Welcome to CITS2002 Systems Programming

The unit explores the role of contemporary operating systems and their support for high-level programming languages, how they manage efficient access to computer hardware, and how a computer's resources may be accessed and controlled by the C programming language.

The unit will be presented by Dr Chris McDonald.

Our UWA Handbook entry

> *Understanding the relationship between a programming language and the contemporary operating systems on which it executes is central to developing many skills in Computer Science. This unit introduces the standard C programming language, on which many other programming languages and systems are based, through a study of core operating system services including processes, input and output, memory management, and file systems.*
>
> *The C language is introduced through discussions on basic topics like data types, variables, expressions, control structures, scoping rules, functions and parameter passing. More advanced topics like C's run-time environment, system calls, dynamic memory allocation, and pointers are presented in the context of operating system services related to process execution, memory management and file systems. The importance of process scheduling, memory management and interprocess communication in modern operating systems is discussed in the context of operating system support for multiprogramming. Laboratory and tutorial work place a strong focus on the practical application of fundamental programming concepts, with examples designed to compare and contrast many key features of contemporary operating systems.*
>
> **Prerequisite:** CITS1401 Computational Thinking with Python, or CITS2401 Computer Analysis and Visualisation (this unit is not suitable for first-time programmers).

Topics to be covered in CITS2002 Systems Programming

It's important to know where we're heading, so here's a list of topics that we'll be covering:

- **An introduction to the ISO-C programming language**
  The structure of a C program, basic datatypes and variables, compiling and linking.
  We will focus on the C11 language standard.

- **An introduction to Operating Systems**
  A brief history of operating systems, the role of contemporary operating systems, the relationship between programming languages, programs, and operating systems.

- **An overview of computer hardware components**
  The processor and its registers, the memory hierarchy, input and output (I/O) and storage components.

- **C programs in greater detail**
  Arrays and character strings, user-defined types and structures, how the computer hardware represents data, functions, parameter passing and return values.

- **Executing and controlling processes**
  Creating and terminating processes, a program's runtime environment, command-line arguments, accessing operating system services from C.

- **Managing memory**
  Allocating physical memory to processes, sharing memory between multiple processes, allocating and managing memory in C programs.

- **Files and their use in programs**
  The file management system, file allocation methods, file and directory operations and attributes, file input and output (I/O), raw and formatted I/O, unbuffered and buffered I/O functions.

By the end of this unit you'll have this knowledge - it just won't all be presented strictly in this order.

Here is our unit's schedule.

Systems-focussed Standards

In this unit we'll introduce a number of *standards* relevant to systems programming. Formal standards are used to define nearly all aspects of computing, notably data-representations, file-formats, programming-languages, networking protocols, web (communication) interfaces, and encryption and authentication.

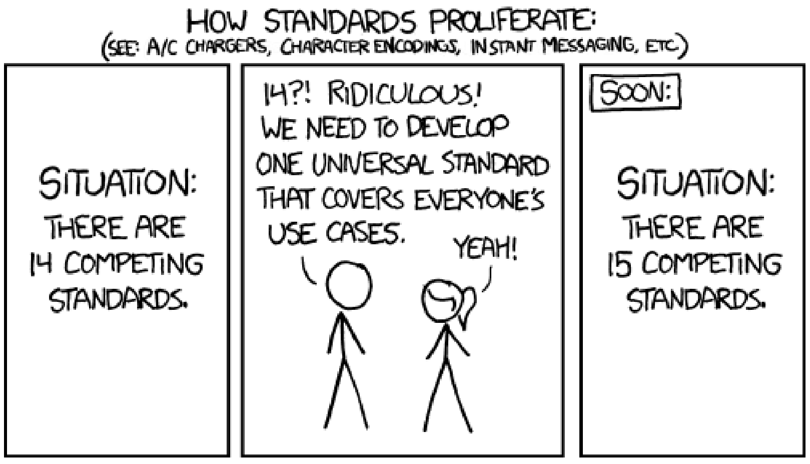
Formal standards in computing are often *very* large. For example, the formal standard for the C11 programming language (used in this unit) is 660 pages. You are not expected to understand these standards in depth (they will not be examined), but as part of professional development you're encouraged to skim them for an appreciation of their role in computing.

Standards discussed in this unit

- **C11** - the ISO/IEC 9899:2011 programming language standard standardizes a set of features supported by common contemporary compilers, such as *gcc* and *clang*. In this unit we focus on C11, despite it being superseded by C17 (standard ISO/IEC 9899:2018), because C11 is widely supported in the computing environments you'll use (and C17 is not yet widely supported).

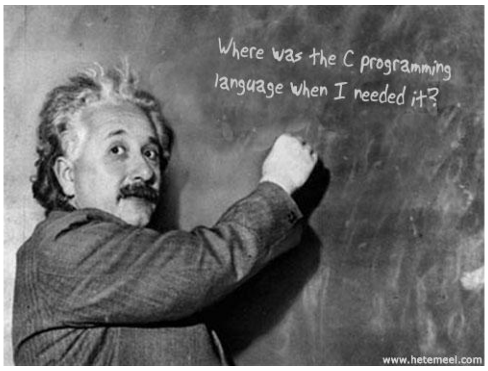
- **POSIX** - the Portable Operating System Interface is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the both system- and user-level application programming interfaces (API), along with command line shells and utility interfaces [Wikipedia].

  While POSIX is often associated with open-source systems (such as Linux), the first POSIX-certified system was Microsoft's Windows-NT v3.5 in 1999!


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Why teach C?

Since its beginnings in early **1973**, the C programming language has evolved to become one of the world's most popular, and widely deployed programming languages. The language has undergone extensive formal standardization to produce the ANSI-C standard in 1989, the **ISO-C99** standard in 1999, ISO-C11 (revision) in **Dec 2011**, and ISO-C18 in **June 2018** (which introduces no new language features, only technical corrections and clarifications to defects in C11).



C is the programming language of choice for most systems-level, engineering, and scientific programming:

- most of the world's popular operating systems, Linux, Windows and macOS, their interfaces and file-systems, are written in C,
- the infrastructure of the Internet, including most of its networking protocols, web servers, and email systems, are written in C,
- software libraries providing graphical interfaces and tools, and efficient numerical, statistical, encryption, and compression algorithms, are written in C,
- the software for most embedded devices, including those in cars, aircraft, robots, smart appliances, sensors, mobile phones, and game consoles, is written in C,
- the software on the Mars Phoenix Lander was written in C,
- much of the safety-critical software on the F-35 joint strike fighter, is written in C, but
- C was not used on the Apollo-11 mission!

Is C still relevant? [refs: Tiobe index, Please stop citing TIOBE].
(The Tiobe survey is based on search-engine queries - is not about the *best* programming language or the language in which *most lines of code* have been written).

Of note, in July 2023, the Tiobe survey rates C, Python, and Java as almost identical in 'popularity' (whatever that means). Of course, popularity is a poor measure of quality - otherwise, McDonald's Restaurants would receive Michelin stars.

So, we'll not focus on popularity, but on the relevance and appropriate uses of C.

Other interesting surveys:

- Stackoverflow's Developer Survey Results May 2023.
- Jetbrains' The State of Developer Ecosystem in 2023
- HackerRank's 2024 HackerRank Developer Skills Report

Other Systems Programming Languages?

A (limited) number other programing languages are used for contemporary systems programming, notably *Go*, *Nim*, *Rust*, *Swift*, and *Zig*. All have been strongly influenced by C, and attempt to address shortcomings of C.

In particular, Rust was officially added to the Linux kernel in December 2022

So what is C?

> *A programming language that doesn't affect the way you think about programming isn't worth knowing.*
>
> *— Alan Perlis, 1st Turing Award winner*

In one breath, C is often described as a good general purpose language, an excellent systems programming language, and just a glorified assembly language. So how can it be all three?

C can be correctly described as a general purpose programming language -
a description also given to Java, Python, Visual-Basic, C++, and C#.

C is a *procedural* programming language, not an *object-oriented* language like Java, (parts of) Python, Objective-C, or C#.

C is more *mow(the_lawn);* than *lawn.mow_thyself();*

C programs can be "good" programs, if they are:

- well designed,
- clearly written,
- written for portability,
- well documented,
- use high level programming practices, and
- well tested.

Of course, the above properties are independent of C, and are offered by many high level languages.

- C has programming features provided by most procedural programming languages - strongly typed variables, constants, standard (or *base*) datatypes, enumerated types, user-defined types, aggregate structures, standard control flow, recursion, and program modularization.
- C does not offer tuples or sets, Java's concept of classes or objects, nested functions, subrange types, and has only recently added a Boolean datatype.
- C does have, however, separate compilation, conditional compilation, bitwise operators, pointer arithmetic, and language independent input and output.

An important note: C and C++ are very different languages, with some common syntax and semantics. Webpages, blogs, career-positions, and even textbooks that promote "C/C++" as a single language, do not know what they are talking about.

A Systems Programming Language

C is frequently, and correctly, described as an excellent *systems programming language*.

C also provides an excellent operating system interface through its well defined, hardware and operating system independent, *standard library*.

The C language began its development in 1972, as a programming language in which to re-write significant portions on the Unix operating system:

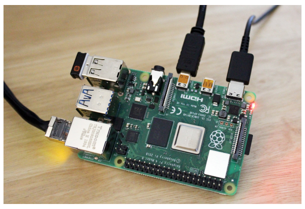
- Unix was first written in assembly languages for PDP-7 and PDP-11 computers.
- In 1973 Dennis Ritchie was working on a programming language for operating system development. Basing his ideas upon BCPL, he developed B and finally created one called C.
  (Yes, there is a language named 'D', but it's not a descendant of C)
- By the end of 1973, the UNIX kernel was 85% written in C which enabled it to be ported to other machines for which a C compiler could be fashioned.
- This was a great step because it no longer tied the operating system to the PDP-7 as it would have been if it remained in assembly language. In 1976 Dennis Ritchie and Stephen Johnston ported Unix to an Interdata 8/32 machine. Since then, Unix and Linux have been ported to over 260 different processor architectures.

Today, well in excess of 95% of the Unix, Linux, macOS, and Windows operating system kernels and their standard library routines are all written in the C programming language - it's extremely difficult to find an operating system *not* written in either C or its descendants C++ or Objective-C.

Portability on different architectures

C compilers have been both developed and ported to a large number and type of computer architectures:

- from 4-bit and 8-bit microcontrollers,
- through traditional 16-, 32-, and 64-bit virtual memory architectures in most PCs and workstations,
- to larger 64- and 128-bit supercomputers.



Compilers have been developed for:

- traditional large instruction set architectures, such as Intel x86, AMD, ARM, Motorola 680x0, Sun SPARCs, and DEC-Alpha,
- newer reduced instruction set architectures (RISC), such as RISC-V, SGI MIPS, IBM/Motorola PowerPC,
- smartphones, home theatre equipment, routers and access-points, and
- parallel and pipelined architectures.

All it requires is a ported C compiler

Once a C compiler has been developed for a new architecture, the terabytes of C programs and libraries available on other C-based platforms can also be ported to the new architecture.

What about assembly languages?

It is often quoted that a compiled C program will run only 1-2% slower than the same program hand-coded in the native assembly language for the machine.

But the obvious advantage of having the program coded in a readable, high level language, provides the overwhelming advantages of maintainability and portability.

Very little of an operating system, such as Windows, macOS, or Linux, is written in an assembly language - in most cases the majority is written in C.

Even an operating system's device drivers, often considered the most time-critical code in an operating system kernel, today contain assembly language numbered in only the hundreds of lines.

The unreadability of C programs

C is described as nothing more than a glorified assembly language, meaning that C programs can be written in such an unreadable fashion that they look like your monitor is set at the wrong speed.

(in fact there's a humorous contest held each year, The International Obfuscated C Code Contest to design fully working but indecipherable code,
and the (defunct) Underhanded C Contest whose goal is to write code that is as readable, clear, innocent and straightforward as possible, and yet it must fail to perform at its apparent function).

Perhaps C's biggest problem is that the language was designed by programmers who, folklore says, were not very proficient typists.

C makes extensive use of punctuation characters in the syntax of its operators and control flow. In fact, only the punctuation characters

@ ` and $

are *not* used in C's syntax! (and DEC-C once used the $ character, and Objective-C now uses the @).

It is not surprising, then, that if C programs are not formatted both consistently and with sufficient white space between operators, and if very short identifier names are used, a C program will be very difficult to read.

To partially address these problems, a number of text-editors, integrated development environments (IDEs), and beautification programs (such as *indent*) can automatically reformat our C code according to consistent specifications.

Criticisms of C's execution model

- C is criticized for being too forgiving in its type-checking at compile time.

  It is possible to *cast* an instance of some types into other types, even if the two instances have considerably different types.

  A pointer to an instance of one type may be coerced into a pointer to an instance of another type, thereby permitting the item's contents to be interpreted differently.

- Badly written C programs make incorrect assumptions about the size of items they are managing. Integers of 8-, 16-, and 32-bits can hold different ranges of values. Poor choices, or underspecification can easily lead to errors.

- C provides *no runtime protection* against arithmetic errors.

  There is no exception handling mechanism, and errors such as division-by-zero and arithmetic overflow and underflow, are not caught and reported at run-time.

- C offers *no runtime checking* of popular and powerful constructs like pointer variables and array indices.

  Subject to constraints imposed by the operating system's memory management routines, a pointer may point almost anywhere in a process' address space and seemingly random addresses may be read or written to.

  Although all array indices in C begin at 0, it is possible to access an array's elements with negative indices or indices beyond the declared end of the array.

There are occasions when each of these operations make sense, but they are rare.

C does not hold the hand of lazy programmers.

We avoid all of these potential problems by learning the language well, and employing safe programming practices.

What is the best programming language?

The question, even arguments, of whether Python, C, Java, Visual-Basic, C++, C# .... is the best general purpose programming language is *pointless*.

> *C and C++ are only the foundation because they happened to become popular due to a bunch of miscellaneous factors, not because they are inherently great inventions in themselves. Also, they (and their standard libraries) evolved over time to their current state.*
>
> *It's like saying English and Spanish are the most important languages because they are fundamentally the "best-invented" ones, not because of the accidents of fate that were colonial expansion, WWII, and the Internet.*
>
> *— Anon*

The important question is:

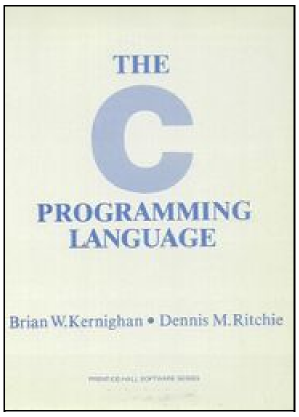
- **"which language is most suited for the task at hand?"**

This unit will answer the questions:

- **"when is C the best language to use?"** and
- **"how do we best use C's features for systems programming?"**

Through a sequence of units offered by Computer Science & Software Engineering you can become proficient in a wide variety of programming languages - procedural, object-oriented, functional, logic, set-based, and formal - and know the most appropriate one to select for any project.

The Standardization of C - K&R C



Despite C's long history, being first designed in the early 1970s, it underwent considerably little change until the late 1980s.

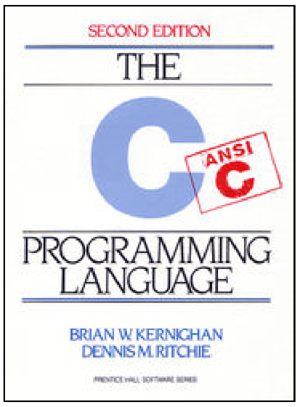
This is a very lengthy period of time when talking about a programming language's evolution.

The original C language was mostly designed by Dennis Ritchie and then described by Brian Kernighan and Dennis Ritchie in their imaginatively titled book *The C Programming Language*.

The language described in this seminal book, described as the *"K&R"* book, is now described as *"K&R"* or "old" C.

228 pages.

The Standardization of C - ANSI-C (K&R-2)



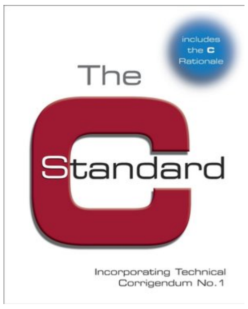
In the late 1980s, a number of standards forming bodies, and in particular the American National Standards Association X3J11 Committee, commenced work on rigorously defining both the C language and the commonly provided standard C library routines. The results of their lengthy meetings are termed the ANSI-X3J11 standard, or informally as ANSI-C, C89, or C90.

The formal definition of ANSI-C introduced surprisingly few modifications to the old *"K&R"* language and only a few additions.

Most of the additions were the result of similar *enhancements* that were typically provided by different vendors of C compilers, and these had generally been considered as essential extensions to old C. The ANSI-C language is extremely similar to old C. The committee only introduced a new base datatype, modified the syntax of function prototypes, added functionality to the preprocessor, and formalized the addition of constructs such as constants and enumerated types.

272 pages.

The Standardization of C - ANSI/ISO-C99 and ISO/IEC 9899:2011 (C11)



A new revision of the C language, named ANSI/ISO-C99 (known as C99), was completed in 1999.

Many features were "cleaned up", including the addition of Boolean and complex datatypes, single line comments, and variable length arrays, and the removal of many unsafe features, and ill-defined constructs.

753 pages.

imageGenerator.net
BREAKING NEWS - C11 STANDARD OUT
Olympics delayed, national holiday announced.

A revision of C99, ISO/IEC 9899:2011 (known as C11), was completed in December 2011.

In this unit we will focus exclusively on C11,
and only mention other versions of C when the differences are significant.

If the C compiler on your computer system does not support C11, you will be able to undertake most exercises using C99 - **but are strongly encouraged to upgrade your C compiler**

(a topic discussed in our first workshop).

What C Standardization Provides

These quite formal standards specify the form and establishes the interpretation of programs written in the C programming language. They specify:

- the representation of C programs;
- the syntax and constraints of the C language;
- the semantic rules for interpreting C programs;
- the representation of input data to be processed by C programs;
- the representation of output data produced by C programs;
- the restrictions and limits imposed by a conforming implementation of C.

They *do not* specify:

- the mechanism by which C programs are transformed for use by a data-processing system;
- the mechanism by which C programs are invoked for use by a data-processing system;
- the mechanism by which input data are transformed for use by a C program;
- the mechanism by which output data are transformed after being produced by a C program;
- the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;
- all minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

What's (deliberately) missing from the C language?

At first glance, the C language appears to be missing some commonly required features that other languages, such as Java, provide in their standards.

For example, C *does not* provide features for graphics, networking, cryptography, or multimedia.

Instead, C permits, enables, and encourages additional 3rd-party libraries (both open-source and commercial) to provide these facilities. The reason for these "omissions" is that C rigorously defines what it does provide, and rigorously defines how C must interact with external libraries.

Here are some well-respected 3rd-party libraries, frequently employed in large C programs:

Application domain	(a sample of) 3rd-party libraries
operating system services (files, directories, processes, inter-process communication)	OS-specific libraries, e.g. glibc, System32, Cocoa
web-based programming	libcgi, libxml, libcurl
data structures and algorithms	the generic data structures library (GDSL)
GUI and graphics development	OpenGL, GTK, Qt, wxWidgets, UIKit, Win32, Tcl/Tk
image processing (GIFs, JPGs, etc)	GD, libjpeg, libpng
networking	Berkeley sockets, AT&T's TLI
security, cryptography	openssl, libmp
scientific computing	NAG, Blas3, GNU scientific library (gsl)
concurrency, parallel and GPU programming	OpenMP, CUDA, OpenCL, openLinda (thread support is defined in C11, but not in C99)

Lecture 1 Summary

- Understanding the relationship between a programming language and contemporary operating systems is central to developing many skills in Computer Science.
- This unit introduces the C11 programming language, and fundamental operating system concepts such as processes, memory management, file-systems, and operating system services.
- Following standards is critical to developing robust, portable, and easy to maintain systems programs. This unit makes constant reference to the C11 and POSIX standards.
- Debates over which is the *best* programming language are pointless. Of greatest importance is choosing and understanding a programming language well-suited for the systems or application domain.
- Despite its age, C is very deliberately introduced as a vehicle to explain systems-programming, because of its history and influence on other languages, simplicity, continued widespread use, standardisation, and portability.
- Similarly, POSIX is introduced because of its widespread use, consistency, standardisation, and portability.