Recorded Lecture Errata: “Introduction to Programming”, *ELE00029C*

Prepared on 9th March 2023. Edited by Oliver Dixon <od641@york.ac.uk>.

Editor’s Notes. These notes were collated for the purposes of identifying serious errors and recommendations of poor practice in the *ELE00029C* “Introduction to Programming” lecture series. The Editor hereby places the entire errata in the Public Domain; readers with an appropriate academic interest are encouraged to improve and disseminate this document through relevant channels. This PDF is available on-line at `https://www-users.york.ac.uk/~od641/29c/export.pdf`. The LaTeX source archive is available under `errata-source.tar.gz`, in the same HTTP web directory. The Editor strongly encourages all comments regarding this document to be directed, in plain text, to the e-mail address displayed above. General remarks concerning the *ELE00029C* module are also warmly welcomed; such enquiries may also be sent to the first-year Electronic Engineering course representatives: Thomas Mason <tm1451@york.ac.uk> and Adam Gottesmann <ag1676@york.ac.uk>.

General Notes. In both cases, the relevant videos were uploaded in the first quarter of 2021, and reuploaded in the first quarter of 2022. Although the video files are not byte-equivalent, the content is essentially identical, with minor sections of audio being silenced, largely excluding points at which the Lecturer mentioned any year-dependent information such as introductory speeches and details of submission due-dates. It is also noted that these forty-minute videos are the only recorded lectures visible to students, from a course supposedly lasting nine weeks. The given timestamps correspond to the reuploaded videos.

Lecture V: “Introduction to C”

`https://www.youtube.com/watch?v=vNcKEJjf6B8` (Published 17th January 2021)
`https://www.youtube.com/watch?v=xQC6CjDfxoQ` (Reuploaded 27th January 2022)

1. {`4:41`} C is a general-purpose language designed for processors, not field-programmable gate arrays. Although C-to-FPGA compilers do exist, demonstrating the dominance of C with the example of an FPGA is nothing short of ridiculous. These compilers often create discrete and crude state machines, rendering C as a very inefficient development tool for such applications. Verilog and VHDL are used to describe logic circuits; C is used to describe programs for execution on traditional microprocessors.

2. {`4:43`} It is explicitly claimed that C is “fast”, but no comparison or reference point is provided. A poor programmer or compiler may render C to be significantly slower than similar compiled counterparts, such as Go or Rust. At this point, the Lecturer should have stated the performance benefit of C, on average, in relation to other well-defined benchmarks, as was done on the fifth page of the first ShockSoc Programming Support Session laboratory script.

3. {`5:47`} It is correctly stated that C [binaries] have a “relatively small runtime image”, however this terminology is not explained, and will be completely alien to the majority of the cohort.

4. {`6:31`} C is not a low-level language, as is suggested by the Lecturer. By the industry-standard categorisation of programming languages, it is a third-generation language, in the same category of abstraction as Python or Java. It is also not possible to “get as close to the machine as you want”. By their very nature, portable languages are generally incapable of explicitly exploiting many processor-specific features and addresses, since such access would imply processor-dependency. Efficiently mapping high-level C constructs to native instructions is the primary purpose of any compiler.

5. {`9:26`} The concept of a C "data structures" is not well-defined. Whilst C primitive types tend to be statically allocated within thread data and stack frames, it is incorrect to imply that C is incapable of dynamically allocated data, just as it is incorrect to claim that Python is incapable of storing statically allocated data. The term "less efficient" is used throughout, but this is similarly poorly defined. Less time-efficient? Storage-efficient?

6. {`10:26`} It is incorrect to claim that a language is bound to a development environment. At this point, the Lecturer implies that the Code::Blocks IDE is standard for C development, whilst the IDLE suite is standard for Python development. This is wholly untrue, and these aforementioned environments are never used in industry. No mention of an abstracted build chain is mentioned, as seen in the ShockSoc Programming Support Session lab scripts, and there is a strong resultant implication of coupling between the programming languages and IDEs. This has confused some students in the cohort, many of whom seemed surprised to witness code being written in plain text editors with detached compiler, linker, and debugger arrangements.

7. {`11:06`} The statement examples are unhelpful. The figure (an uncaptioned pixillated raster) suggests that a conditional expression is part of a "statement", since the `if` keyword is shown on the same line as the `y = 3;` assignment operation. Furthermore, the example of `i = i + 1;` contradicts the third bullet point, supposedly providing a definition for "statement", as a smaller expression can be derived that is semantically equivalent: `i++;`.

8. {`12:06`} The Lecturer's remarks concerning variable addresses are confusing, as they seem to be haphazardly interchanged with the definition of a pointer, the concept of which will not be introduced for many weeks. This slide also contains a very poor-quality image representing a partial source code listing, again lacking a caption. At this stage, symbols and symbolic referencing should be introduced in an abstract sense, to obviate confusion between the interrelated notions of run-time symbol addresses, compile-time symbol addresses, symbol debugging names, and the real symbol values.

9. {`12:45`} The term "pre-defined words" is used on the slides and during the delivery, as opposed to the standard "keywords" term. Stating that a symbol name has been pre-defined is equivocal, as it does not specify the stage at which the reference was established. In particular, the wording of "pre-defined" could refer to keywords, compiler extensions, or compile-time constants, amongst (many) others.

10. {`14:35`} The example comment is poorly aligned and of dreadful image quality. The single-line comment, in which a comment begins with a double-oblique, is not mentioned. Nor is it mentioned that the exemplar syntax may be used to create a comment spanning multiple lines, and may not be nested.

11. {`14:53`} The code example makes the common mistake of including the `void` keyword within the parentheses of a function implementation, for a function taking no arguments. Whilst valid for function signatures, this should be generally omitted in modern code. The Lecturer continues to claim that "C programs must have a `main` function", which is untrue. Only standalone executable binaries (as opposed to shared object binaries intended to be dynamically linked) must have an *entry point*, but this does not have to be identified as `main`. This may be of an arbitrary name, set accordingly with linker options.

12. {`15:29`} For some unknown reason, all example variable declarations are of the same `int` type, and none are initialised. Other types are mentioned in the main body of the slideshow, typeset in a variable-width font, but none of these are demonstrated in the example code. The Lecturer also asserts that "C does not allow you to [port over one type to another]", but given the existence and ubiquity of typecasting, this is another inaccurate statement. The concepts around explicit typecasting are not raised at any point during examined presentations.

13. {`16:01`} This entire slide appears to have been designed without the aid of a monitor, or keyboard, or mouse. Unhelpful and confusing parallels to Python continue to be drawn, as opposed to simply stating that “C follows standard arithmetic precedence”. The caveats and nuances around integer and floating point division are completely skipped.

14. {`18:36`} The Lecturer begins discussing conditionals, despite having not completed describing all classes of operators. Bitwise operators (conjunction, disjunction, exclusive disjunction, inversion, and shifting) are not mentioned at any point. The modulus operator is also ignored. This slide continues to inexhaustively list various conditional statements, bypassing the conditional ternary operator and its various uses in variable assignment and instantiation. Pointless analogues to Python syntax continue to be made.

15. {`19:36`} The conditional nesting example consists of poor structure, and many compilers will issue warnings due to the potential ambiguity induced from nesting `else` statements without explicit braces to unequivocally define the scope of each execution block. Additionally, the flow of execution does not make any sense. The first nested conditional could be intuitively achieved with the logical conjunction operator introduced a few minutes prior, thus rendering this a very poor example of wise and minimal nesting. Also, the reasoning supporting the area of a circle becoming $A = r^2$ as opposed to $A = \pi r^2$, when $5 < r \leq 10$, remains unexplained.

16. {`20:28`} Despite claiming otherwise, the Lecturer does not provide an example of conditionals explicitly using braces until the next slide. The optional rationale for omitting braces in cases of single-statement execution blocks is not explained at all.

17. {`21:06`} The explanation of the `switch-case` construct is incomplete. The Lecturer does not explicitly state the restriction of `switch` conditionals to integer-like types, nor does he explain the run-time differences to nested `if-else` chains. In a live lecture, when a student asked whether there was such a difference, the Lecturer claimed there was none, while reiterating the incorrect assertion that `switch` statements act purely as syntactical sugar.

18. {`21:28`} The Lecturer states that “each `case` [label] will contain a `break` command”. This is not true, and many classical programming constructs, such as Duff’s Device, are built on the possibility of fall-through `case` statements. Furthermore, the provided example is poor, as it does not exemplify a real application of a `switch`. The following listing demonstrates the manner in which a sensible programmer would implement the same algorithm.

```c
const char * const n_strs [ ] = { "one", "two", "three",
        "four", "five", "six", "seven", "eight", "nine" };

/* Assuming 'number_entered' is a previously defined integer... */
puts ( ( number_entered > 0 && number_entered < 10 )
                ? n_strs [ number_entered ]
                : "Number is not between one and nine." );
```

The Lecturer’s example also omits a final newline character in the final `printf` function call. Unless `stdout` is explicitly flushed, the text contained within the `default` block may not be displayed at all. (The `puts` function used in the listing above automatically appends a newline to the given constant string.)

19. {`23:38`} Again, the presented example is unhelpful and incomplete. Despite mentioning global declarations and “user-defined” functions (in which `main` is not included, for unknown reasons), these are not shown in the example. Only a single preprocessor statement and entry point is presented, and the content of `main` is unintuitive and syntactically superfluous.

20. {`28:36`} The Lecturer continually seems to confuse Twitter terms with standard English, repeatedly referring to the pound sign (#) as a “hash-tag”. The presentation slide also purports that C has many “in-built” functions, when in reality, the language itself has none, aside from a few notable pseudo-functions such as `sizeof` and GNU’s introspective `typeof`. He does not mention the notion of the standard library, nor the process through which implementations

thereof are linked with client programs. The role of header files, proving function and data signatures as opposed to executable code, is also left untouched.

21. {`29:20`} The ‘`\n`’ character is an ANSI escape sequence for the ASCII line feed (LF, #10) character, and is not a `printf`-specific placeholder, as is suggested. The list of format specifiers is also incomplete, neglecting formatters for displaying unsigned types and padding specifiers.
22. {`32:35`} The example code contains a serious bug: the `while` statement is terminated with a semicolon, and the following braces create an arbitrary scope unrelated to the loop. As the “`number >= 10`” will always evaluate true, this program will cause an infinite loop. The indentation is also misaligned.
23. {`33:22`} The claims that “an ampersand is specific to `scanf`” and “`scanf` likes to read into addresses” is confusing and unnecessary. Instead of describing the basic concepts of pointers and addressing, the Lecturer omits any level of detail and begins to describe the `return` statement without providing any description of *why* a function may need to return a value. In the particular case of the `main` entry point function, the Lecturer asserts that the return value is a “confirmatory code”, “but in this case does not really give us much”. This is nonsensical; program return status codes are an integral part of operating system scheduler structure and should not be dismissed in such a casual manner.
24. {`34:54`} At this point, the Lecturer presents a slide describing ANSI escape sequences and pointers, despite having (poorly) introduced them many minutes prior. Although the presented information is largely correct, it does not make pedagogical sense to supply such knowledge multiple slides after an incomplete and misleading introduction. ASCII/Unicode tables, to describe the notion of all characters existing as integer bytes, are not discussed.
25. {`36:11`} The `do-while` example is contains substandard formatting, with incorrect brace indentation and an incomplete syntax description for the condition; conditions should be surrounded in parentheses and terminated with a semicolon. A similar error was made some slides prior, in which the Lecturer purported the syntax of a standard `while` loop to be “`while` *condition* `do {...}`”, which is incorrect.
26. {`37:57`} Although the Lecturer correctly mentions the optional nature of the initialisation and conditional statements in the `for` loop syntax, he suggests that infinite loops cannot occur with `for` loops (“executes a block of statements a fixed number of times”), which is clearly incorrect. Although both of the examples are finite, the general description is partially flawed.

Lecture VI: “Functions, Arrays & Pointers in C”

`https://www.youtube.com/watch?v=E4saN7DWw4o` (Published 2nd February 2021)
`https://www.youtube.com/watch?v=QkzHohvY5Oo` (Reuploaded 31st January 2022)

1. {`2:20`} The provided raster figure is of atrocious quality and does not contribute in any form to the content of the slide. Again, code is set in a variable-width typeface, and the concept of relating Python functions to C functions has confused a large section of the cohort.
2. {`3:09`} The indentation of the sample code is inconsistent; some sections use double-space indentation, and others use quad-space. Further, the accompanying diagram doesn’t make sense, and it bears no resemblance to the sampled code being displayed. The `stdio.h` header file is included for some unknown reason, despite the sample having multiple basic C syntax violations.
3. {`4:40`} No attempt is made to explain the call stack, or stack frames, or thread data. The figure is of exceptionally low quality and does not correctly depict hierarchical modularisation; for a superior description and set of descriptive vector images, confer with the third Programming Support Session lab script, in which these concepts are correctly described in detail.

4. {`8:23`} The function definition is incomplete; no remarks concerning function qualifiers (such as `static` or `inline`) are made, and it is not clear that the various arguments can be of different types, as the placeholder is uniformly named `type`. The parentheses surrounding the return value are superfluous, aside from in the case of a `void` function, in which case the entire `return;` statement is unneeded.
5. {`9:00`} In a list of example C data-types, `void` is included. `void` is a keyword, and not a data-type. A void pointer (`void *`) may be valid, but this is not made clear during the presentation.
6. {`9:34`} The function prototype syntax is not correct; the terminating semicolon is omitted. The Lecturer does not explain that in prototypes, argument names are not necessary; members of the cohort seem consistently confused between function prototypes and implementations, and they do not understand that certain symbol names may be skipped in some cases.
7. {`10:05`} The image presents an inaccurate schema of a standard C program. To be consistent with previously described skeletons, “header files” should be generalised to “preprocessor statements”. This diagram also implies that non-`main` function implementations cannot appear before the entry point, which is not true; pre-implementation prototypes are optional, and are only required in cases where the function is being called prior to its initial definition.
8. {`12:19`} The `printf` call should be followed with an explicit `fflush(stdout)` call, since there is no terminating new-line character to implicitly flush the text to standard output. In rare cases, dictated by a race condition, the prompt text will not be visible before the `scanf` function begins to block the main thread. The example also does not follow the form described elsewhere, in the same slide; `void` is a not a type, and thus does not have an associated local symbol name. It is very atypical to encapsulate a simple expression, consisting of a single variable, within parentheses for the operand of a `return` statement.
9. {`13:50`} The Lecturer correctly states that the exemplar library functions are not formally “built-in”, however this contradicts the earlier (incorrect) assertation that C has “many built-in functions”. (This was originally stated at {`28:36`} during Lecture V.)
10. {`14:27`} It is incorrect to claim that POSIX `stdio.h` library functions are “often treated as standard by most programmers”. `printf`, and `scanf` even more so, are very rarely used in production code for large projects. The majority of students in this cohort will be developing programs for microcontrollers and other embedded systems, as electronic engineers tend to do, in which cases these functions are never available.
11. {`15:55`} This list is very strange; it is implied that functions can only take a maximum of two arguments, which is obviously untrue: “. . . functions with *both* arguments. . . ”.
12. {`16:26`} The code listing has a very large and distracting attribution notice, within which the Lecturer deemed it appropriate to include a full version history of a seven-line function. The `printf` calls are not properly terminated with a new-line character or `fflush(stdout)` call, there is a distinct lack of error-checking, and the indentation is nonsensical. Scope is explained poorly over a single line, and the hierarchical nature of scoping is not mentioned. For the function prototype of `checkEven`, but not the implementation, the parentheses should contain the `void` keyword.
13. {`19:25`} The Lecturer claims that true and false values are represented with one and zero respectively; this is not correct. True values are represented as non-zero, and false values are represented as zero. The examples in this slide are unreasonably complex compared to the immediately previous listings, and the Lecturer does not explain the concept of any expression involving a boolean relational operator being reduced to zero or non-zero at runtime.
14. {`20:09`} The second example purports to show a “random box” being drawn to the screen, but no random number-generation nor drawing/blitting code is listed. The example function, `projectile_collides_with_box`, would more accurately be described as implementing some collision-detection logic.

15. {`21:59`} The filing cabinet cartoon coupled with poor indentation and unclear colouring makes this slide exceptionally difficult to read. The final example is incorrect, in which the Lecturer attempts to compare the output of an array index in C and Python. The Python equivalent to the C `printf` function is `print`; the implicit printing is only performed by some IDLE-based debuggers, and certainly does not write anything to `stdout`. A sensible parallel to Python’s implicit printing is a C debugger’s inspection facility, such as the GDB `print` command, but this is obviously not mentioned.

16. {`24:08`} The `data_table` array initialisation line is terminated with a colon, as opposed to a semicolon. This is a syntax error, and would not compile. The amateurish term of “squiggly brackets” is repeatedly used, instead of the standard “braces”.

17. {`34:12`} In some (many) cases, an implicit cast will be made by the compiler between similar types for pointers. In this example, it would be generally acceptable to assign a character address to an integer pointer, as any subsequent dereferencing will be unaffected by the type width promotion. Although, the Lecturer is correct that “willy-nilly” typing is a generally poor idea.

18. {`35:39`} The example listing contains no indentation or formatting white-space. The memory addresses are printed in decimal, using the `printf` ‘`%d`’ specifier, as opposed to the correct ‘`%p`’. The latter formatting indicator prefixes values with ‘`0x`’ and prints subsequent values in hexadecimal, as is standard for address formatting. During a live Zoom lecture, the Lecturer reiterated that memory addresses should be printed with ‘`%d`’ or ‘`%u`’ specifiers, and seemed shocked and mildly confrontational when a viewer raised the existence of the suitable ‘`%p`’.

19. {`43:15`} The spacing throughout the example listing seems random and overly dense, making the entire slide difficult to read. The declaration and initialisation of the `temp` variable in the `swap` function is superfluously split across three lines, and there are two `printf` functions that will always create identical output.

. . .

This final point concludes the current errata, as of 9th March 2023. If more lectures are uploaded to YouTube in a similar fashion, this errata will be updated within a reasonable time frame. Please e-mail Oliver Dixon `<od641@york.ac.uk>` in the event of any questions or queries regarding this document or module.

Programming Support Sessions
Wednesdays, 2pm: P/T/401.
"The Instant Cure-All for ELE00029C!"