Systems Programming and Portability

In this unit we've focused on system programming - understanding the interface between the operating system and application programs.

Operating systems are the best examples of programs that need to be aware of hardware's specifications and limitations, and to successfully hide as much of this detail from potential applications through good software engineering practices.

If the operating system, itself, has any chance of being *ported* to different architectures, its own implementation must identify and isolate its hardware dependencies.

Unix, the historic forefather of Linux and macOS (and many others), was the first portable operating system, reimplemented in C to support its migration from early Digital Equipment Corp (DEC). minicomputers. C itself was invented specifically for the purpose of enabling Unix to be portable.

> *We here at Bell Laboratories were truly dumfounded when this visitor from an unknown school in Australia reported his elegant procedure.*
>
> *— Doug McIlroy, Head Unix Research Group, Bell Laboratories*

- UNIX: a portable operating system? [Miller, 1978].
- Unix portability: underutilized in embedded development [Crooks, 2002].
- The First Port of UNIX [Reinfelds, 1978].

Today, of course, we see Linux ported to nearly every form of contemporary architecture because:

- hardware-dependent code has been identified and isolated,
- software abstractions and application-programming interfaces hide hardware characteristics from applications, and
- successful applications do not introduce, or depend upon, any hardware dependencies.

What is portability?

A program may be considered *portable* if it can be 'moved', migrated, to different computing environments.

These environments do not just include different operating systems, running on different forms of hardware, but can include different (human) interfaces and natural languages.

Many (most?) operating systems are written in C and are, in theory, portable. This is possible because C toolchains (the pre-processor, compiler, and linker) are supported by header files and libraries that have 'extended' the language, without requiring the language, itself, to be changed.

(the above paragraph is not strictly correct, as C11 has recently added new features aiding portability, such as in-language support for Unicode).

C is portable at the level of its source-code

C programs require compiling in their *new* computing environment, or *cross-compiled* on an existing environment with knowledge of the destination hardware architecture and able to provide the necessary libraries. Examples include being able to develop programs on an Intel-based Linux platform, destined for an ARM-based Raspberry Pi platform (also running Linux), or developing a program under Apple's macOS destined for an iPhone (and then both uploaded (by network or cable) to the new environment).

C's source-level portability is in contrast to:

- Java's use of an *architecture-independent bytecode*. Java's source code is compiled on one platform, and the resulting bytecode copied to a destination platform with a *platform-specific* implementation of a Java Virtual Machine (JVM) to interpret the bytecode.
- Python's portability coming from its interpretation of its source code with a *platform-specific* Python interpreter.

Your C compiler's version and default language standard

Now a decade since C11 was released, and contempory compilers, such as *gcc* and *clang*, support all C11 features (on hosted platforms), and support requests for backward compatability from earlier standards (*cc -std=cXX ...*).

While easy to determine the version of a *compiler* being used:

```
macOS-prompt> cc --version
Apple clang version 15.0.0 (clang-1500.0.40.1)
Target: arm64-apple-darwin23.0.0
Thread model: posix

Ubuntu-prompt> cc --version
cc (Ubuntu 12.3.0-1ubuntu1~23.04) 12.3.0
```

compiler front-ends support many languages and versions, so knowing the compiler's version is not much use. Instead, we need to know how our source code is being compiled, *at compile time*. We can test against the *__STDC_VERSION__* preprocessor token, and then (possibly) compile different code/functions in our program:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
#if __STDC_VERSION__ >=  201710L
    printf("hello from C18!\n");

#elif __STDC_VERSION__ >= 201112L
    printf("hello from C11!\n");

#else
    #error This program demands features from C11 or later

/*
#elif __STDC_VERSION__ >= 199901L
    printf("hello from C99!\n");
#else
    printf("hello from (ANSI-C) C89/C90!\n");
 */

#endif

    return 0;
}
```

This assists our goal of portable programming by ensuring that a program's required features are supported by the local compiler, and its default compilation arguments.

Pre-defined preprocessor tokens

The recent examples enabling detection of C language standard and operating system platform, are a small, but important sample of the information available when compiling programs.

We can see the pre-processor's pre-defined tokens with:

```
prompt> cc -dM -E - < /dev/null
(133 lines on MacOS, 385 on Ubuntu/Linux ...)
```

Some of the following examples (not specifically related to portability) taken from: gcc's Standard Predefined Macros

The standard predefined macros are specified by the relevant language standards, so they are available with all compilers that implement those standards.

__FILE__

This macro expands to the name of the current input file, in the form of a C string constant.

__LINE__

This macro expands to the current input line number, in the form of a decimal integer constant.

`__FILE__` and `__LINE__` are useful in generating an error message to report an inconsistency detected by the program. C99 also introduced `__func__`, and GCC has provided `__FUNCTION__` for a long time.

__STDC__

In normal operation, this macro expands to the constant 1, to signify that this compiler conforms to ISO Standard C.

__STDC_VERSION__

This macro expands to the C Standard's version number, a long integer constant of the form `yyyymmL` where *yyyy* and *mm* are the year and month of the Standard version.

__STDC_HOSTED__

This macro is defined, with value 1, if the compiler's target is a *hosted environment*. A hosted environment has the complete facilities of the standard C library available.

Detecting the target operating system platform

Similarly, at compile-time we can determine the operating system platform *for which we're compiling* (note, if we're *cross-compiling*, this will not be our native platform).

Based on this information we can conditionally report an inability to support specific platforms, or can include our own implementation of functions not otherwise available.

```
#ifdef _WIN64
   //define something for Windows (64-bit)
#elif _WIN32
   //define something for Windows (32-bit)
#elif __APPLE__
    #include "TargetConditionals.h"
    #if TARGET_OS_IPHONE && TARGET_IPHONE_SIMULATOR
        // define something for simulator
    #elif TARGET_OS_IPHONE
        // define something for iphone
    #else
        #define TARGET_OS_OSX 1
        // define something for OSX
    #endif
#elif __linux
    // linux
#elif __unix // all Unix-derived systems not detected above
    // Unix
#elif __posix
    // POSIX
#else
    #error unrecognized operating system platform
#endif
```

Detecting the target operating system platform, *continued*

In addition to detecting operating system versions and characteristics using the C preprocessor, we can do the same with other utilities.

Within *Makefiles* we can invoke external programs, capture their output into a*make* variable, and then conditionally execute different commands or apply different command-line options:

```
OS := $(shell uname)
NAME  = project2
BACKUPTGZ       = $(HOME)/backup/$(NAME).tgz

backup:
        $(MAKE) --no-print-directory prepare
ifeq ($(OS),Darwin)                                     # for macOS
        COPYFILE_DISABLE=1 tar zcf $(BACKUPTGZ) .
else                                                    # assuming Linux
        tar zcf $(BACKUPTGZ) .
endif
```

And, unsurprisingly, we can perform the same command sequence within *shellscripts*:

```
#!/bin/bash
OS=$(shell uname)
NAME=project2
BACKUPTGZ=$(HOME)/backup/$(NAME).tgz

case "$OS" in
    Darwin)
        echo hello from macOS
        COPYFILE_DISABLE=1 tar zcf $(BACKUPTGZ) .
        ;;
    Linux)
        echo hello from Linux
        tar zcf $(BACKUPTGZ) .
        ;;
    *)
        echo 'unsupported platform!'
        exit 1
        ;;
esac
```

Employing the correct sized integers for portability

In most of our C programming (laboratories and projects) we have employed the standard **int** datatype whenever we have simply wished to count something, or to loop a small number of times.

We have not cared (probably not even thought) whether the host architecture supported integers of length 16-, 32-, or 64-bits, but have been confident (on laptops and desktops) that integers were at least 32-bits long; meeting our typical requirements.

For different applications, the actual storage size of an integer may be significant, and a portable program should enforce its requirements. For example, if we required an array to store temperature samples on, say, an Internet-of-Things (IoT) device, then an 8-bit integer may be *sufficient*, or *necessary* if we required a million of them.

C99 introduced the standard header file <*stdint.h*> which defines the C99 *base types* required to employ integers of exactly the required size, together with their limits. An extract:

```
typedef signed char          int8_t;
typedef short int            int16_t;
typedef int                  int32_t;
# if __WORDSIZE == 64
typedef long int             int64_t;
# else
typedef long long int        int64_t;
# endif
#endif
....
/* Minimum of signed integral types.  */
# define INT8_MIN            (-128)
# define INT16_MIN           (-32767-1)
# define INT32_MIN           (-2147483647-1)
# define INT64_MIN           (-__INT64_C(9223372036854775807)-1)
/* Maximum of signed integral types.  */
# define INT8_MAX            (127)
# define INT16_MAX           (32767)
# define INT32_MAX           (2147483647)
# define INT64_MAX           (__INT64_C(9223372036854775807))
```

Similar support is provided for unsigned integers, and float-point numbers of different lengths (32-, 64-, 128-bits).

Employing the correct form of these datatypes is critcal in many application domains demanding portable software - including networking protocols, cryptography, and image processing.

Employing the correct sized integers for portability, *continued*

While the C99 and C11 *<stdint.h>* header file defines the datatypes, it doesn't define how we may perform input and output on them, independent of their actual storage size. C99 further standardized the new header file *<inttypes.h>* to achieve this.

When using standard C functions like *printf()* and *sscanf()*, we can employ C's ability for the *compiler* (i.e. at compile-time, not run-time) to concatenate string constants. Within the *<inttypes.h>* header file, *PRIi64* may be, for example, defined as the string *"i"* or *"li"* depending on the target environment's architecture:

```
#include <stdint.h>
#include <inttypes.h>

    int64_t    nbytes;
    ....
    printf("%" PRIi64 "MB\n", n / (1 << 20) );
```

Similar support exists within the C99 and C11 standards for varying sized pointers (typically 32- or 64-bits), the ability to perform I/O on their character (string) representations, and to select the appropriate sized integer so that it may hold a pointer value.

Portable programs are 'team-players'

Simply porting a program to a different computing environment does not guarantee that the program will be able to operate successfully, or be accepted by users, in the new environment.

Systems-focused programs also need to 'fit in' with the new computing environment, to both *interoperate* with existing utilities, and also contribute something new.

This requires programs to make use of existing operating system supported runtime features and interfaces in a consistent manner. This makes it easier for users to quickly understand and benefit from the newly ported program.

An excellent introduction to this topic is The Art of Unix Programming, by Eric Steven Raymond, 2003:

Chapters 1 and 5 are the most relevant to the material discussed in this lecture.
In addition, some other good Chapters/Sections that are not too long or dry (in order of relevance), are:

Chpt 10 - Configuration: What Should be Configurable?; Environment Variables; Command-Line Options
Chpt 11 - Unix Interface Design Patterns: The Filter Pattern -> The ed Pattern
Chap 19 - Open Source
Chpt 16 - Reuse

A example of 'team-players' - filters

One of the most successful ideas introduced in early Unix systems was the interprocess communication mechanism termed a *pipe*. Pipes enable shells (or other programs) to connect the output of one program to the input of another, and for arbitrary sequences of pipes - a *pipeline* - to *filter* a data-stream with a number of transformations.

A great pipeline example, providing a rudimentary spell-checker:

```
prompt>  tr -cs 'A-Za-z' '\n' < inputfilename | sort -u | comm -23 - /usr/share/dict/words
```

Programs typically used in pipelines are termed *filters*, and they work in combination because of their simple communication schemes which do not add 'unexpected detail' to their output, so that programs reading that output as their input only have the expected data-stream to process.

It's for this reason that programs don't produce verbose natural-language descriptions of their output, no headings for tables of data, unless a specific command-line option requests it. Just the facts.

Unicode support in C11

One of the long-overdue features added to the C11 standard is support for Unicode character sets, through UTF-8, UTF-16, and UTF-32 encodings.

C was missing this feature for a long time, and C programmers had to use third-party libraries such as *IBM's International Components for Unicode (ICU)*.

Before C11, we only had **char** and **unsigned char** types, 8-bit integer variables used to store ASCII and Extended ASCII characters. By creating arrays of these ASCII characters, we could create ASCII strings.

Portable programs should not be limited to communicating only in English, or ISO-Latin languages. There are *thousands* of other natural languages, employing character sets other than the English alphabet. Portable program should support these without requiring a different program, or source-code base, for each language.

ASCII and Extended-ASCII - 8-bit character sets

The ASCII standard has 128 characters each stored in 7 bits. Extended-ASCII adds another 128 characters to total 256 characters; an 8-bit or one-byte variable is sufficient. See *man ascii*.

Support for ASCII characters and strings is fundamental, and will never be removed from C. C11 adds support for new character sets and, therefore, new strings require a different number of bytes, not just one byte, for each character.

Suddenly, characters may be of different lengths (1-, 2- or 4-bytes long), and it's the *value* of the character that determines its length. Consider how this would affect an inplementation of, say, the standard C11 *strlen()* function, which just counts the bytes found until the NULL-byte!

Unicode support in C11, *continued*

The Unicode standard introduced mechanisms supporting more than one byte to encode all characters in ASCII, Extended-ASCII, and 'wide' characters in *thousands* of different natural languages. These methods are termed *encodings*.

Unicode defines 3 well-known encodings: UTF-8, UTF-16, and UTF-32:

- **UTF-8** uses the first byte for storing the first half of ASCII characters, and following next bytes, usually up 4, for the other half of ASCII characters together with all other wide characters. Hence, UTF-8 is considered as a *variable-sized* encoding.
- Like UTF-8, **UTF-16** uses one or two *words* (each word occupying 16 bits) for storing all characters - In both UTF-8 and UTF-16, a smaller number of bytes are used for more frequent characters. Most of the characters require *up to* two bytes. Hence it is also a *variable-sized* encoding.
- **UTF-32** uses exactly 4 bytes for storing the values of *all* characters; therefore, it is a *fixed-sized* encoding. UTF-32 uses a fixed number of bytes (4) even for ASCII characters, but does restore our idea of 'counting' charcaters, and enables individual characters to act as array indicies..

Note that C11 does not define new standard functions to operate on Unicode strings, therefore we have to write a new *strlen()* function for them.

However, many Unicode conversion functions are defined in the new *<uchar.h>* header file.

An excellent introduction to Unicode - unicodebook.readthedocs.io/unicode_encodings.html
some thoughts on their support in C11: Unicode operators for C,
and some example code: Unicode in C11