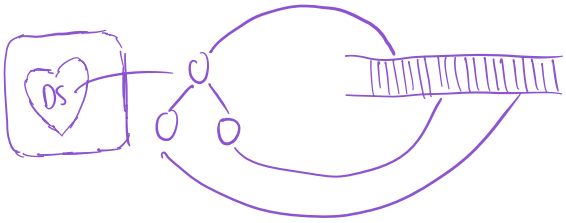
MEMORY MANAGEMENT

WHY MEM MGMT in OS?

- see "Motivation to OS" again



WHAT exactly is MEM MGMT?

- overriding C++ new/delete

THE M$ QNS:
WHY do we want to reinvent the ????ing C++ new/delete ???

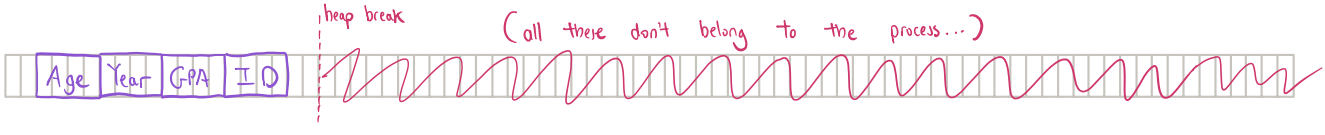
omo z...
lagg...
crash...

THE M ANS:
consider the

```
struct Student {
    int Age;
    long Year;
    float GPA;
    long ID;
}
```

Q: WHAT is sizeof(Student)? (assume 32-bit WIN)
A: 4 bytes * 4 = 16 bytes

```
Student* s1 = new Student();
```




```
Student* s2 = new Student();
```

this will trigger...

① allocate() from proc heap
- find empty slot
- reduce fragmentation
- etc...

② OUT_OF_MEM_ERR

③ SYS_CALL_TRAP (context switch) — only SYS has oversight across procs

USER MODE

KERNEL MODE

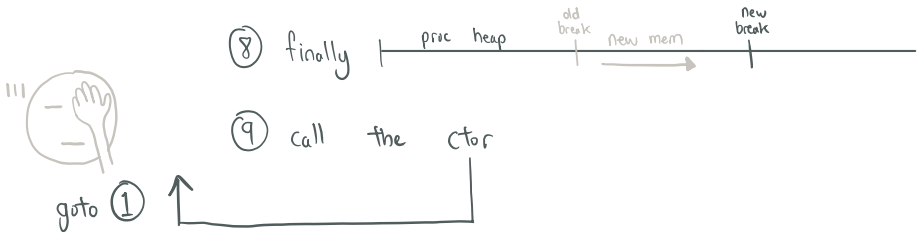
④ lock()
- prevent 2 procs alloc same mem

⑤ allocate() more mem to proc

⑥ BUT maybe unlucky... hit OUT_OF_RAM_ERROR

⑦ swap old/unused pages to HDD

tl;dr
THIS is very BAD



this is WHY we have ...

CUSTOM ALLOCATORS

MAIN IDEA:

① alloc a "chunk" (a.k.a. pages) of mem at one go

```
ObjectAllocator* oa = new ObjectAllocator(sizeof(Student), config);
```

config = { newdel, objects, pages, debug, padbytes, header, alignment }

ObjectSize_ = 16 bytes
ObjectsPerPage_ = 4

PageSize_ = 4 * 16 + sizeof(void*)
= 68 bytes

4 bytes (for 32-bit)

② when user `Student* pStudent = static_cast<Student*>(oa->Allocate());`

return 1 block from a page instead of calling `new Student();`

③ when user `oa->Free(pStudent)`

"put" back the block into the free list

④ ... reuse blocks by adhering to ② and ③

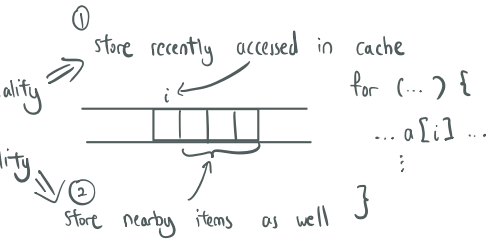
Q: Show what happens to the ptrs in mem.

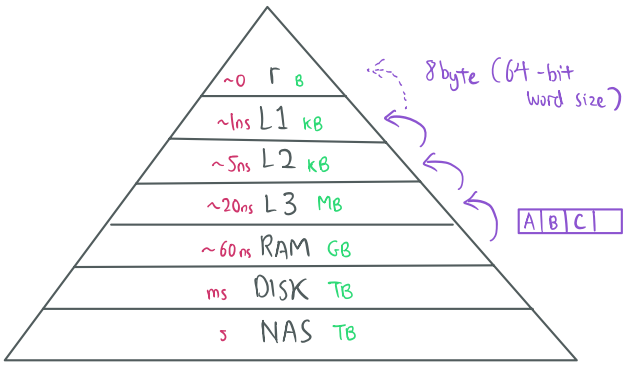
now let's look deeper at

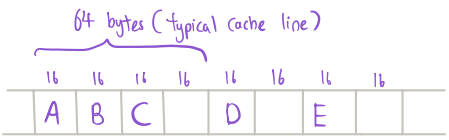
WHY CUSTOM OAs CAN BE BETTER THAN BUILT-IN

① performance boost with LOCALITY OF REFERENCE

- Temporal Locality ① store recently accessed in cache
- Spatial Locality ② store nearby items as well







Q: Describe what happens when

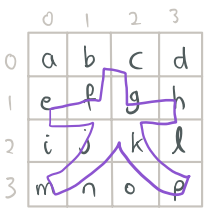
do(A), do(C), do(A), do(B), do(D)

miss — fetch cache line [A|B|C|]

hit — hit — hit

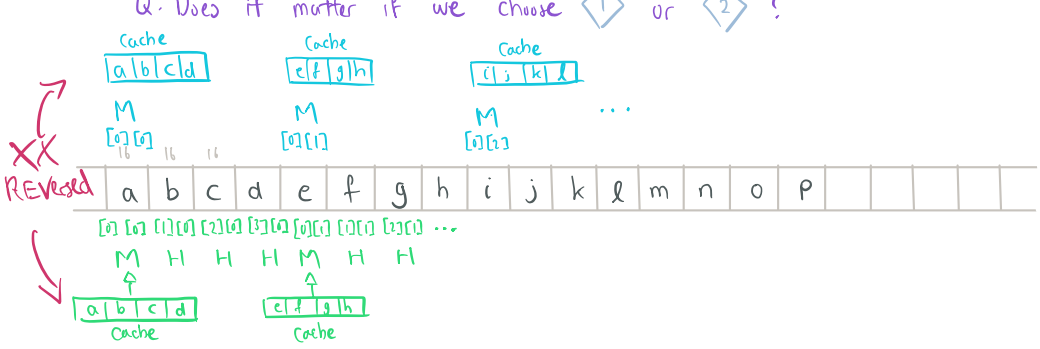
miss — fetch cache line [D|E|]

now consider this bmp operation ...




```
// bmp[1][2] = g
Pixel bmp[4][4]; // sizeof(Pixel) = 16, cacheline = 64
for (auto i=0; i<4; ++i)
    for (auto j=0; j<4; ++j)
        clear(bmp[i][j]); //<1>  vs  clear(bmp[j][i]); //<2>
```

Q: Does it matter if we choose <1> or <2> ?



and time for Misses >>> Hits (much slower)

⇒ <1> is better <2>

② customized to specific app needs

- KNOWN OBJ SIZES
  - fixed entity types
  - max objs in scene
- ALLOC PATTERNS
  - same sequence between levels
  - fixed durations

③ MEM DEBUGGING

- count allocs / deallocs (detect leaks)
- track alloc sizes
- time between allocs
- etc...

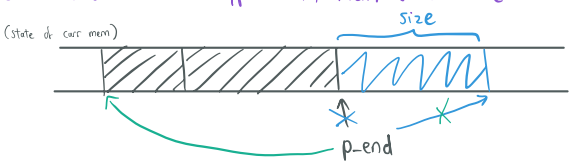
So what are some common

CUSTOM ALLOCATOR TYPES ?

LINEAR ALLOCATOR

```
oa -> Allocate (size);
oa -> FreeAll ();
```

Q: Show what happens in mem after the above stmts are ran...



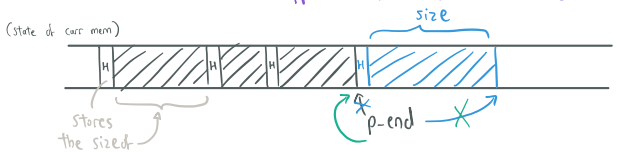
- ✓ fast
- ✓ insignificant space overhead
- × no individual frees

} Suitable for e.g. gfx screen buffer
- clrScr() every frame

STACK ALLOCATOR

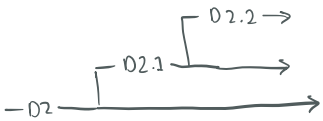
```
oa -> Allocate (size);
oa -> Free ();
```

Q: Show what happens in mem after the above stmts are ran...



- ✓ still fast
- ✓ slight space overhead
- ✓ can individually Free() BUT only LIFO

} Suitable for e.g. hierarchical game levels

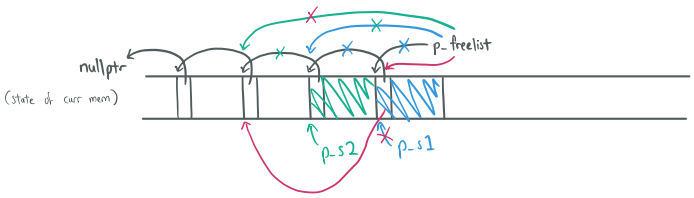


POOL ALLOCATOR

after alloc "chunk", pre-split into blocks, i.e. populate the free list

```
Student* p_s1 = static_cast<Student*>(oa->Allocate());
Student* p_s2 = static_cast<Student*>(oa->Allocate());
oa->Free(p_s1); // return to free list for reuse
```

Q: Show what happens in mem after the above stmts are ran...



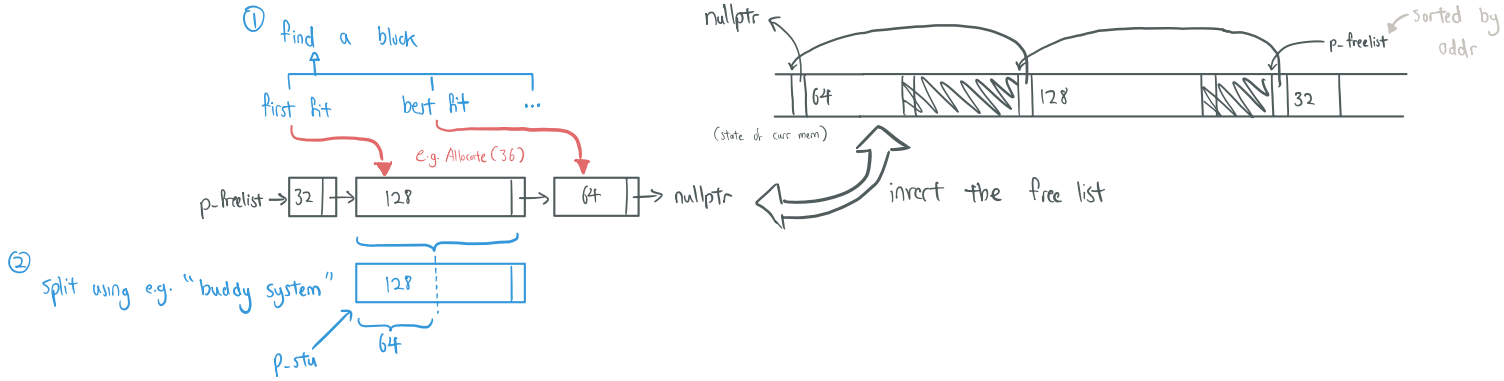
- ✓ still fast
- ✓ fixed-size blocks
- × some space overhead

} suitable for e.g. game entities
- roughly same size
- frequent arbitrary alloc/frees

FREELIST ALLOCATOR (a.k.a. pooling with dynamic sizing)

(ignoring the casts...)

```
p_stu = oa->Allocate(size);
```



`oa->Free(p_stu)` ① link back to free list (in order of address)

② merge if contiguous (coalescence)

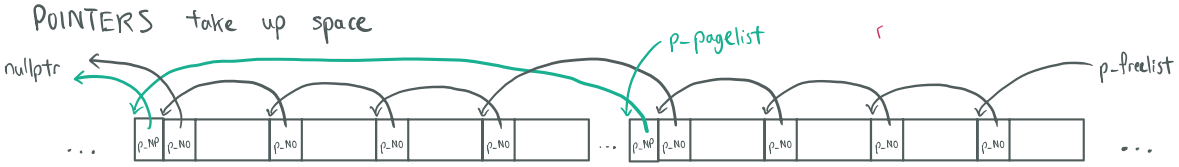
✓ general purpose
× not as fast (still faster than malloc)

Suitable for ANYTHING
- BUT always consider more specialized 1st
- we can also note that as it gets more generic, speed decreases.

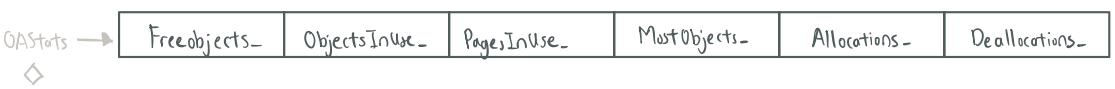
visit github.com/mtrebi/memory-allocators to see some benchmarks of the above

MEM ALLOCATOR OVERHEAD

POINTERS take up space



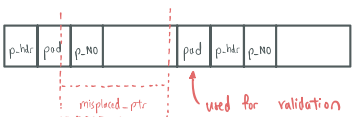
STATISTICS take up space



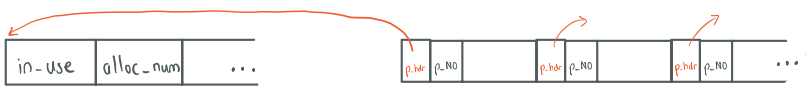
DEBUGGING
- e.g. `if (Allocations_ == Deallocations_) noLeaks = true;`

IMPROVE OA DESIGN
- e.g. `if (MostObjects_ > threshold) ++ObjectsPerPage_;`

and then padding (for validation)



and then there're HEADERS...



ALIGNMENT also takes up space

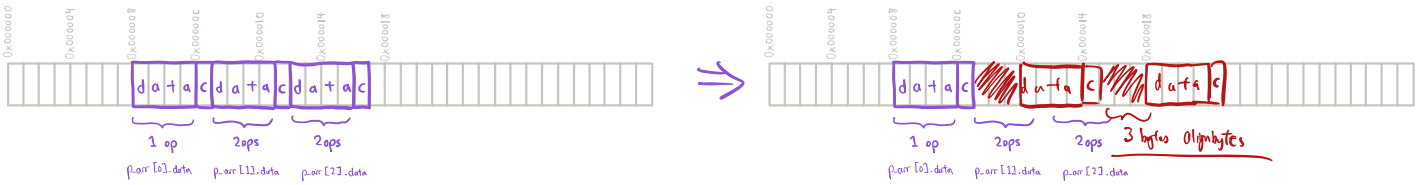
e.g.
```
struct Intcy {
    int data; //4
    char c; //1
}
Intcy* p_arr = new Intcy[3];
```

atomic (smallest) unit of operation

$\frac{32}{8\,bit} = 4$ byte word

Q: Show the above array in mem below ... (assume 32-bit archi)

Q: How many atomic operations to access each element of the array?



Q: WHAT is the soln to this problem?

A: Add alignbytes until data within boundaries.

lastly, a note about **FRAGMENTATION**

2 types

INTERNAL

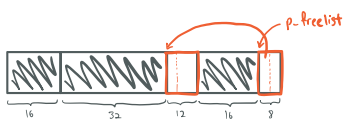
e.g. a fixed-size pool allocator of 16-byte blocks

```
p1 = static_cast<SixteenByte*>(oa->Allocate());
p2 = static_cast<SixteenByte*>(oa->Allocate());
p3 = static_cast<FourByte*>(oa->Allocate());
p4 = static_cast<SixteenByte*>(oa->Allocate());
```

(fragment is within allocated mem)

EXTERNAL

e.g. a freelist allocator



what if now we want

```
p_0 = static_cast<SixteenByte*>(oa->Allocate());
```

(fragment is in free mem)