CSC / CPE 357

Systems Programming

UNIX Philosophy

- Make each program do one thing well. To do a new job, **build afresh** rather than complicate old programs by adding new features.

- Expect the output of every program to become the input to another, as yet unknown, program.

Doug McIlroy, Elliot Pinson and Berk Tague, 1978

The C Programming Language

- Created in 1972 by Dennis Ritchie, evolved alongside UNIX
- “Low-level” language that exposes underlying features of machine architecture
- Procedural, not object-oriented
- Small standard library, compared to Java, Python, C++, etc.

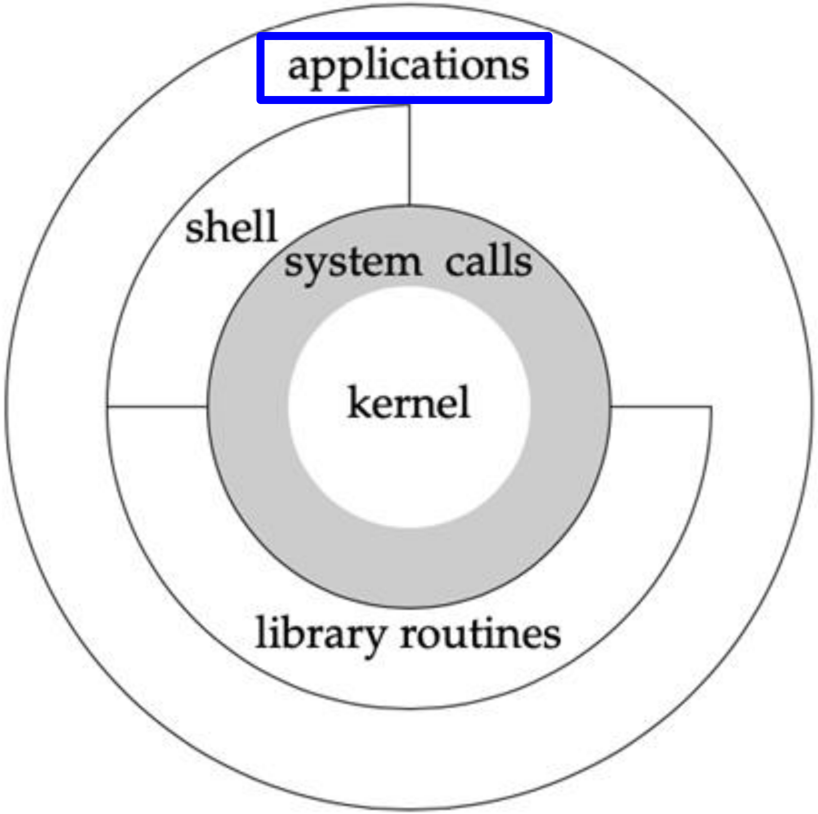


Figure 1.1 Architecture of the UNIX operating system

Diagram from: Advanced
Programming in the UNIX
Environment, 3rd Ed.

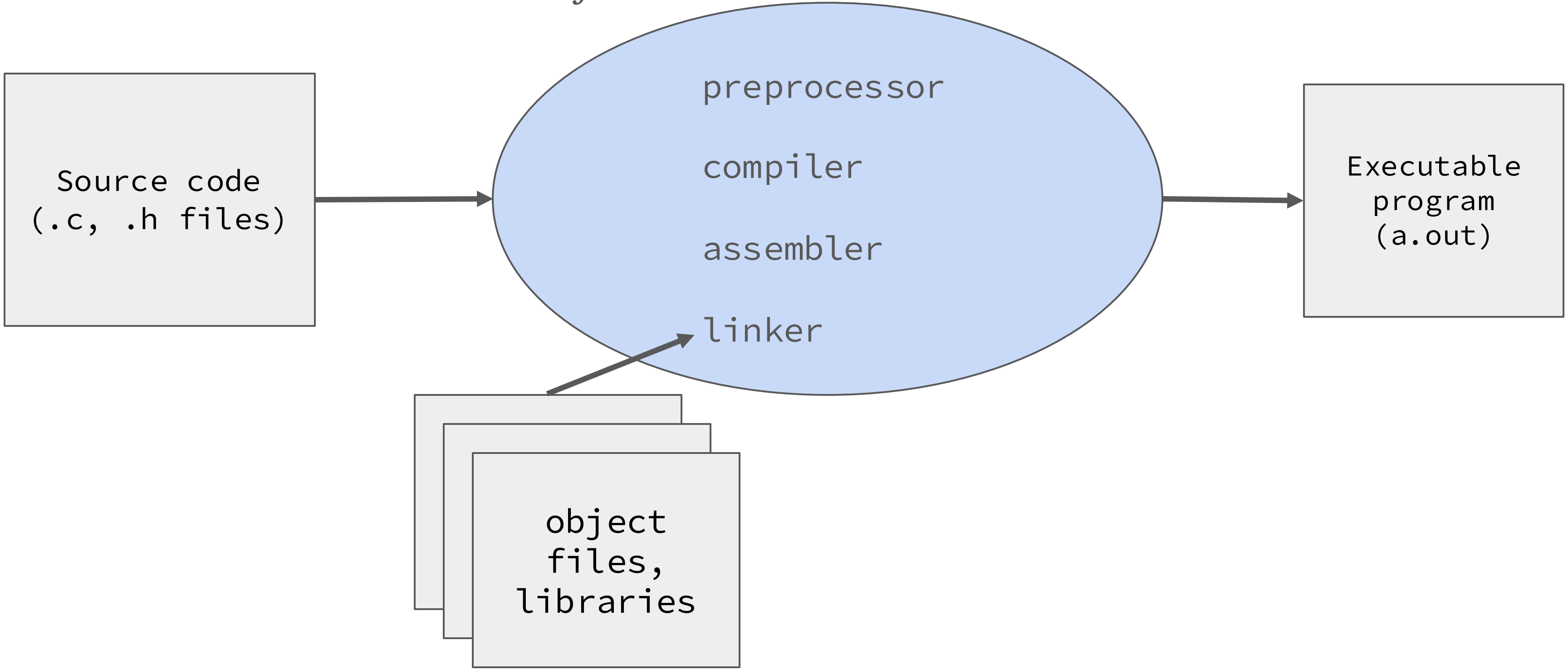
C Filenames

- **`.c`** C source file
- **`.h`** Header file
  - type definitions, function prototypes and declarations
- **`.o / a.out`** Compiled object file

C Compiler / Linker

A **compiler** generates object code files (machine language) from source code.

A **linker** combines these object code files into an executable.



Source code
(.c, .h files)

preprocessor

compiler

assembler

linker

Executable
program
(a.out)

object
files,
libraries

C Program Layout (*.c)

```
#include <system_header.h>
#include "local_header.h"

#define macro_name macro_expr


/* declare functions */
/* declare external variables & structs */

int main(int argc, char* argv[]) {
       /* the code */
}

/* define other functions */
```

C Data Types

C data types map to typical hardware capabilities:

- **`char`** a single byte, capable of holding one character (eg. letter)
- **`int`** an integer, with a size that matches the "natural" size of integers on the host machine (no less than 16 bits)
- **`float`** single-precision floating point
- **`double`** double-precision floating point
- **`boolean`** (*no direct support*) 0 is false nonzero is true.

Variable Declarations

In C, variables must be declared before use. A **declaration** introduces an identifier and describes its type. A declaration is what the compiler needs to accept references to that identifier. For example:

```
extern int i;  // extern indicates that a variable is available elsewhere
               // resolving is deferred to the linker
```

A **definition** actually instantiates/implements an identifier. It's what the linker needs to link references to those entities:

```
int i = 42;

char upper_a = 'A';
```

A definition can be used without a declaration.

Data Type Qualifiers

The data type qualifier **`short`** may be applied to integers to provide different lengths of integers where appropriate; **`long`** may be applied to integers or doubles.

Typically: `short int` is 16 bits, `long int` is 32 bits and `int` is either 16 or 32 bits.

```
    short int course_no;
        long int counter;
```

The "`int`" may be omitted. Equivalent declarations:

```
    short course_no;
        long counter;
```

Data Type Qualifiers

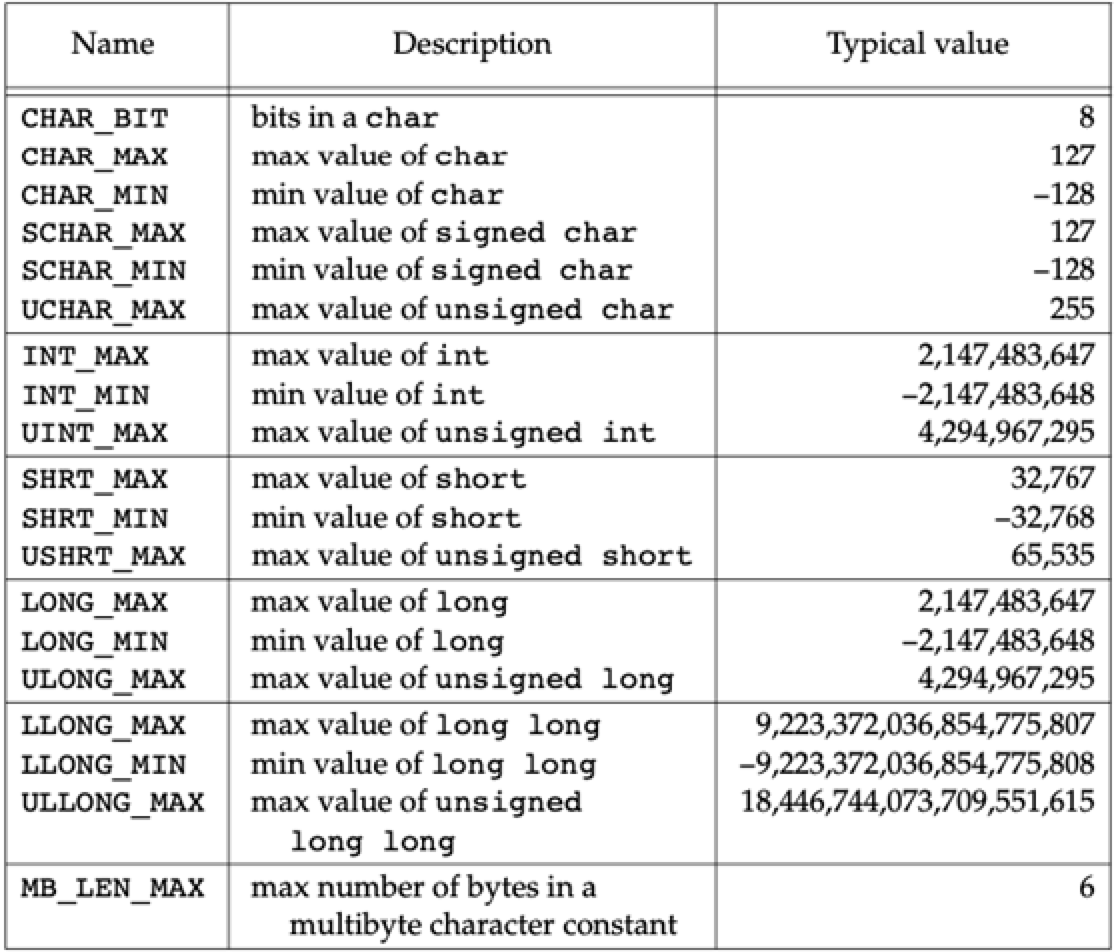
Data type qualifiers control certain behavior. Examples:

- **const** specifies that a variable's value cannot be changed

```
const float pi = 3.14159;
pi = 3.2;  // invalid
```

- **unsigned** holds a value that is always positive or zero

```
unsigned int i;
i = -1;
```



Name	Description	Typical value
`CHAR_BIT`	bits in a `char`	8
`CHAR_MAX`	max value of `char`	127
`CHAR_MIN`	min value of `char`	–128
`SCHAR_MAX`	max value of `signed char`	127
`SCHAR_MIN`	min value of `signed char`	–128
`UCHAR_MAX`	max value of `unsigned char`	255
`INT_MAX`	max value of `int`	2,147,483,647
`INT_MIN`	min value of `int`	–2,147,483,648
`UINT_MAX`	max value of `unsigned int`	4,294,967,295
`SHRT_MAX`	max value of `short`	32,767
`SHRT_MIN`	min value of `short`	–32,768
`USHRT_MAX`	max value of `unsigned short`	65,535
`LONG_MAX`	max value of `long`	2,147,483,647
`LONG_MIN`	min value of `long`	–2,147,483,648
`ULONG_MAX`	max value of `unsigned long`	4,294,967,295
`LLONG_MAX`	max value of `long long`	9,223,372,036,854,775,807
`LLONG_MIN`	min value of `long long`	–9,223,372,036,854,775,808
`ULLONG_MAX`	max value of `unsigned long long`	18,446,744,073,709,551,615
`MB_LEN_MAX`	max number of bytes in a multibyte character constant	6

Data Type Qualifiers (continued)

In addition to qualifiers such as const and unsigned, the following modifiers allow fine-grained control:

- **`static`** link now
- **`register`** store value in a CPU register, if possible
- **`extern`** extern declarations (for variables in libraries)
- **`volatile`** for things that change by themselves
- **`restrict`** promises no pointer aliasing

We will return to these as they become relevant.

C Data Structures

- **Arrays** are contiguous chunks of memory
  - No default initialization of memory content
  - No bounds checking (length not stored)

- C-**strings** are null-terminated arrays of characters
  - `char x[] = "hi";`
  - `string.h` has helpful library/utility functions

- **Structs** are collections of fields (variables)
  - "Object-like" but no methods

The main() Function

```
int main(int argc, char* argv[])
```

- **`argc`** contains the count of arguments on the command line
  - Executable name counts as one, plus one for each argument

- **`argv`** is an array of the arguments as strings

- Example: `$ ./a.out 14 hi`
  - `argc = 3`
  - `argv[0]="./a.out", argv[1]="14", argv[2]="hi"`

`Error Handling in C`

- No built-in exception handling (no try/catch)

- Errors are returned as integer error codes from functions
  - Error handling is inelegant
  - `CONSTANT_NAMES` are defined to avoid "magic" integer values – need to look up in documentation

- Global variable **`errno`** holds value of last system error

Error Handling

- Processes exit (e.g., return from `main`) with status code

- Standard codes found in `stdlib.h`:
  - `EXIT_SUCCESS (usually 0)`
  - `EXIT_FAILURE (non-zero)`

- “Crashes” trigger signals from OS (e.g., `SIGSEGV` for segfault)

C Functions

- Parameters: all passed by **value**

- Function declarations (prototypes)
  - Specifies function arguments and return type
  - Example: `int power(int base, int n);`

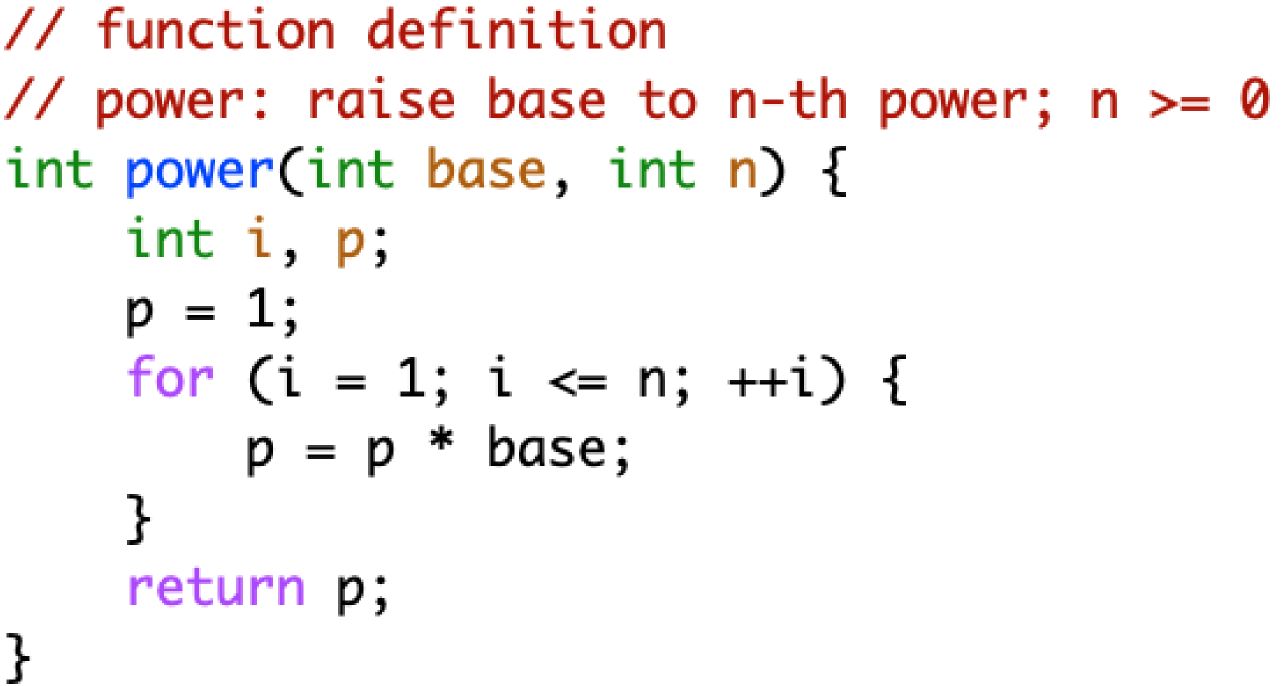
- Function definitions

Function Declaration vs. Definition

- Declaration
  - Function prototype, external variable declaration
  - Often placed in header files (`.h`) incorporated via `#include`
  - Should appear before first use in all files that use the function

- Definition
  - Code for function, or variable definition that creates storage
  - Must be *exactly one definition* of each thing (no duplicates)

Function Declaration vs. Definition



```c
// function definition
// power: raise base to n-th power; n >= 0
int power(int base, int n) {
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i) {
        p = p * base;
    }
    return p;
}
```

Gist

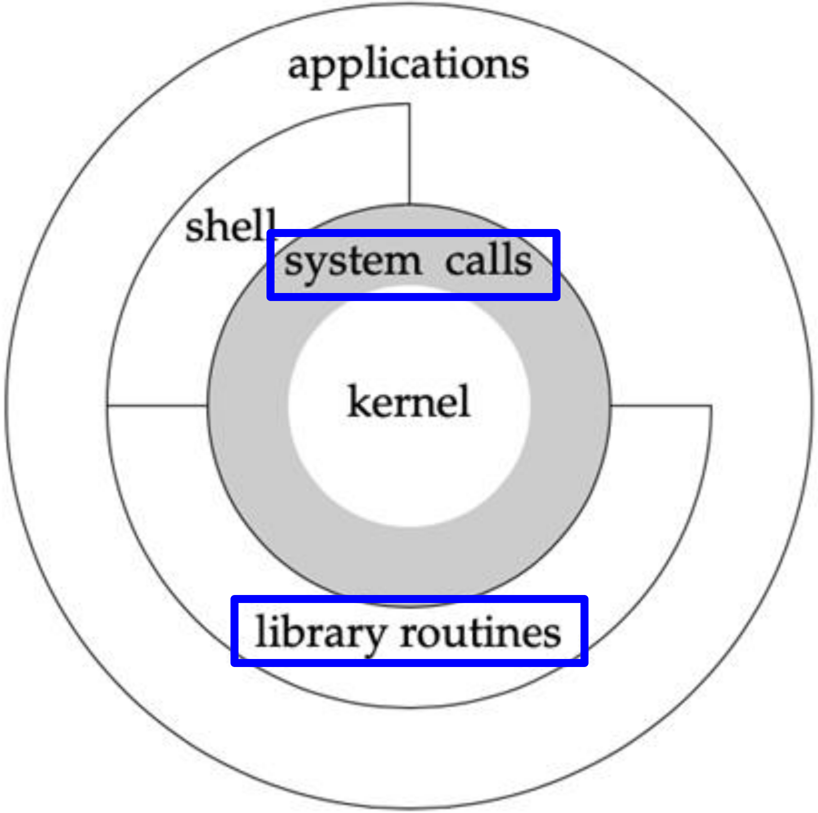


Figure 1.1 Architecture of the UNIX operating system

Diagram from: Advanced Programming in the UNIX Environment, 3rd Ed.

System Calls and Library Functions

- **System call**: entry point directly into the kernel
    - Linux provides ~400 system calls
    - Exposed as regular C functions

- Contrast with: **library functions**, which do not represent a direct entry point into the kernel
    - `printf()` library function invokes the `write()` system call
    - `malloc()` library function invokes the `sbrk()` system call
    - many library functions do not involve system calls, examples:
        - `strcpy()` copy a string
        - `atoi()` convert ASCII to integer

- Manual pages:
    - "section 1" for general UNIX commands: `man 1 cd` (or, equivalently: `man cd`)
    - "section 2" for system calls: `man 2 sbrk`
    - "section 3" for library functions: `man 3 printf`

Code Style

Overall goals:

- Correctness
- Readability / Maintainability
- Security
- Performance

Many different code style conventions (often specified at the team/company level) Good starting points:

- Linux Kernel Coding Style (a related checkpatch tool)
- CS50 Style Guide
- SEI CERT C Coding Standard (Secure Code)