extension
- We will go into more detail about each part of this assessment in your first lab

PLEASE NOTE THAT YOU MUST ACHIEVE A PASS - I.E. 40 PERCENT OR ABOVE - IN BOTH COURSEWORK AND EXAM IN ORDER TO PASS THIS COURSE

Moodle

- More detail on both learning sessions (lectures, labs etc.), coursework and other information about the course can be found on our Moodle pages:
  - `http://moodle.bath.ac.uk/course/view.php?id=30475`
  - `https://moodle.bath.ac.uk/course/view.php?id=57411`
- Please check these Moodle pages regularly
- They will be used for announcements and distribution of material for lectures, labs, coursework
- We will put copies of the lecture slides on Moodle each week (after the lectures)
- Along with other information about the coursework
  - e.g. a more detailed version of the aims and objectives shown on previous slides

Support

- If you have questions, we are more than happy to help
- But please take the initiative
  - Ask the lecturers
  - Ask the lab tutors
  - Use the Moodle forums
    - ★ Though please don't post code on them
  - Use the mailing list programming1@lists.bath.ac.uk
  - Copies of both your Moodle forum posts and your mailing list questions will be sent to the lecturers and all the tutors
- Contacting us as a group is (very) likely to get a faster response than contacting any one person individually

Additional resources

- You should get into the habit of supplementing course material with your own reading e.g.
  - Kelley and Pohl, 2003, "C by dissection"
  - McDowell and Pohl, 2006, "Java by dissection"
- remember that you can buy books second hand (e.g. from abebooks.co.uk) as well as new (e.g. from amazon.co.uk)

Additional resources

- You should also use the internet as a source of information e.g.
  - Google
    - ★ the name of a concept that you dont understand (e.g. “recursion”)
    - ★ an instruction that isnt working as you expected
    - ★ undesired compiler output
    - ★ etc.
  - Get familiar with the places that professional programmers give/offer advice e.g.
    - ★ stackoverflow.com

Other support

- Whilst you should be targeting professional programmers' discussions with your reading, you may, occasionally find that you need to review the fundamentals of a programming topic.
- In that case, consider the possibility of an onlne tutorial (from KhanAccademy, Udemy etc.).
- As a starting point, you may want to consider the list of providers here:
  - https://www.upwork.com/blog/2014/02/10-top-sites-online-education/
- Be aware, however, that we will examine and assess your progress against our curriculum rather than theirs. Make sure that you are familiar with all the material on our slides
- **Please tell us if you are struggling. As a rule of thumb, if you feel that you need the "from scratch" support provided by an online tutorial, you should also be asking for help from the lab tutors and/or Michael/Rachid**

Other support

- Other forms of support include
  - Your personal tutor
  - The computer society
  - Peers (as long as you do not work together for classwork)

Plagiarism

- Do, however, be careful of plagiarism and collusion
  - Reuse: Acknowledging someone else's work (fine)
  - Plagiarism: Presenting someone else's work as your own (bad)
  - Collusion: Working closely with others when work is meant to be individual (bad)

How to succeed on this course

- Attend the lectures
- Attend the labs
- Spend time on this unit outside the lectures and labs
- **Do not wait until the last minute to do your work**
- Get programming experience
- Don't be afraid to make mistakes; learn from them

How to succeed on this course: time

- This course represents 2/5 of your credits this semester
- Assuming 37.5 working hours per week
- Then you can assume (as we do) that this course will take 15 hours per week
- Assuming 3 hours in lectures and tutorials
- And a minimum of two hours in labs
- This leaves 10 hours for you to
  - do further research into your programming
  - and practise what you learn
- Please use them!

First Programs

What is a program and what is programming?

- A program is a set of instructions that tells a computer how to perform a computation
- Example computations include item Mathematical computations
  - solving a system of equations
  - finding the roots of a polynomial
- Symbolic computations
  - Searching and replacing text in a document
  - Interpreting text in a file (e.g. interpreting another program)

- Programming can be thought of as the process of breaking a large, complex real world task up into smaller and smaller sub-tasks until eventually the sub-tasks are simple enough to be performed with a simple (programmable) instruction. e.g.
  - Get data from the keyboard, or a file, or some other device.
  - Perform basic mathematical operations e.g. addition and/or multiplication.
  - Check for certain conditions and execute an appropriate sequence of statements in line with those condition
  - Perform some action repeatedly, usually with some variation
  - Display data on the screen or send data to a file or other device.

- It is helpful think of two types of programming languages:
  - High-level
    - ★ Java
    - ★ Python
    - ★ C++
    - ★ Perl
    - ★ C (arguably)
  - Low-level (assembly or machine languages) - Computer specific
    - ★ Intel x86 (Pentium), Apple - IBM PC compatibles
    - ★ Motorola/ IBM PowerPC - IBM workstations and servers
    - ★ Sun SPARC - Sun workstations and servers
- Computers only execute low level languages.
- Programs written in high level languages are 'translated' to the machine language of the specific computer.

- Advantages of low-level programming languages:
  - program can be tuned to specific computer for maximum execution speed and minimum memory consumption.
- Advantages of high-level programming languages:
  - Easier to program
  - Less time to write
  - Easier for people to read
  - Easier for people to correct
  - Much easier to port, or modify to run on different computers
- Today almost all programs are written in high-level programming languages.

A specific programming language: C

- In the first part of this course, we will use the C programming language as a basis for discussing the specification, testing, repair and documenting introduced in previous slides
- (In the second part we will use Java as a basis for further discussion)

A first C program: HelloWorld.c

```c
/* Hello World program */
#include <stdio.h>

main()
{
        printf("Hello World");
}
```

```
Hello World
```

- The code on the previous slide prints the phrase “Hello World” on whichever device you are using to receive output (usually a screen).
- You will notice that it takes quite a lot of code simply to produce that output
- It is worth taking a moment to discuss each part of that code

- The first part of the code is a simply a note to the programer(s)

```
/* Hello World program */
```

- In this case, the text is marked as a comment by /* at the start of the line and */ at the end

- Commenting is an absolute must.
- It allows you to
  - explain your code
  - refresh your memory when you return to it
  - help others to read your code
  - re-use your code
- We will be giving marks for comments in your coursework (and taking them off when you dont provide them)

- The two following approaches can both be used to insert comments into C code

```
/* Everything ignored on one line */
```

and

```
/* Everything ignored on
* each
* line of text
* marked as part
* of a continuous comment
*/
```

- The second line of code tells the compiler to use ("include") code from another file

```
#include <stdio.h>
```

- This allows us to re-use code that has already been written – another important concept to which we will return later

- The third line of code tells the compiler that the following lines i.e. everything between { and } makes up the main part of the program
- When the program is run, the first of those lines will be executed first
- The name given to the code marked as main() is "The main() function"
- Every C program must have a main() function

```
main()
{
        code goes here
}
```

- The next line of code (i.e. the content of the main() function)is the part that does the printing:

```
printf("Hello World");
```

- It is useful to think of this code in two parts – the command or function that tells the computer what to do:

```
printf();
```

- and the (input) data to which that command is applied

```
"Hello World"
```

- it is also worth noting that the line ends with a semicolon (";")
- C uses a semicolon to indicate the end of each instruction

- Before looking at the “Hello World” example
- We introduced the idea of instructions written in “High-level” programming languages being “translated” to machine specific instructions

- The translation of programs written in high level languages to machine specific instructions can be achieved in more than one way:
- We can for example use an Interpreter
  - Input (Code) $\rightarrow$ Interpreter $\rightarrow$ Final Output
    - ★ The Interpreter reads in the (high level) source code
    - ★ After interpretation, the final output is delivered to the user e.g. appears on a screen

- An alternative approach is to use a compiler
  - Input code $\rightarrow$ Compiler $\rightarrow$ Object Code $\rightarrow$ Execution $\rightarrow$ Output
    - ★ The compiler reads in the source code
    - ★ and generates object code as output
    - ★ you execute that object code (one way or another)
    - ★ and the end result is delivered to the user e.g. appears on a screen

- All high level programming languages (e.g. C, Java, Python) can, in theory be interpreted or compiled.
- As we will see when we move to Java later in the course, more complex conversions also exist
  - e.g. compilation to an intermediate form of code which is then interpreted
- Usually, however, a 'standard' set of tools are provided that follow one route or another
- In that context, you will hear discussion of C, Python and Java being 'Compiled languages' (since the standard tools provided follow the compilation route)
- We will use that convention in this course

In order to compile and run our example C program "Hello World" on a University UNIX or Windows machine:

- create a file called HelloWorld.c
- Note: use a plain text editor such as Notepad++ or emacs (Do not use Word)
- save it in a location that you can find again from the command line
- navigate to that location and run the gnu compiler:

continued on next slide

- Then type the following command
  - gcc HelloWorld.c
  - Note: You can use other variations of this command - see below.
  - To start with, however, stick to this one
- When the compiler has compiled (and linked) your program it will create an executable (runnable) version. By default, that executable version of your program will be called a.out on a UNIX machine and a.exe on a Windows machine
- you can now run the executable version of your program by typing ./a.out on the command line
- You should now see the output generated by the program

- **We have put a video on moodle showing the compilation and execution of a program being compiled and run**
- the tutors will help you with compilation and running a program locally in the first lab
- either way, it is a useful exercise to copy the HelloWorld program at the start of this section and make sure that you can compile and run it without making any changes

- If you have never compiled and run a program before, you can try entering your code into the online environment at https://www.codechef.com/ide
- that environment handles the compilation and run steps for you
- It allows you to enter your code and (after pressing a button) see the results of running your code on screen
- Make sure however that you have chosen C (Gcc-4.9.2) in the box above the coding area

- In the HelloWorld example, above, the text (more formally a "String" of characters) can of course be replaced with different text (a different "String").
- We could, for example, print "Goodbye World":

```
/* Goodbye World program */
#include <stdio.h>

main()
{
        printf("Goodbye World");
}
```

```
Goodbye World
```

- We could also use two printf() commands to print both Strings to the screen:

```
/* Hello World program */
#include <stdio.h>
main()
{
        printf("Hello World \n");
        printf("Goodbye World");
}
```

```
Hello World
Goodbye World
```

- Note the use of the newline (\n) at the end of the first print statement. Without that newline, the program would try to print both statements on the same line

- We would, however, have to recompile and re-run the code once we had made that change in order to see the new String printed to the screen
- Strings which cannot be changed once the program is running (e.g. “Hello World” and “Goodbye World”, above) are described as String “literals”.

- A more flexible approach to printing data to the screen is to use a variable whose value can be changed whilst the program is running ("at runtime")

Variables

- Variables can be thought of as named containers (locations in memory) that hold some information (a value)

```
a = 3;
```

Variables

- In C all variables have a 'type' – i.e. a description of the data that can be held within the named container
- Providing a description of the 'type' allows the C compiler to allocate an appropriate amount of memory for data held within the container.

Variables

- When discussing variables (and many other programming topics), you will find that a shorthand descripton is often used to describe that variable (named location in memory that holds a value)
  - There is sufficient memory reserved at the location assiciated with the name ‘a’ to hold an integer becomes
  - the variable ‘a’ refers to an integer or
  - a is an integer

Variables

- A full list of C variable (or data) types can be found online
- but at this point, you dont need to understand them all
- to start with, we will focus on just three:
  - variables that refer to an integer are of type int
  - variables that refer to a floating point number are of type float
  - variables that refer to a floating point number that is allocated (approximately) double the amount of memory allocated to a float are of type double

```
/*declaration and assignment
*of an integer (int) variable named myInt
*and a float named myFloat
*/
int myInt=7;
float myFloat=7.264;
```

- You **declare** a variable by specifying its type and name
  - int myInt
- You **instantiate** a variable the first time that you assign a value to it
  - myInt=7;
- You can continue to assign values to a variable (overwriting the previous value each time)
  - myInt=24;

- If you try to access a name before it is been properly created (declared and instantiated), you will get an error or the program will crash.

- This code prints the integer 7 to the screen

```
/* Print an Integer */
#include <stdio.h>
main()
{
   int a=7;
   printf("The integer is %d\n", a);
}
```

```
The integer is 7
```

- This code does not print a to the screen (it returns an error or crashes the program)

```
/* Print an Integer */
#include <stdio.h>
main()
{
  printf("Integer that you have entered is %d\n", a);
  int a=7;
}
```

- Note that the %d in the printf() statement in these examples
- %d is an example of a format specifier
- It tells printf() that an integer should be included as part of the string to be printed (or more specifically, a decimal integer, hence the 'd')
- The integer (i.e. the one refered to by a) is then specified after a comma
- you will need to use a format specifier each time you try to include a variable in your print statements

- The format specifier for a floating point number (i.e. a float or a double) is %f

```
#include <stdio.h>
main()
{
  float a=32.654;
  printf("the float held in a is %f\n", a);
  a=1.7;
  printf("the float held in a is %f\n", a);
  a=156.797;
  printf("the float held in a is %f\n", a);
}
```

```
the float held in a is 32.653999
the float held in a is 1.700000
the float held in a is 156.796997
```

- C is pretty liberal in the range of names that it allows you to use for variables:
- Characters which may be used in a C variable name include
  - Underscore (_)
  - Upper Case Letters (A-Z)
  - Lower Case Letters (a-z)
  - Digits (0-9)
- so permissable variable names include a, _myInt and _myInt99

- Some characters may not, however, be used in variable names:
- Blanks and commas may not be used
- special symbols other than underscore may not be used
- first character should be an upper case character, a lower case character or an underscore

- there are also some reserved words (words that C reserves for specific uses), which may not be used
- you cannot, for example use the word int as a variable name since C reserves it for use as a type identifier (see previous slides)
- a list of reserved words can be found at http://crasseux.com/books/ctutorial/Reserved-words-in-C.html

Building more complex computations: Some vocabulary

- an **operator** is a character which describes an action to be carried out
- the actions themselves are called operations
  - +, -, *, / and = are all operators

- a **statement** is the smallest standalone element of a **programming** language which expresses some action to be carried out
  - e.g. an assignment statement:
    - ★ int a = 5;
- **statements** can contain any number of operations
  - e.g. int a=1+2*3/4-5+6-7*8/9

- **An expression** is a combination of constants, variables, operators and functions that .. produces another value
  - e.g. 1+2*3/4-5+6-7*8/9 in the statement above.

- We can use printf() to print the value of an expression

```
#include <stdio.h>
main()
{
  int a=4;
  printf("twice the value of a is %d\n", a+a);
  printf("three times the value of a is %d\n", 3*a);
  printf("half four times a is %d\n", 4*a/2);
}
```

```
twice the value of a is 8
three times the value of a is 12
half four times a is 8
```

- note the use of the mathematical operators +,-, * (multiply) and / (divide)

- When more than one operator appears in an expression the order of evaluation depends on the rules of precedence.
- C follows the same rules as mathematics: PMDAS
- Parenthesis have highest precedence
  - used to force desired evaluation order:
    - ★ 2*(3-1) is 4, and (1 + 1)*(5-2) is 6
  - can also be used to make expression easier to read:
    - ★ (*minute* * 100)/60
- Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence.
  - 2*3-1 yields 5 rather then 4, and 2/3-1 is -1, not 1

- NB dividing one integer by another (e.g. 22/7) returns another integer (3) rather than a more accurate result (3.1428)
- If you want to return a floating point result, you need to use at least one floating point value in the original
  - 22.0/7
  - 22/7.0
  - 22.0/7.0

- mathematical functions
  - There are, of course, situations in which we might want to perform more complex computations than can easily be achieved simply through addition, subtraction, division and multiplication
  - We might, for example want to be able to calculate the sqare root of a number or raise a second number to a power

- One approach to performing these computations is to write code to do within each program requiring that functionality
- A much simpler route to achieving the same goal, however, is to see whether someone else has written the code that you need and use theirs
- Whether you write that code yourself or use some that has been written by other people, it is likely that you will want to collect together operations, expressions and or statements into **functions** that perform the necessary computations for you

- **functions** are procedures or routines which do something with the data given to them (e.g. printf(), main()).
- note that some programming languages make a distinction between a **function**, which returns a value, and a **procedure**, which performs some operation but does not return a value
- we will return to the definition of **functions** shortly

- for many common mathematical computations, pre-written code is provided in files that are installed as standard with the C programming language
- those files are collectively known as the C libraries

- Pre-defined mathematical functions continued
  - More specifically, common mathematical functionality is provided through a set of 'functions' that are contained in the math library
  - A function is a named sequence of statements that performs a desired operation (such as calculate a square root or raise a number to a power)
  - We have already seen a C library function being used in the HelloWorld program
  - i.e. the printf() function is provided by a library accessed via stdio.h
  - (actually, we have seen two functions, the second is the main() function, which *is not* pre-defined in a C library - we have been writing that one ourselves)

- Before we can use C's pre-defined mathematical functions, we need to tell the compiler how to find them
  - e.g. a function which, when provided with a number as input, gives us the square root of that number as output
  - e.g. a function which, when provided with two numbers as input, gives us the first number raised to the power of the second number as output
- i.e. we need to tell the compiler that the location of these functions is provided in a file called math.h

- If we want to use those functions, we must explictly include math.h in our code

```
/* Print three floating point numbers */
#include <stdio.h>
#include <math.h>
main()
{
 do something;
  printf(the result of doing something);
}
```

- NOTE: When including math.h, you must also add -lm to your compile command
- i.e. when compiling your code, per the Interpreting, Compiling and Running section, above,
- use gcc ProgramName.c -lm
- (rather than simply gcc ProgramName.c)

- In order to use the printf() function, we used stdio.h to import it
- We then used the name of that function to ask for a computation to be carried out - printf()
- And provided input data for that command to be carried out on - "Hello World"

- in order to calculate the square root of a number, we need to follow a parallel path:
- include the file describing the location of the function that will be executed
- use the name of the function in question, which turns out to be sqrt()
- and provide data of an appropriate type as input to that command - in this case, a double e.g. 4.0

- in this case, we also need to take an extra step - work out what to do with the result of the calculation
- the sqrt() function returns us a double
- we can store that double in a variable, print it to screen or both

```c
/* program to calculate the square root of x*/
#include <math.h>
#include <stdio.h>

int main()
{

double x = 4.0;

double result = sqrt(x);
printf("The square root of %f is %f\n", x, result);
}
```

```
The square root of 4.000000 is 2.000000
```

- note the inclusion of two variables in the printf() statement

```c
/* program to calculate the yth power of x*/
#include <math.h>
#include <stdio.h>

int main()
{
double x = 4.0;
double y = 2.0;

double result = pow(x,y);
printf("%f to the power of %f is %f", x, y, result);
}
```

```
4.000000 to the power of 2.000000 is 16.000000
```

- note the inclusion of three variables in the printf() statement

```
The C library function
double sqrt(double x)
returns the square root of x

The C library function
double pow(double x, double y)
returns x raised to the power of y
```

General form:

```
The C library function
returnedType functionName(input data and types)
returns the result
        of performing specified computation
        on the input data
```

- Just like mathematical operators, C functions can be combined within complex expressions
  - 1+sqrt(4);
  - 1+pow(3,2);
- And included directly in print statements
  - printf(" The square root of 4 is %f ", sqrt(4));
  - printf(" The square root of x is %f ", sqrt(result));

- Note also that math.h provides values for some common mathematical constants including pi and e
- Once you have #included math.h in your C program, you can refer to those mathematical constants using M_PI and M_E

```
#include <stdio.h>
#include <math.h>
main()
{
 double x=M_PI;
 double y=M_E;
 printf("PI plus one equals %f, e= %f", x+1, y);
}
```

```
PI plus one equals 4.141593, e= 2.718282
```

Errors and Debugging

- As you develop as programers and computer scientists (a key aim of this course), we will encourage you to draw together approaches and techniques that you might expect to find in other disciplines e.g.:
  - Mathematics
    - ★ Use formal languages to denote ideas (specifically computations)
  - Engineering
    - ★ Design
    - ★ Assemble components into systems
    - ★ Evaluate trade-offs among alternatives
  - Natural Science
    - ★ Observe behaviour of complex systems
    - ★ Form hypotheses
    - ★ Test predictions

- The single most important skill for a computer scientist is problem solving
  - Working out how to formulate problems
  - Thinking creatively about solutions
  - Expressing solutions clearly and accurately
- Learning to program is learning to solve problems

- A problem that you will encounter frequently is dealing with programming errors
- The results of those errors are more commonly known as called bugs.
- The process of tracking them down and correcting them is called debugging.
- With problem solving, decomposition, debugging is one key skills for successful programers
- Bugs are part of the programers life, you cannot escape them, you just have to find them and solve them
- Practise makes perfect

- Syntax Errors
  - The easiest type of error to identify (and, arguably, to make)
  - C (the language we will be using at the start of this course) cannot execute a program unless it is syntactically correct.
  - You have the compiler to find them and tell you what is wrong
  - It returns an error message without starting the program.
  - Syntax refers to the structure of your program and rules about that structure.

- Example Syntax Errors
  - In English, a sentence must begin with a capital letter and end with a single full stop.
  - this sentence contains a syntax error.
  - So does this one
  - A C example with mismatched parenthesis:
    - (5 /2))

- Run Time Errors
  - Do not occur until the program is run and the particular faulty line is executed.
  - Called exceptions because something exceptional (and usually bad) has happened.
  - Execution of your program is normally terminated at this point.
  - Information about the current state of the program is printed.
  - Sometimes hard to fix, as they could be input or hardware dependent.
  - Your code should take this into account. This is called robustness

- Example Run Time Errors
  - An example when executing a Java program is a division-by-zero error:
  - $5/0$

- Logical Errors
  - The program runs successfully: no error messages ...
  - ... but it does not do what you wanted it to do ...
  - ... instead it does what you told it to do.
  - Debugging logical errors requires you to work backwards from the observed output (if any) to determine what the program is actually doing internally.

- Experimental Debugging
  - Debugging is one of the most important skills you will acquire in the unit
  - Many programers believe debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
  - Debugging is a form of detective work:
    - ★ You are confronted with clues, and you have to infer the processes and events that lead to the results you see.
  - Debugging can also be seen as an experimental science:
    - ★ Form hypothesis about cause of error.
    - ★ Modify program, and predict new outcome.
    - ★ If results match prediction, hypothesis is correct.
    - ★ Otherwise, modify hypothesis.

- *"When you have eliminated the impossible, whatever remains, however improbable, must be the truth" (Sherlock Holmes, from A. Conan Doyle's The Sign of Four)*

- Experimental Programming and Debugging
  - At some point, programming and debugging converge:
  - Programming can be seen as the process of debugging a program until it does what you want.

- Experimental Programming and Debugging
  - Start with a working program that does something.
  - Make small modifications, debugging them as you go.
  - Always have a working program.
  - Test-Driven Development and Agile Programming are examples of techniques that use this approach
  - Linux is an example of the output that you can achieve with this kind of approach
  - Linus Torvalds started out writing a program to explore the Intel 80386 chip.
  - It simply switched between printing AAAA and BBBB . . . . and evolved to Linux!

- While this week's concepts are fresh in your minds (first programs, functions, data, types and errors),
- We would like you to try a small debugging exercise.
- The code on the following slide does not work as intended
- Work in small groups (approx 3 people) to
  - identify and fix the errors
  - decide which kind of error has occured in each case (syntax, runtime or logical)
- These lecture notes are now Moodle if you would like to use them
- And you may use the ide introduced this morning to run the code and get clues about what is going on

```c
/*this code, when run,
* doesnt give the expected output
* The output should be PI plus c equals 5.141593
* i.e. the actual result of adding c to PI
* Please find and repair the errors
* So that the program runs as expected
* (It might be useful to categorise the errors)
*/
#include <stdio.h>
main()
{
 int a=7;
 float b= 32.0
 a=62;
 b=17.0;
 printf("PI plus c equals %d", M_PI+b);
 double c=2.0;
}
```

Next Steps: Coursework and Assessment

- On Thursday each week, We will put up the coursework exercises that you need complete and hand on the Monday, eleven days later
- The first set of lab exercises will go up on Thursday October 6th
- and should be handed in at 5pm on October 17th
- Each set of exercises is accompanied by a detailed set of instructions
- The relationship between those exercises and your overall mark for this course is described on the following slides

Coursework and Assessment

- The formal assessment of this course comes from a combination of
  - Coursework (50%)
    - ★ Lab Exercises
    - ★ Larger Coursework
  - A final exam (50%)
- In order to pass this unit, you will need to obtain a pass mark (40% or more) in both coursework and exam

How to succeed: Time management

- Dont forge that strong time management can make this course feel considerably easier
  - See the Bath Student Union website (url down when we wrote these slides)
  - http://counseling.uchicago.edu/page/asap-time-management

Assessment

- For CM10227, the unit's coursework requirements (50% of your total marks) can completed in one of two ways:
  - option a):
    - ★ 6 lab exercises
    - ★ plus two larger courseworks
  - option b):
    - ★ 2 lab exercises (the first two)
    - ★ plus an advanced programming lab exercise
    - ★ plus two larger courseworks (as above)

Assessment

- For CM50258, you have fewer choices:
  - 4 lab exercises
  - one larger coursework
  - plus one code analysis
- Note, however, that you can attempt the other other CM10227 exercises
- and we will give you indicative marks and feedback for each one
- though the additional marks will not count towards your grade for the course

Lab Exercises

- Each of the lab sheets will correspond to one or two lectures
- Feedback on your solutions can be obtained from the lab tutors
- You are strongly encouraged to attempt all labs

Lab Exercises

- Lab exercises will be marked as follows:
- Weekly lab sheets
  - 60%-100%: 10 marks
  - 40-59%: 5 marks
  - 0-39%: 0 marks
- Advanced programming exercise
  - correctly working 25 marks
  - design and program structure: 15 marks

Late Submission

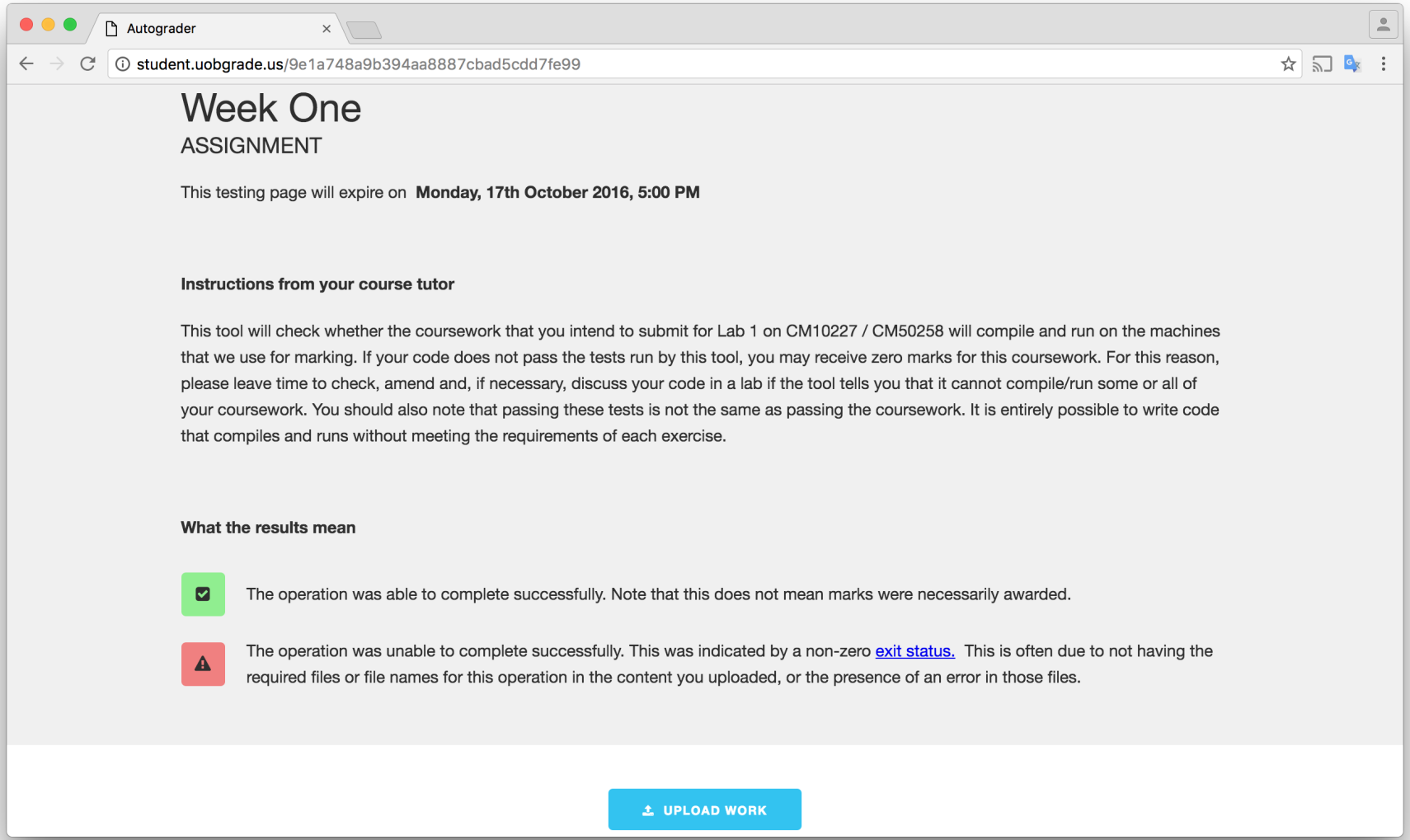
- 1 minute late: 40 percent max
- 5 days late: 0 percent max
- Extensions
  - can only be given by the Director of Studies (Fabio Nemetz)
  - will only be given for special circumstances

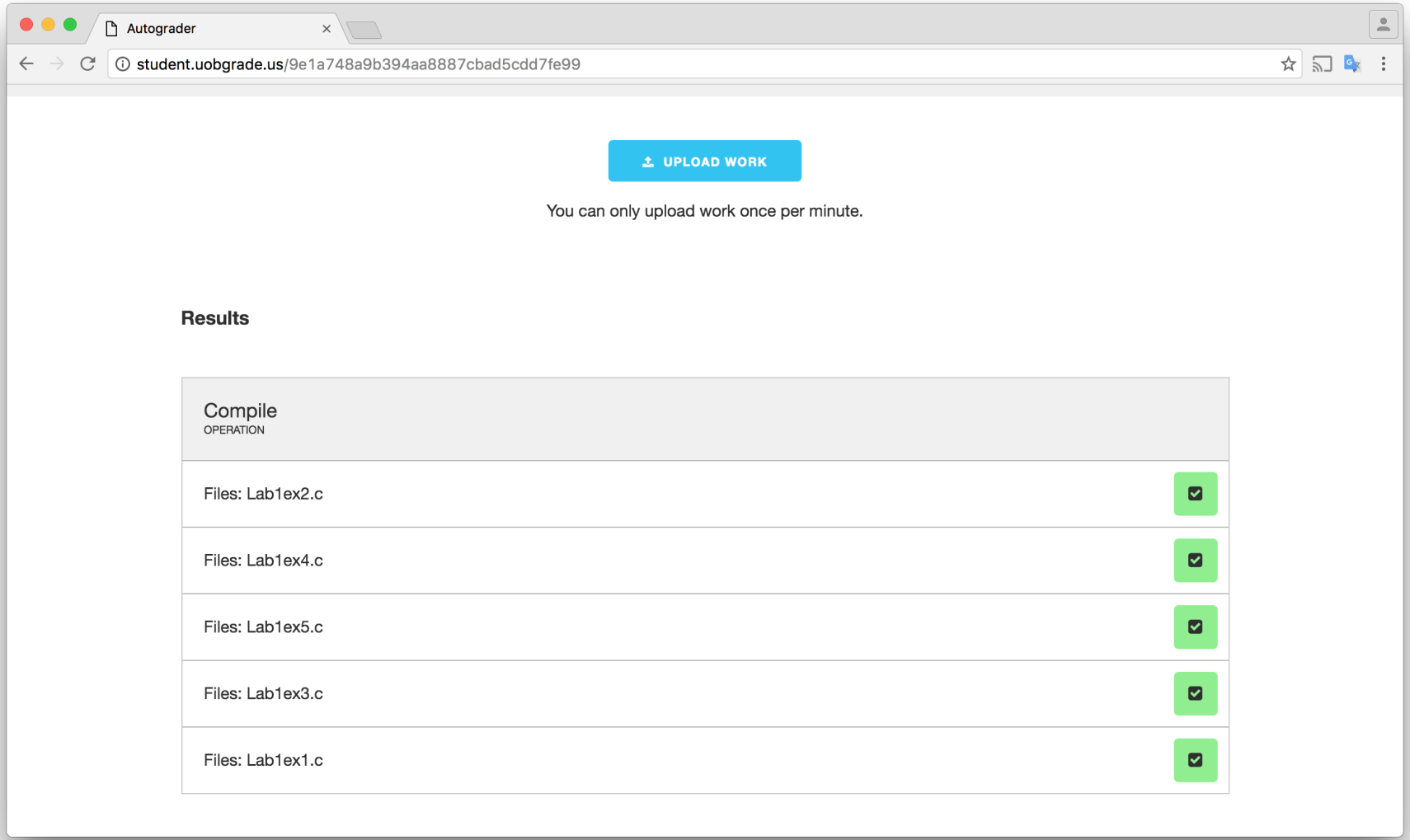
Marking

- We use a semi-automated marking tool when grading coursework
- The tool generates some mark automatically
- We then add marks for the quality and style of your code "by hand"
- Comments are also written "by hand"

Marking

- For the first time this year, we have also opened a "student-facing" part of that tool
- Which will allow you to check that your code compiles and run before you submit it
- (It won't, however, tell you the mark that you will recieve for that code)
- We will put links to that part of the tool on Moodle with each lab sheet
- (Note that the tool definitely works in Google Chrome. Performance in other browsers may vary)


Autograder
student.uobgrade.us/9e1a748a9b394aa8887cbad5cdd7fe99
Week One
ASSIGNMENT
This testing page will expire on Monday, 17th October 2016, 5:00 PM
Instructions from your course tutor
This tool will check whether the coursework that you intend to submit for Lab 1 on CM10227 / CM50258 will compile and run on the machines that we use for marking. If your code does not pass the tests run by this tool, you may receive zero marks for this coursework. For this reason, please leave time to check, amend and, if necessary, discuss your code in a lab if the tool tells you that it cannot compile/run some or all of your coursework. You should also note that passing these tests is not the same as passing the coursework. It is entirely possible to write code that compiles and runs without meeting the requirements of each exercise.
What the results mean
The operation was able to complete successfully. Note that this does not mean marks were necessarily awarded.
The operation was unable to complete successfully. This was indicated by a non-zero exit status. This is often due to not having the required files or file names for this operation in the content you uploaded, or the presence of an error in those files.
UPLOAD WORK


Autograder
student.uobgrade.us/9e1a748a9b394aa8887cbad5cdd7fe99
UPLOAD WORK
You can only upload work once per minute.
Results
Compile
OPERATION
Files: Lab1ex2.c
Files: Lab1ex4.c
Files: Lab1ex5.c
Files: Lab1ex3.c
Files: Lab1ex1.c

Feedback

- Formative: tutors during labs (please ask!)
- Lab sheets
  - some realtime feedback via the marking tool
  - individual feedback via Moodle
- Larger Courseworks:
  - detailed individual feedback via Moodle
  - general feedback on Moodle
  - tutors and lecturer (please ask!)
- Exam: No individual feedback allowed. Marks will be provided via Moodle.

Exam

- In the exam, you will be asked to answer 3 questions out of 5 on
  - general programming principles
  - specifics of the C and Java languages
- You may also be asked to
  - write small programs
  - debug small programs
  - explain the output of small programs

Questions?