FCPA 2022  
  
Memory Management

Student Workbook 12

January 25, 2022

Author

Paul Kimball  
Interface Associates

1. Variables

Variables

* A variable is a named region of storage
  + Holds a value (called an object)
* We already saw several properties of a variable
  + Identifier

Scope - local or file scope

* + Data type

Size (in bytes) - can be found with sizeof operator

* + Starting address
  + Value
* Now we'll consider some other features:
  + Placement
  + Initialization
  + Lifetime
  + Type Qualifiers

Data Types

* *Scalar* types
  + A concrete human-meaningful value that you might want to manipulate

A number, e.g., 14

A character, e.g., 'W'

A byte address

* + Come in several different sizes, depending on what they hold

Operations that combine different scalar types can result in truncation, overflow, or conversion of values

## *Aggregate* types are constructed from multiple scalar types

### array

### struct

### union

### enum

"Pointer" Types

* Pointer types are a special kind of scalar type
* A pointer represents the byte address where *something* of interest starts in memory
  + A scalar value

It "points" to a place in memory

* + A sequence of scalar values (an array)
  + A function
  + An allocated chunk of memory
  + The beginning of the stack
  + The hardware address of some sensor
  + Lots of other stuff
* All pointer types are the same size (4 or 8 bytes)
  + But what they point to is not!
  + Operations involving different pointer types can result in amazing magic or dreadful catastrophe
* All pointer types can be cast as "pointer to void" (void \*)
  + (void \*) is the generic pointer type
  + Casting hides the differences between pointers from the compiler!
* Trying to use a *numeric* value (int, float) as a pointer, or vice-versa, is VERY VERY DANGEROUS
  + Meaning, "easy to get it wrong if you're not paying attention"
  + The compiler will give you a serious warning about this, and you should pay close attention

Pointer Variables

* A pointer variable is declared using a pointer type

float **\*** val\_ref; // pointer to float

char **\*** first\_char; // pointer to char

sensor\_event **\*** event; // pointer to sensor\_event

* An unitialized pointer variable is an accident waiting to happen! Always initialize your pointers before you use them
  + NULL (not actually a good value, but you can test it)
  + Another (valid) pointer
  + The address of a string literal ("hello")
  + The address of an existing variable, e.g. &myevent
  + The address of an array or array element
  + The address of a function
  + The address of memory that has been allocated dynamically with malloc
* Attempts to read/write a "bad" address can result in
  + Segmentation Fault

NULL address

Memory beyond the current program break

Kernel memory

* + Bus error

Unaligned access

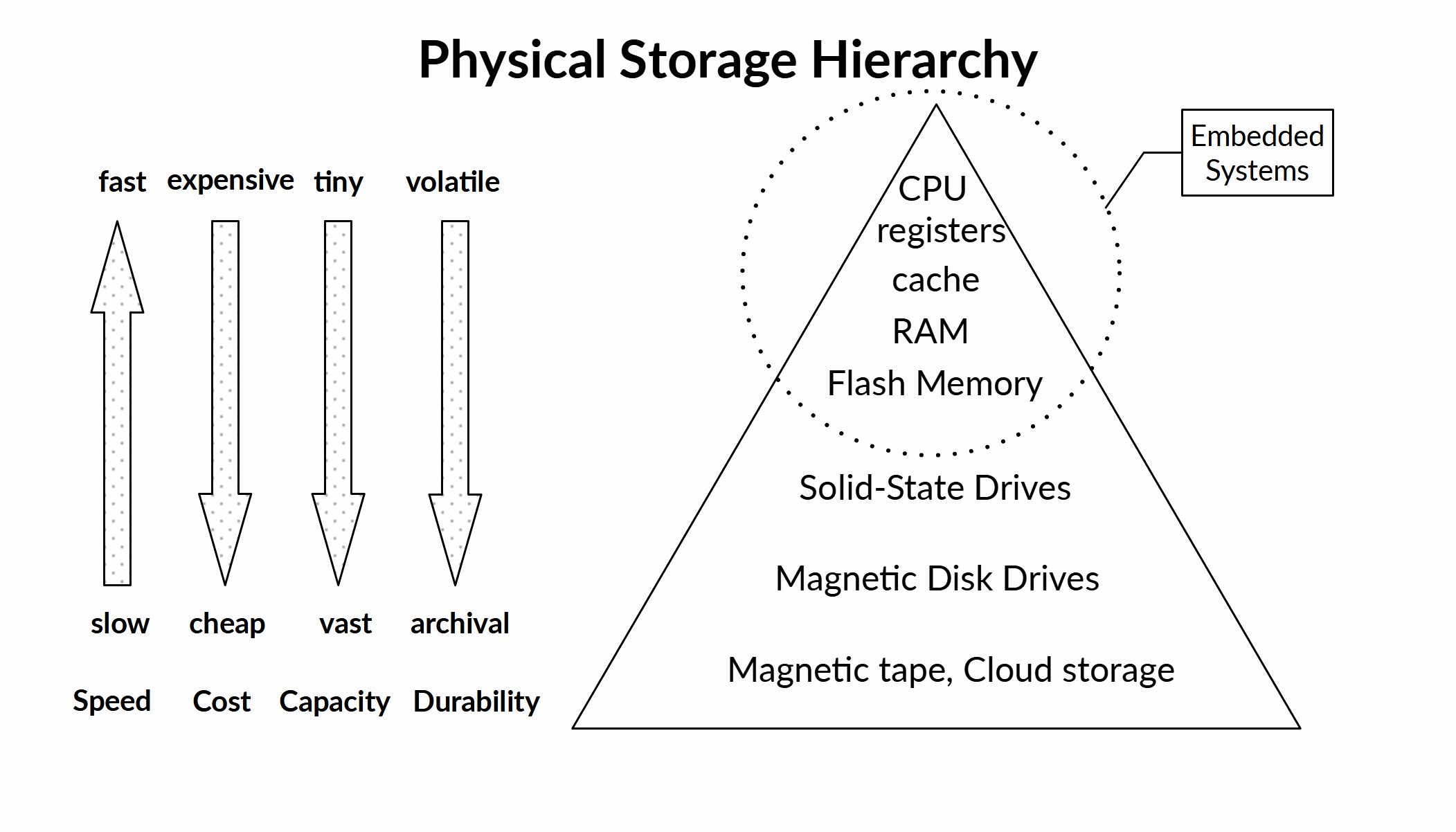
* + Corruption of memory (!)

1. Memory Management

The CPU

* The CPU is like a chipper-shredder. It goes fast, but you have to keep it fed or you're just wa sting time and gas.

Physical Storage



Memory

* Memory is composed of units of storage called "bytes"
  + **A byte is a unit of memory that is large enough to hold a single character in the execution basic character set**

The number of bits in a byte is implementation-dependent

See the value of CHAR\_BIT in limits.h if you want to know

* Each byte has its own address

### Each byte can be read and written separately

But there may be a performance penalty for "unaligned" addresses

* The specification does NOT define

### How much memory is available at execution time

### Which addresses are valid and whether they are accessible

### The CPU's preferred alignment and size of accesses, and whether there is a penalty for unaligned access

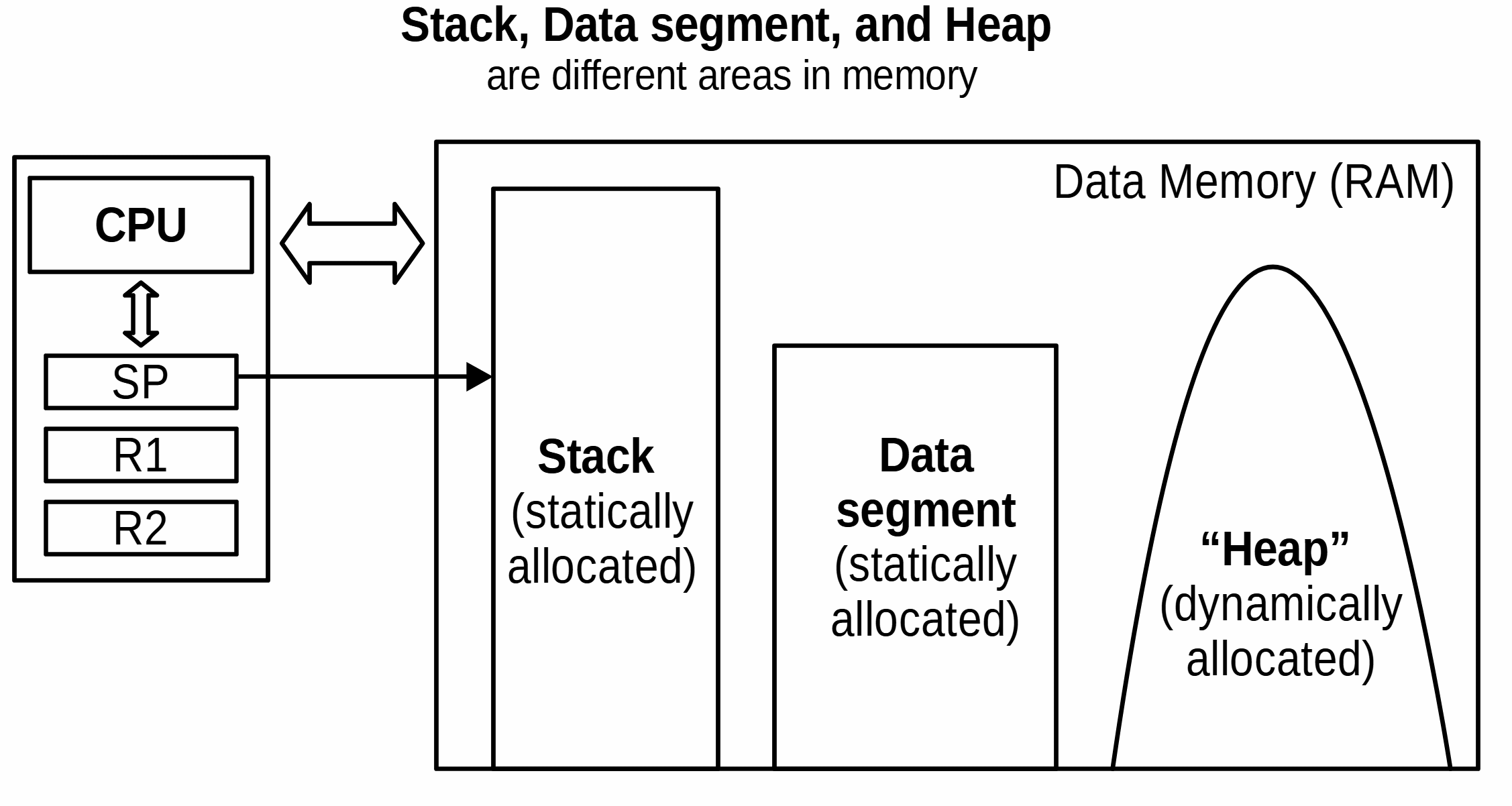
Is Memory Even Real?

* "Real" memory
  + Addresses refer to hardware byte locations in RAM
  + Addresses are established by physical placement of RAM chips and CPU address pinouts
  + **The usual type for freestanding implementations**
* "Virtual" memory
  + Addresses refer to byte locations in an idealized address space
  + The hardware Memory Mapping Unit (MMU) and operating system work together to map virtual addresses to real memory when necessary
  + **The usual type for hosted implementations**
* Virtual memory may exceed real memory (RAM) capacity
  + Page and Swap files on storage devices can hold the overflow
  + Everything slows down when this happens

In the Sea of Memory...

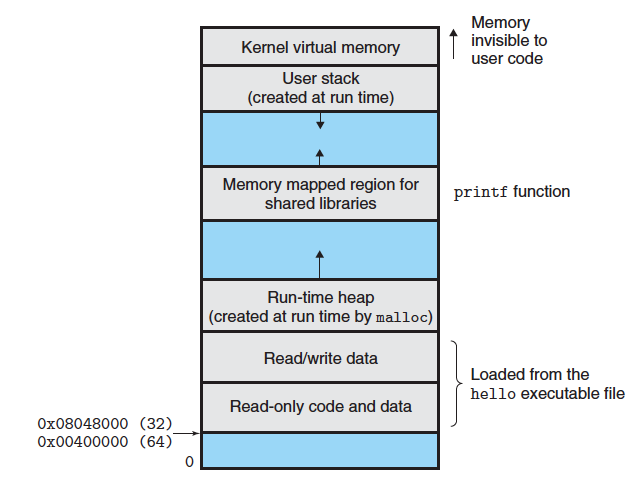
* ... There are some islands that are rumored to exist
* The "text" (or "code") segment - the unchanging land
  + Where the functions and string literals live
* The initialized "data" segment
  + Where static variables are born and raised
* The stack
  + A civilized place where local variables wait their turns in the processor registers
* The heap
  + A wild, magic place where memory can be had for the asking and (maybe) forever
* The detailed facts about these islands are debatable, and will depend on your implementation and your memory maps!

Memory Regions



* The specification never uses any of these words - these are common conventions, however.

A Popular Map



Type Modifiers

* The "cvr" modifiers tell the compiler whether it can perform certain kinds of optimization, or allow certain operations
* const
  + Write access to a const value should be denied by the compiler
* volatile
  + Value may be changed by the environment outside of program control; **Compiler must ensure that the value is checked every time it is used to see if it has changed**
* restrict
  + Applied to pointers that are passed as function parameters. Tells compiler that they will not refer to overlapping areas of memory. **Programmer is responsible for enforcing this.**

Storage class

* Memory is allocated for a variable when it is declared
  + Storage class modifiers affects the lifetime, placement, and default initialization of its value
* auto
  + **The default for all local variables**
  + Lifetime: from point of declaration until end of scope
  + Placement: the stack
  + Initialization: Must be explicitly initialized by programmer, else garbage
* static
  + **The default for variables declared outside of a function (file scope)**
  + **May be used on a local variable to retain its value while out of scope**
  + Lifetime: entire life of running program
  + Placement: initialized data segment (?)
  + Initialization: Initialized to zero
* extern
  + refers to an identifier declared and initialized in another compilation unit
  + Will be resolved by the linker, not the compiler
* register
  + A hint to the compiler that value should be in a CPU register (outdated)
* \_Thread\_local
  + Like auto, but associated with a thread of control

Copying Memory

* These workhorses copy chunks of memory from one location to another.
  + The destination storage must be allocated and big enough !
* The assignment operator ( = )
  + Works for scalar types
  + Works for enums
  + Works for pointers (copies the address only)
  + Works for structs and unions

As long as they DO NOT have pointer members

* + **Does NOT work for arrays or NULL-terminated strings**
* memcpy, memmove
  + General-purpose memory copy functions
  + Copies as though value was arrays of unsigned char
  + **Works on anything, as long as it's referenced through an address**
* strcpy, strncpy, strcat, strncat
  + The NULL-terminated string library functions
  + Works on NULL-terminated strings
  + Multibyte and wide-character variants too

Allocating Memory

* These functions allocate memory dynamically
* malloc
  + allocates memory
* calloc
  + allocates and zeroes memory
* realloc
  + expands (and maybe moves) a previously allocated memory block
* free
  + deallocates previously allocated memory
* aligned\_alloc (C11)
  + allocates aligned memory