Some abstractions 
and a peek at their implementation

Jeff Chase
Duke University
Some concepts for today

• Abstraction
  – **Examples**: heap vs. files

• OS structure
  – **Examples**: library vs. kernel

• Process structure
  – Virtual memory + address space

• Resource management
  – The “parking problem”: how to allocate variable-size blocks (“parking spaces”) from a shared space (“lot”)?

• Workload and performance
Abstraction

• Separate:
  – Interface from internals
  – Specification from implementation

• Abstraction is a double-edged sword.
  – “Don’t hide power.”

• More than an interface…

It’s a **contract** for how an object is to be used and how it is to behave, across many variations of its specific use and implementation.

We want abstractions that are simple, powerful, efficient to implement, and long-lasting.
Abstraction(s)

- A means to organize knowledge
  - Capture what is common and essential
  - Generalize and abstract away the details
  - Specialize as needed
  - Concept hierarchy

- A design pattern or element
  - Templates for building blocks
  - Instantiate as needed

- E.g.: class, subclass, and instance
A simple module (or “component”)

- A set of procedures/functions/methods.
- An interface (API) that defines a template for how to call/invoke the procedures.
- **State** is just data maintained and accessed by the procedures.
- A module may be a **class** that defines a template (**type**) for a data structure. A class may have multiple instances (**objects**) constructed from the template, each with its own state. The class procedures (**methods**) operate on a specific object.

*Abstract Data Type (ADT):* a module/object whose **state** is manipulated only through its **API** (Application Programming Interface).
OS Platform: A Model

Applications/services. Run as processes which may interact and serve one another.

Libraries/frameworks: packaged code used by multiple applications

OS platform: same for all applications on a system E.g., classical OS kernel

OS mediates access to shared resources. That requires protection and isolation.

[RAD Lab]
A simple C program

```c
int main()
{

}
```
Memory/storage is an abstraction

- Storage: an array of locations.
- Locations are numbered: each has an address that uniquely identifies the location.
- Each location can store a value.
- Very simple API
- Very simple contract
- Many variants
Today’s abstraction(s)

- **Variably-sized storage objects**
  - Heap blocks
  - Virtual memory segments
  - Files

- Each of these objects is a **named sequence of bytes** with various properties, similarities, differences.
  - How are they named?
  - How are they created and destroyed (released, freed)?
  - How does a program use them (e.g., access data at an offset)?
  - Can they grow? Persist? Be shared? How big/small?  
  - How does the system manage the space they occupy?
  - How does the system protect them?
A **storage object** is a sequence (or array) of numbered bytes.

We can store any kind of data in it: the bytes are “fungible”. That just means that any type of data may be encoded as a sequence of bytes.

The number of each byte (location) in the object is its **offset**.

A **pointer** references a location in the object.

$0x28 = 40$

We can also reference a location at an **offset** (displacement) from a pointer.

For memory, the ISA supports **base + offset addressing** directly.

`movq -32(%rbp), %rcx`
Abstraction: files

Program A

Library

open "a/b"
write ("abc")
read

Program B

Library

open "a/b"
read
write ("def")

system call trap/return

OS kernel
Files: hierarchical name space

- root directory
- mount point
- user home directory
- external media volume or network storage

- /usr
  - local
  - bin
- /home
  - jeremy
  - chris
- /mnt
  - msdos_dir
  - nfs_dir
Files as “virtual storage”

- Files have variable size.
  - They grow (when a process writes more bytes past the end) and they can shrink (e.g., see `truncate` syscall).

- **Most files are small, but most data is in large files.**
  - Even though there are not so many large files, some are so large that they hold most of the data.
  - These “facts” are often true, but environments vary.

- Files can be sparse, with huge holes in the middle.
  - Create file, seek to location X, write 1 byte. How big is the file?

- Files come and go; some live long, some die young.

- How to implement diverse files on shared storage?
VAS example (32-bit)

- The program uses virtual memory through its process’ **Virtual Address Space:**
- An addressable array of bytes…
- Containing every instruction the process thread can execute…
- And every piece of data those instructions can read/write…
  - i.e., read/write == load/store on memory
- Partitioned into logical **segments** (**regions**) with distinct purpose and use.
- Every memory reference by a thread is interpreted in the context of its VAS.
  - Resolves to a location in machine memory
Memory segments: a view from C

- **Globals:**
  - Fixed-size segment
  - Writable by user program
  - May have initial values

- **Text (instructions):**
  - Fixed-size segment
  - Executable
  - Not writable

- **Heap and stack:**
  - Variable-size segments
  - Writable
  - Zero-filled on demand
Dynamic memory (heap)

"Heap" memory, also known as "dynamic" memory, is an alternative to local stack memory....The programmer explicitly requests the allocation of a memory "block" of a particular size, and the block continues to be allocated until the programmer explicitly requests that it be deallocated. Nothing happens automatically. So the programmer has much greater control of memory, but with greater responsibility since the memory must now be actively managed. The advantages are...

*Lifetime.* Because the programmer now controls exactly when memory is allocated and deallocated, it is possible to build a data structure in memory, and return that data structure to the caller. This was never possible with local memory which was automatically deallocated when the function exited.

*Size.* The size of allocated memory can be controlled with more detail. For example, a string buffer can be allocated at run-time which is exactly the right size to hold a particular string. With local memory, the code is more likely to declare a buffer size 1000 and hope for the best....

[From *Essential C: Pointers and Memory*: http://cslibrary.stanford.edu/101]
Find it on the course web

Pointers and Memory

By Nick Parlante

Abstract
This document explains how pointers and memory work and how to use them—from the basic concepts through all the major programming techniques. For each topic there is a combination of discussion, sample C code, and drawings.

[From Essential C: Pointers and Memory: http://cslibrary.stanford.edu/101]
Programming the heap looks pretty much the same in most languages.…

- The heap is an area of memory available to allocate areas ("blocks") of memory for the program.
- There is some "heap manager" library code which manages the heap for the program. The programmer makes requests to the heap manager, which in turn manages the internals of the heap. In C, the heap is managed by the ANSI library functions malloc(), free(), and realloc().
- The heap manager uses its own private data structures to keep track of which blocks in the heap are "free" (available for use) and which blocks are currently in use by the program and how large those blocks are. Initially, all of the heap is free.
- The allocation function requests a block in the heap of a particular size. The heap manager selects an area of memory to use to satisfy the request, marks that area as "in use" in its private data structures, and returns a pointer to the heap block. The caller is now free to use that memory by dereferencing the pointer. The block is guaranteed to be reserved for the sole use of the caller — the heap will not hand out that same area of memory to some other caller. The block does not move around inside the heap — its location and size are fixed once it is allocated.
- Generally, when a block is allocated, its contents are random. The new owner is responsible for setting the memory to something meaningful.

From Essential C: Pointers and Memory: http://cslibrary.stanford.edu/101]
Heap abstraction, simplified

1. User program calls heap manager to **allocate** a block of any desired size to store some dynamic data.

2. Heap manager returns a pointer to a block. The program uses that block for its purpose. The block’s memory is reserved exclusively for that use.

3. Program calls heap manager to free (**deallocate**) the block when the program is done with it.

4. Once the program frees the block, the heap manager may reuse the memory in the block for another purpose.

5. User program is responsible for initializing the block, and deciding what to store in it. Initial contents could be old. Program must not try to use the block after freeing it.
Heap manager

Program (app or test)

Heap manager

OS kernel

Dynamic data (heap/BSS)

Stack

sbrk system call

"Set break (4096)"

alloc "0xA" alloc "0xB" free "0xA" "ok"

"break"

4096
Heap: dynamic memory

**Heap segment.** A contiguous chunk of memory obtained from OS kernel. E.g., with Unix `sbrk()` syscall

A **runtime library** obtains the block and manages it as a “heap” for use by the programming language environment, to store dynamic objects.

E.g., with Unix `malloc` and `free` library calls.

Allocated heap blocks for structs or objects. Align!
Using the heap (1)

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    char* cb = (char*) malloc(14);
    cb[0] = 'h';
    cb[1] = 'i';
    cb[2] = '!';
    cb[3] = '\0';
    printf("%s\n", cb);
    free(cb);
}
```

```
chase$ cc -o heap heap.c
chase$ ./heap
hi!
chase$
```
Using the heap (2)

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    char* cb = (char*) malloc(14);
    cb[0]='h';
    cb[1]='i';
    cb[2]='!';
    cb[3]=\0;
    printf("%s\n", cb);
    int *ip = (int*)cb;
    printf("0x%x\n", *ip);
    free(cb);
}
```

```
chase$ cc -o heap heap.c
chase$ ./heap
hi!
0x216968
chase$
```

Try:
Memory is “fungible”.

Pointers (addresses) are 8 bytes on a 64-bit machine.
Machines desire/require that an n-byte type is aligned on an n-byte boundary.

\[
n = 2^i
\]
Pointer arithmetic

```c
char* cb = (char*) malloc(14);
strcpy(cb, "hi!");

void* ptr = (void*) cb;
ptr = ptr + 2;
cb = (char*) ptr;
printf("%s\n", cb);

free(cb);
```

h=0x68
i=0x69
)!=0x21

chase$ cc -o heap3 heap3.c
chase$ ./heap3
???

chase$
Pointer arithmetic

```c
char* cb = (char*) malloc(14);
strcpy(cb, "hi!");

void* ptr = (void*) cb;
ptr = ptr + 2;
cb = (char*) ptr;
printf("%s\n", cb);

free(cb);
```

```
chase$ cc -o heap3 heap3.c
chase$ ./heap3
!
heap3(5478) malloc: *** error for object 0x7f92a9c000e2: pointer being freed was not allocated
Abort trap: 6
chase$
```
Using the heap (3)

```c
char* cb = (char*)malloc(14);  
strcpy(cb, "hi!");  
free(cb);  
/ *  
 * Dangling reference!  
 *  
 printf("%s\n", cb);  
 int *ip = (int*)cb;  
 printf("0x%x\n", *ip);  
/ *  
 * Uninitialized heap block!  
 *  
 char* cb2 = (char*)malloc(14);  
 printf("%s\n", cb2);  
```
Using the heap (4)

```c
char* cb = (char*)malloc(14);
strcpy(cb, "hi!");
free(cb);
/*
 * Dangling reference!
*/
printf("%s\n", cb);
int *ip = (int*)cb;
printf("0x%x\n", *ip);
/*
 * Uninitialized heap block!
*/
char* cb2 = (char*)malloc(14);
printf("%s\n", cb2);
```

```
chase$ cc -o heap2 heap2.c
chase$ ./heap2
hi!
chase$
```

```
h=0x68
i=0x69
!=0x21
0
```
WARNING

- These behaviors are **undefined**.
- Any program whose behavior relies on the meaning of a dangling reference is **incorrect**.
- For example, a change in the allocation policy of the heap manager could result in different behavior.
- Can a program stay safe from dangling references by just never calling **free**?
Memory leaks in the heap

What happens if some memory is heap allocated, but never deallocated? A program which forgets to deallocate a block is said to have a "memory leak" which may or may not be a serious problem. The result will be that the heap gradually fill up as there continue to be allocation requests, but no deallocation requests to return blocks for re-use.

For a program which runs, computes something, and exits immediately, memory leaks are not usually a concern. Such a "one shot" program could omit all of its deallocation requests and still mostly work. Memory leaks are more of a problem for a program which runs for an indeterminate amount of time. In that case, the memory leaks can gradually fill the heap until allocation requests cannot be satisfied, and the program stops working or crashes. Many commercial programs have memory leaks, so that when run for long enough, or with large data-sets, they fill their heaps and crash. Often the error detection and avoidance code for the heap-full error condition is not well tested, precisely because the case is rarely encountered with short runs of the program — that's why filling the heap often results in a real crash instead of a polite error message.

From Essential C: Pointers and Memory: http://cslibrary.stanford.edu/101]
Heap: parking lot analogy

- Vehicles take up varying amounts of space, but they don’t grow or shrink, and they don’t have “holes”.
- They come and go at arbitrary times.
- Space allocation is exclusive.
- Vehicles are parked in contiguous space: must find a block that is available (free) and big enough.
Heap blocks are contiguous

The storage in a heap block is contiguous in the Virtual Address Space. The term block always refers to a contiguous sequence of bytes suitable for base+offset addressing.

C and other PL environments require this. E.g., C compiler determines the offsets for named fields in a struct and “bakes” them into the code.

This requirement complicates the heap manager because the heap blocks may be different sizes (must use Variable Partitioning). How to pack them into the available space in the heap region?
Variable Partitioning

Variable partitioning is the strategy of parking differently sized cars along a street with no marked parking space dividers.

1

2

3

Wasted space
external fragmentation
Heap manager policy

• The heap manager must find a suitable free block to return for each call to malloc().
  – No byte can be part of two simultaneously allocated heap blocks! If any byte of memory is *doubly allocated*, programs will fail. *We test for this!*

• A heap manager has a *policy algorithm* to identify a suitable free block within the heap.
  – Last fit, first fit, best fit, worst fit
  – Choose your favorite!
  – Goals: be quick (first-fit), and use memory efficiently (others)
  – Behavior depends on *workload*: pattern of malloc/free requests

• This is an old problem in computer science, and it occurs in many settings: *variable partitioning.*
Dynamic Storage-Allocation Problem

How to satisfy a request of size $n$ from a list of free holes.

- **First-fit**: Allocate the *first* hole that is big enough.
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

First-fit and best-fit [generally] better than worst-fit in terms of speed and storage utilization.

50-percent rule: Given $N$ allocated blocks another $1/2N$ will be lost due to fragmentation $\Rightarrow 1/3$ of memory lost.
Best fit, worst fit, first fit

- People used to study the relative merits of these algorithms for variable partitioning. Let’s not.

- Which is best at reducing fragmentation? “It depends.”

- “What does it depend on?” **Workload**: the particular pattern of requests (e.g., malloc/free) that we receive.
  - Sizes requested
  - Order of malloc/free

- In general, we won’t know the workload in advance, and we avoid assumptions about it that limit generality.

- But if we **do** know in advance, then we can optimize.
Fixed Partitioning

Wasted space
internal fragmentation
Heap manager: “the easy way”

```c
static void* freespace = NULL;

void* dmalloc(size_t n) {
    if (freespace == NULL)
        freespace = (void*)sbrk(4096);
    void* bp = freespace;
    freespace = freespace + ALIGN(n);
    return bp;
}
```

Why is this a bad way to implement a heap manager?
“The easy way” isn’t good enough

- “The easy way” approach can’t free the blocks!
  - It doesn’t track the borders between blocks, or their sizes.
  - It allocates only from the “top” of the heap, never from the middle, so it can’t reuse freed blocks anyway.
  - It’s a stack! It is fast and easy for local variables, but it’s not good enough for a heap:

It can only free space at the front!

Admittedly this ferry picture illustrates FIFO (first-in-first-out: queue) rather than LIFO (last-in-first-out: stack), but you get the idea. It’s restricted, and I want to come and go as I please, i.e., allocate space when I want and free it when I want.
Programs run as independent processes.

Protected OS kernel mediates access to shared resources.

Each process has a private virtual address space and one or more threads.

Protected system calls...and upcalls (e.g., signals)

Threads enter the kernel for OS services.

The kernel code and data are protected from untrusted processes.
**Man mmap**

#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flag, int filedes, off_t off);

Returns: starting address of mapped region if OK, MAP_FAILED on error

The mmap() system call causes the pages starting at addr and continuing for at most len bytes to be mapped from the object described by fd, starting at byte offset offset. If offset or len is not a multiple of the pagesize, the mapped region may extend past the specified range. Any extension beyond the end of the mapped object will be zero-filled.

The addr argument is used by the system to determine the starting address of the mapping, and its interpretation is dependent on the setting of the MAP_FIXED flag. If MAP_FIXED is specified in flags, the system will try to place the mapping at the specified address, possibly removing a mapping that already exists at that location....
Another way

• Java’s new operator calls malloc “under the hood”.
  – Allocate a heap block for each new object.
  – How big should the heap block be? Who decides?
  – Java initializes the block by automatically calling a type-specific constructor defined by the program.
  – C++ is the same.

• But in Java there is no free (or delete)!
  – No dangling references!

• How does Java avoid memory leaks?
Digression: Garbage collection

• Java has new but no free (or delete)!
  – No dangling references!
• Q: How does Java avoid memory leaks?
• A: automatic garbage collection
• Java is strongly typed: e.g., it tracks and controls pointers, and it knows where all of them are and what object types they reference.
• Java knows if an object has no references pointing to it, and it calls free for you!
• But Java has no pointer arithmetic: your program cannot manipulate virtual addresses as numbers, as in C.
Digression: reference counting

• How does a program know when to free an object?
  – **Option 1**: careful control over ownership: always know what part of the program is “responsible” for the object.

• But what if an object could be in use in multiple places? E.g., referenced from multiple data structures?
  – **Option 2**: “Last one out turn out the lights.” Free the object when the last reference is destroyed/discarded.

• But how to know if a given reference is the last?
  – Answer: **reference counting**.
Reference counting

Used in various applications and programming language environments, and in the kernel, e.g., Unix file management.

- Keep a count of references to the object.
- Increment count when a new reference is created (shallow copy).
- Decrement count when a reference is destroyed.
- Free object when the count goes to zero.

[http://rypress.com/tutorials/objective-c/memory-management.html]
Concepts

- Reference counting and reclamation
- Redirection/indirection
- Dangling reference
- Binding time (create time vs. resolve time)
- Referential integrity

Note: these are foundational concepts that will reappear in different forms in various systems we discuss this semester. You should understand each general concept and recognize its instances when you see them!

For example, for each kind of naming system: what is the object? How is the object named? What operations bind the name to the object? What operations increment/decrement the refcount (if there is one)?
A **file** is a named, variable-length sequence of data bytes that is **persistent**: it exists across system restarts, and lives until it is removed.

**Unix file syscalls**

```c
fd = open(name, <options>);
write(fd, “abcdefg”, 7);
read(fd, buf, 7);
lseek(fd, offset, SEEK_SET);
close(fd);
creat(name, mode);
fds = open(name, mode, O_CREAT);
mkdir (name, mode);
rmdir (name);
unlink(name);
```

An **offset** is a byte index in a file. Programs may read and write files **sequentially**, or **seek** to a particular offset and read/write there. This is called a “logical seek” because it seeks to a particular location in the file, independent of where that data actually resides on storage (it could be anywhere).
Unix file I/O

char buf[BUFSIZE];
int fd;

if ((fd = open("../zot", O_TRUNC | O_RDWR) == -1) {
perror("open failed");
exit(1);
}
while(read(0, buf, BUFSIZE)) {
if (write(fd, buf, BUFSIZE) != BUFSIZE) {
perror("write failed");
exit(1);
}
}
Unix file commands

- Unix has simple commands to operate on files and directories (“file systems”: FS).
- These commands are ordinary C programs that come with the system, for you to run from the command line.
- Some just invoke one underlying kernel syscall.
  - `mkdir`
  - `rmdir`
  - `rm` (unlink)
  - “ln” and “ln -s” to create names (“links”) for files
- What are the commands to create a file? Read/write a file? Truncate a file?