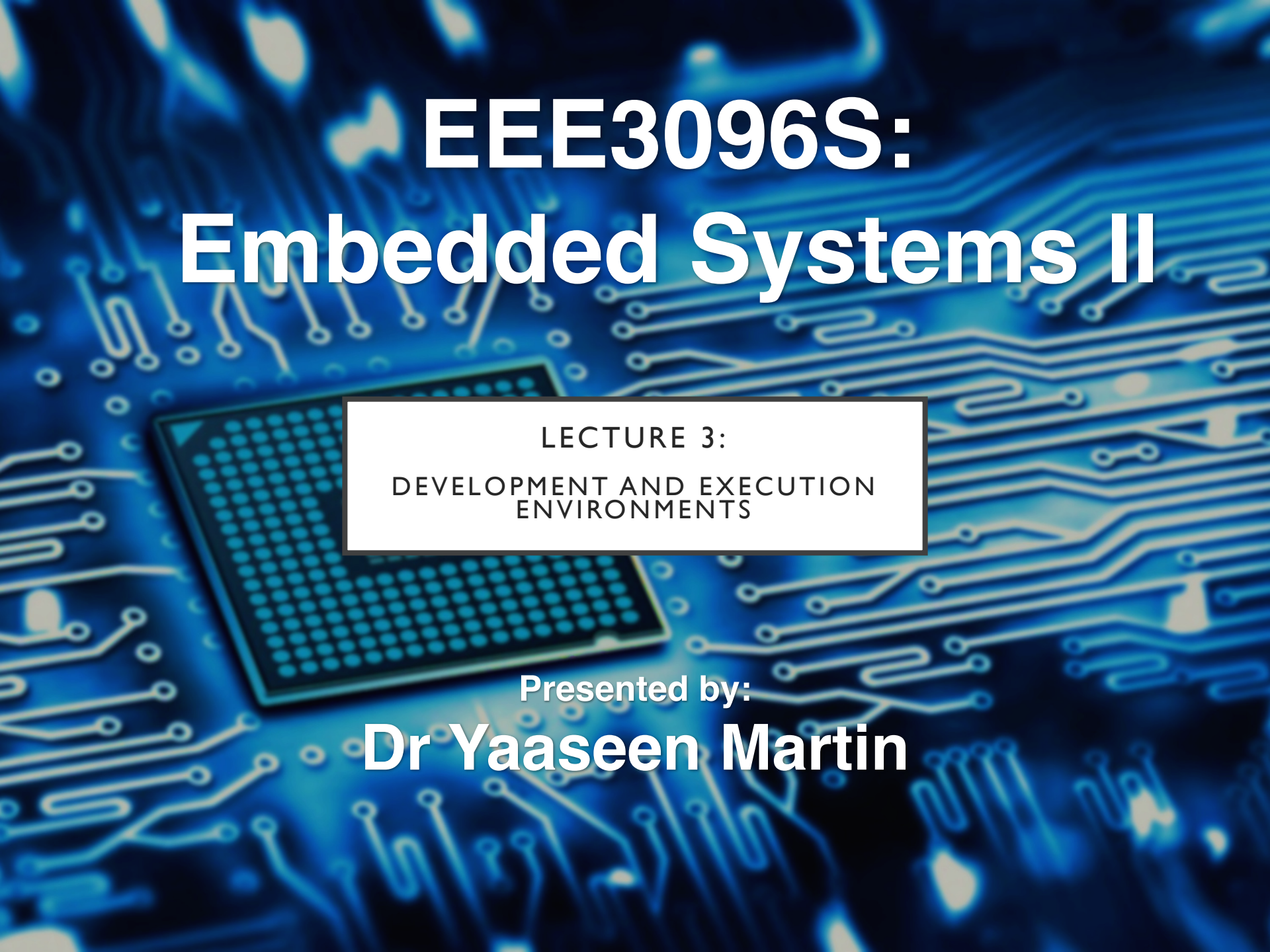


EEE3096S: Embedded Systems II

LECTURE 3:

DEVELOPMENT AND EXECUTION ENVIRONMENTS

Presented by:

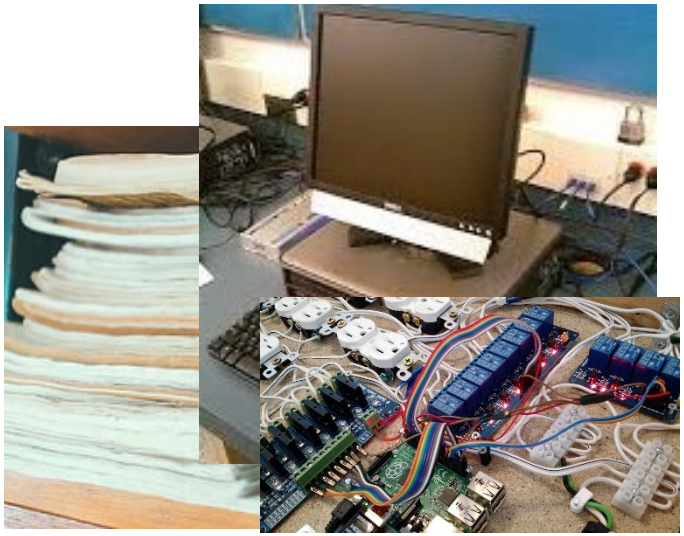
Dr Yaaseen Martin

THE DEVELOPMENT ENVIRONMENT

The **development environment** = the environment where development occurs

More specifically it comprises:

- Equipment used for development
- Tools used for development
  - e.g. Compilers, Linkers, etc.
- Other Resources
  - e.g. Manuals, Datasheets, etc.



THE PC DEV. ENVIRONMENT

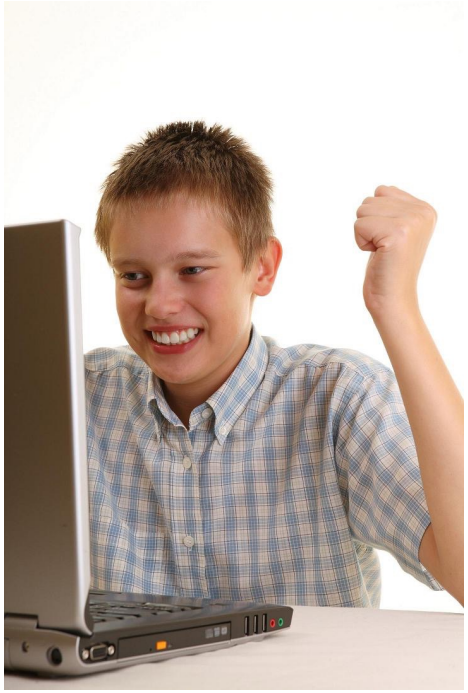
The PC application development environment is:

- Well defined,
- Uses standard tools

These tools depend on an OS, but not on a particular machine.

There are many benefits to this:

- Toolchain is fully configured and optimised
- Debugging is made easy with an IDE
- Full set of support libraries available
- Simply use the tools and things work (usually)



THE EMBEDDED DEV. ENVIRONMENT

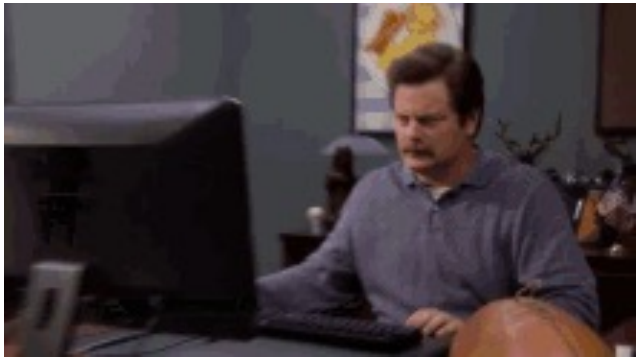
- The development environment for ES is generally not so well defined…
- Tools depend on the software vendor, the hardware vendor, peripherals used, etc., & limited combinations of these*
- Toolchain often built from scratch
- Debugging needs special techniques
- Few, if any, support libraries may be available
- Significant time often spent in preparing a custom development environment

*Why do I add this disclaimer? Because sometimes your selected tool supports only certain peripherals for a particular platform, or certain processors for a given compiler etc.

THE EMBEDDED DEV. ENVIRONMENT

Toolchains are often built from scratch

- Sure, many processor vendors provide a comprehensive set of tools… e.g. compilers, debuggers, etc…
- But often additional tools are needed. e.g. tools to
  - Programme memory devices (e.g. EEPROM programmer)
  - Configure peripherals (e.g. to configure radio frequency front-end)
  - Debug (may require special techniques or tools)
  - Some of these tools you may need to built yourself, some may be purchased or available as open-source solution)



RELEVANCE TO THE ENGINEER

- Compared to a PC developer, the embedded systems developer must:
  - Organise & maintain the development environment for the target platform
  - Know more about the development tools, which they need to configure and build
  - Know more about underlying hardware to configure the tools and write drivers
  - All above is also important during design and earlier steps (e.g., deciding partitioning decision)

PROCESSOR STANDARDISATION ISSUES

- Without standardisation…
  - Performance unpredictability
  - Difficulty in writing low level code
  - Difficulty in interfacing with peripherals
  - Difficulty in doing “the last little bit” of optimisation
- Standardised processor architectures and machine languages generally preferred (e.g. ARM cores and ARM / ARM Thumb instruction sets)

Execution Environment & Compiling Toolchain

ANSI C:

American National Standards Institute

C

WHAT IS C?

I know that you already know what it is and have used it …

… but have you thought why it is still so commonly used?

- C is a very popular, powerful and flexible programming language
- The “lowest common denominator” – C is extremely simple, without sacrificing flexibility.
- Some people refer to C as a “portable assembly language”. It is extremely close to the hardware.
- C offers a very low level of abstraction compared to Java, Python, etc.
- C++ is based on C and C++ had C compatibility as one of its main design goals.

THE HISTORY OF C

- C was developed at AT&T Bell labs in 1960s - 1970s, culminating in first main version in 1972.
  Led primarily by Dennis Ritchie.
- C was developed as a language to use to write Unix (a point many students don't know nowadays).
- C was intended to be a language for the development of *system software* (rather than *application software – which was novel in its time*).

Dennis Ritchie

The C Father

- For C to be useful for writing operating systems, C needed to be *high level* enough to be portable, while *low level* enough to get sufficient control over hardware…

  This is what makes it so useful for embedded software!

WHY ANSI (ISO) C?

- C offers the embedded systems programmer *just enough* abstraction.
- Enough abstraction to write portable code.
- Enough abstraction to make writing (and reading) code simple.
- *Not too much* abstraction, to allow efficient access to the hardware without burdensome wrappings.
- *Especially* as C is extremely popular and widely supported on nearly every micro-processor/controller.
- The C language is powerful, flexible and expressive.

ANSI = American National Standards Institute
ISO = International Organization for Standardization

WHY NOT [MUCH] C++?

- C++ has not seen as widespread use in embedded system development as has standard C.
- C++ contains a lot of baggage and language constructs which can cause bloat.
- C++ programs are (depending on style) less efficient than C programs.
- Most simply: C++ just doesn't have the same very wide compiler support for embedded system platforms.

This situation is changing. Compilers are improving, memories are getting larger, and C++ is gaining "mindshare".

C TUTORIALS (USEFUL RECAPPING)

- You already know C from ES1 and possibly C++ in comp. science courses.
- *However*, if you're one of the few students that haven't already done C, or you need a refresher, then here's two recommended online tutorials, which you can choose depending on whether you want the convenience of doing it online or if you prefer a stand-along option:

If you just want a quick refresher and don't want to spend time setting up tools or copying the code examples, then this is an excellent option:
https://www.learn-c.org/

OFFLINE OPTION

If you want to download things and do the exercises offline, then follow these tutorials (note that these are more thorough and go a bit deeper than the above online option):
https://www.cprogramming.com/tutorial/c/lesson1.html

EXECUTION ENVIRONMENT ('EE')

- The term 'execution environment' (EE) refers to those components that are used together with the application's code to make a complete computing system, including: the processors, networks, operating systems and so on.
- A programming language has some type of computer-based environment, called the 'runtime environment' (RTEnv) or 'runtime system' for which the resultant program is modelled to run in and to connect with.
- The runtime environment should, more accurately, be considered a large part of the 'execution environment'…
- In reality, especially for large systems where there are multiple languages and processes used (e.g. Python, C and CUDA) the execution environment may comprise multiple runtime environments – hence EE is a superset of RTEnv.

RUNTIME ENVIRONMENT (RTENV)

- This ‘runtime environment’ (RTEnv) or ‘runtime system’ addresses various issues, especially the aspects of:
  - How the program starts
  - How procedures/functions are called and results returned
  - Methods for passing parameters between procedures
  - How the program accesses or manipulates variables
  - Management of application memory
  - Ways to interface to the operating system
  - Inter-process communication (IPC) of the platform [if OS supports this]
  - How to connect to peripherals or to do I/O (note that an OS might not actually support direct control of I/O port control, but only provide higher-level functions like write a data type A to device X)
- The compiler makes assumptions depending on the specific runtime system to generate code for it.
- The RTEnv *usually* (not always!) has responsibility for setting up and managing the stack and heap, and may include features such as garbage collection, threads or other dynamic features provided by the language.
  (if you’re using a very basic compiler or assembler, essentially just converting asm code to instructions and allocating blocks of memory, you would need to explicitly implement stacks or heaps if your application needs them – you may do this regardless for processors with very limited memory)

PC VS EMBEDDED RTENV

- The RTEnv is well standardised for the PC
  - POSIX, Win32 and other API standards
  - Operating Systems have ABI* standards
- It's not so well defined on embedded systems
  - Often no OS to provide neat API standard
  - ABI depends on exact system configuration

*** application binary interface** (**ABI**) defines the low-level interface between programs or between a program and the operating system. An ABI standard defines sizes and structures of data types, endianness, system calls numbers or OS connection, calling convention, format of executable files, etc.
Example: Intel Binary Compatibility Standard (iBCS).
For more detail see: http://en.wikipedia.org/wiki/Application_binary_interface

THE GNU GCC AND BINUTILS TOOLS

THE COMPILING TOOLCHAIN

- A software compiling toolchain includes programs to:
  - convert source code into binary machine code
  - link these together
  - Separately assembled/compile code modules
  - Disassemble binaries
  - Convert formats
- The above are the fairly essential (minimal set) of tools needed. Usually additional tools are provided, such as: debugging support (possibly using ICE*), tools to program the platform, profiling tools, etc.

* ICE = In-circuit Emulator, allows you to replace the processor with a equivalent device that provides additional debugging support, e.g. pause processor, download registers, etc.

A CROSS-COMPILING TOOLCHAIN

Cross-compiler =
A compiler capable of creating executable code for a platform other than the one on which the compiler is running.

Common Examples:
ARM and PIC compilers that are run on a PC (x86 processor) for generating executables for embedded platforms.

Cross-compiler toolchain is thus the broader collection of tools used in developing software solutions for a platform.

Typically, for your development environment, you start with a baseline toolchain, e.g. arm-gcc, and add additional programs and other tools you need to develop your system.

GCC: GNU COMPILER COLLECTION

- This is the GNU project's compiler tool chain. They provide the full source for GCC and documentation explaining how to port it to other processor architectures. So, you could say, GNU's GCC project is a collection of cross-compilers that have a similar runtime environment for which the tools all follow a consistent user interface.

GCC TOOLS

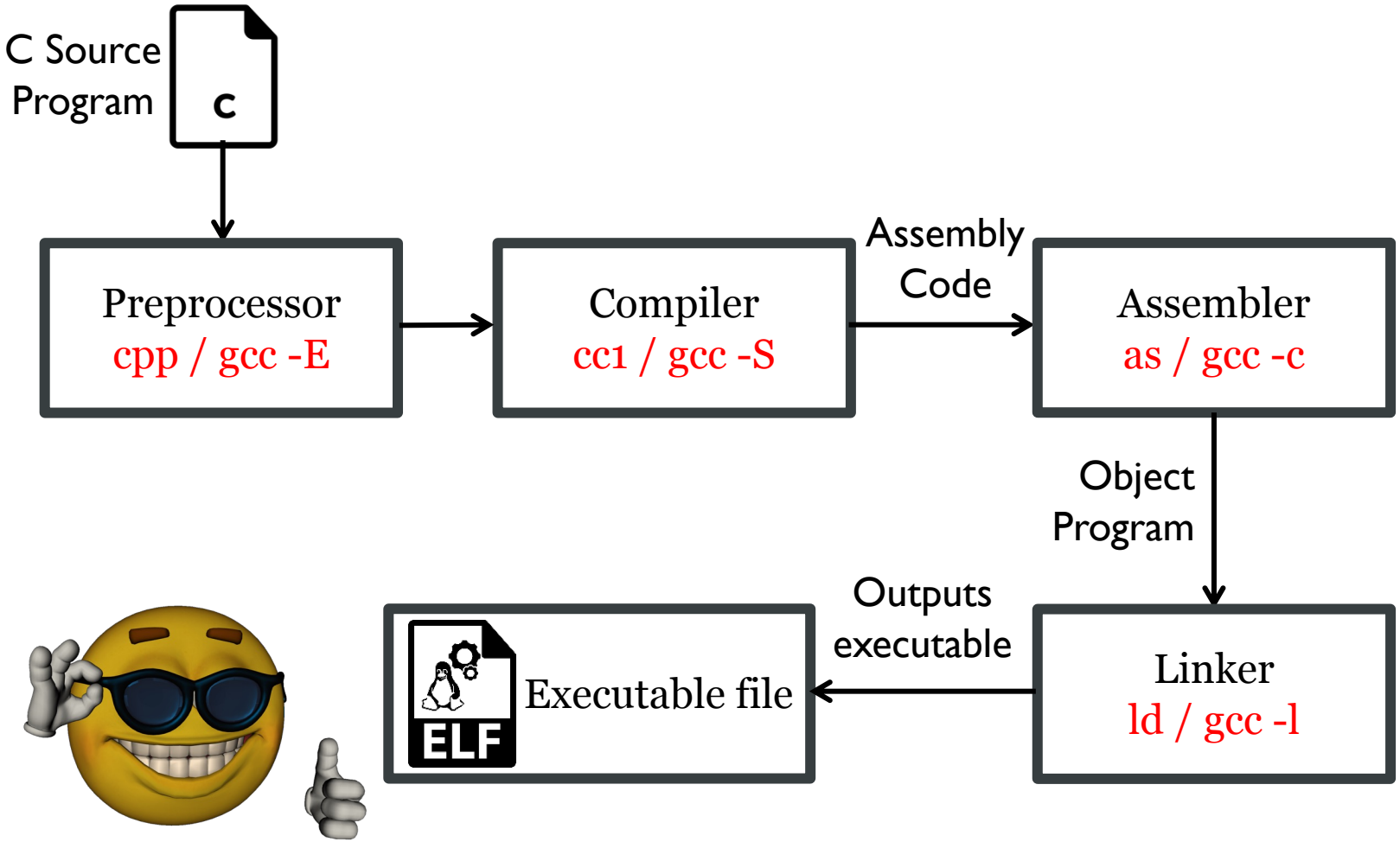
- cpp: c pre-processor for macros
- cc1: ‘c compiler phase 1’ perform semantic routines and translate C into Assembly language
- as: assembler to relocatable object files
- ld: linker
- To view the commands executed to run the stages of compilation (and other verbose output) try running:

```
gcc -v
```

GNU BINUTILS

- The GNU Binutils are a collection of binary tools commonly used with GCC.
- They are used for creating & managing:
  - binary executable programs
  - object files
  - Libraries
  - profile data, and
  - assembly code
- The most commonly used ones are:
  - ld : the GNU linker
  - as : the GNU assembler

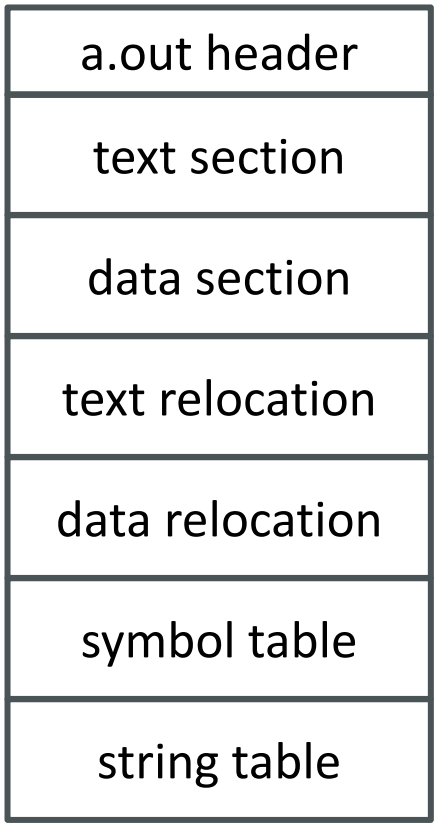
FROM C SOURCE TO EXECUTABLE



A.OUT EXECUTABLE FILE FORMAT

Unix a.out Format

- Hardware Memory Manager Unit (MMU) is needed.
- No relocation at load time (i.e. is a simple absolute format).
- Separate address space for code or instruction (I-space) and data (D-space), each only 64KB.



ELF EXECUTABLE FILE FORMAT

- Designed to support cross-compilation, dynamic linking and other modern system features.
- Three slightly different versions:
  - relocatable,
  - executable, and
  - shared object.
- Provides both:
  - A set of logical sections described by section tables. (for compilers, assemblers, and linkers)
  - A set of segments described by a program header table. (for executable loaders)

ELF FILE STRUCTURE