```
void f() {
  Barrier b{2};
  thread([&b] {
    .
    .
    .
    b.sync();
  });
  b.sync();
}
```

race condition: b will be deleted once end of f is reached
↳ can't use b in thread

```
void f() {
  Barrier* p = new Barrier(2);
  thread([p] {
    .
    .
    .
    p->sync();
  });
  p->sync();
}
```

use pointer to malloc Barrier
↳ will persist forever
but...
how do we delete?
⇓
need garbage collector

How do we know when it's safe to delete?

- You can delete data on the heap when there's nothing pointing to it
  - ↳ how does the computer know?
  - ↳ what if the data points to someone else?

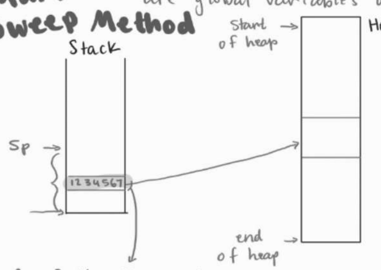
↓ Solution

Tracing Garbage Collector

How can you find all pointers?

- global variables
- you know where all the stacks are
- The things in the heap
  - ↳ These don't have names but... their pointers have names and are global variables or are on stack

Mark & Sweep Method

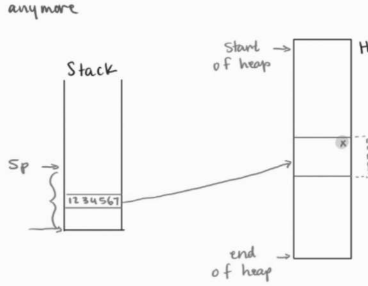


See if this thing looks like a pointer

BUT... what if it's just a random number?
↳ check if the number is in the range of the heap
↳ could check the metadata to see if its malloced

more conservative is good!
↳ better to leave something unused than to delete something being used

* NOTE: There's nothing to prevent someone from adversarially creating a fake pointer
↳ Thats why Tracer isn't used very much anymore



use a bit to mark it as used
↳ There's 1 malloced bit and 1 used bit

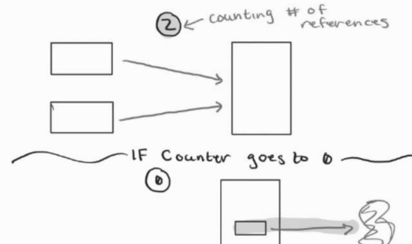
* This is the "Mark" stage

* The sweep stage:
sweep the heap to check the used bit
↳ if unused → free
reset used bit

* Note: works very well in managed system but not in an unmanaged system
↳ can't manipulate pointers (Ex. in Java)

* What if... Someone manipulateds a pointer?
↳ ex. storing an offset instead of the pointer
↳ manipulating for security purposes
* This method would FAIL

Counting Garbage Collector (BETTER METHOD)



- you need some mechanism to track everything that happens to a pointer

```
class SmartPointer {
  private:
    T* ptr;
  public:
    SmartPointer(T* ptr) {...}
    ~SmartPointer() {...}
    T* operator->() {return ptr;}
}
```

This needs to be private so no one can use it directly

how it's managed
↳ there's generally a rule that once you create smartptr you never look at pointer again

override the -> operator

```
Struct Person {
  int* age;
}
  .
  .
  .
{
  SmartPointer<int>   p{new int(10)}
  print(p->age);
}
```

Note: There's no way to enforce people using Smart Pointer

translates to p.ptr.age

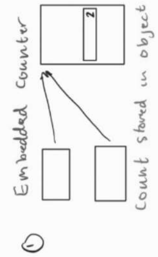
when you reach the end... will automatically call SmartPointer's deconstructor
↳ can decrement the counter

*SmartPointer is NOT a security feature
↳ it's to help you from messing up

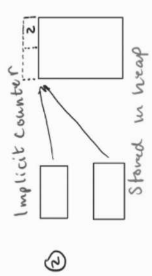
```
class SmartPointer {
  private:
    T* ptr;
  public:
    SmartPointer(T* ptr) {...}
    ~SmartPointer() {...}
    T* operator->() {return ptr;}
    SmartPointer(Smartpointer<T> & src) {...}
}
```

making copy of pointer → increment counter

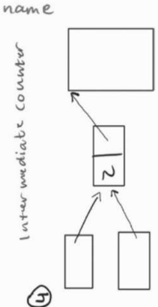
variable name

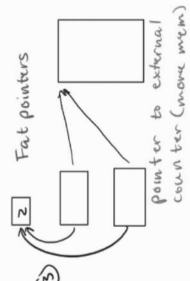


have to plan ahead of time for the object to store the reference counter
↳ Can't have reference counter for primitives
↳ need to store pointer outside object



- pointers are twice as large
- need to allocate 2 things (object AND counter)





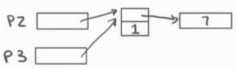
- Also needs to allocate 2 things
- Acts like embedded counter but doesn't need you to plan for your objects to have a counter
- Also makes it so that you can't call * on the object to access the value

Constructor Types

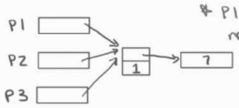
→ Default constructor `StrongPtr<int> p1{};`

→ Pointer Constructor `StrongPtr<int>p2{new int(7)};`

→ Copy Constructor `StrongPtr<int> p3{p2};` → Can also be `StrongPtr<int> p3=p2`
↳ Makes a new object to point to



→ Assignment operator (p1 = p2) p1 = p2
↳ p1 points at same object as p2



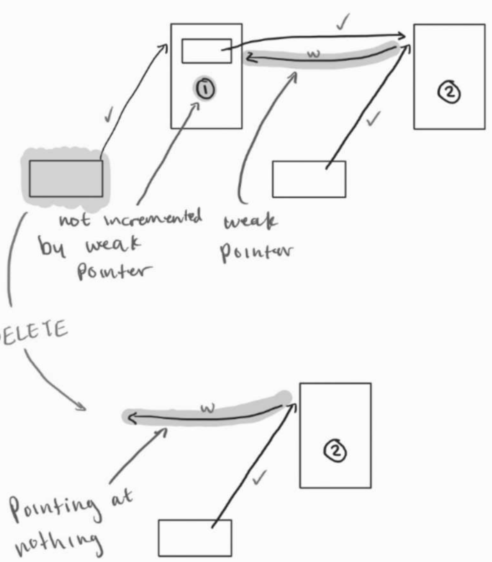
* p1 already exists and you're reassigning to a different object

→ Deconstructor
↳ deletes objects that go out of scope
↳ invoked by compiler
↳ you shouldn't have to call the deconstructor
↳ Now that we're in C++... you don't want to be calling free...
↳ this will stop the constructor from calling deconstructor

NOTE: If you ever try taking the pointer of a StrongPtr you're probably doing something

Weak Pointer

- Doesn't increment the counter
- you need to implement logic to know when to use a strong vs weak pointer



How is weak pointer different from normal pointer:
weak pointer needs to be able to check if its pointing at smth

```
void f(weakPointer<Person> p) {
    StrongPointer <person> s = p.promote();
    if(s != nullptr){
        s.name();
    }
}
```

could be invalid

needs to check if p points to something
↳ if no → return null
if yes → make strongptr

needs to be atomic

need to check

*NOTE: weak pointer should NOT have a → implementation
↳ Implemented by programmer

```
StrongPointer <Barrier>   b{new Barrier};

thread ( [b] {
    :
    b.sync();
})

b.sync();
```

making a copy of b
↳ same intermediate object
different pointer value

Note: you cannot use one smart pointer in multiple places at once
↳ Ex. don't make it global and access it in multiple threads
↳ need to keep access to the intermediate object atomic

FOR YOUR OWN SANITY!!

Throw away a raw pointer as soon as it's wrapped around in a smart pointer??

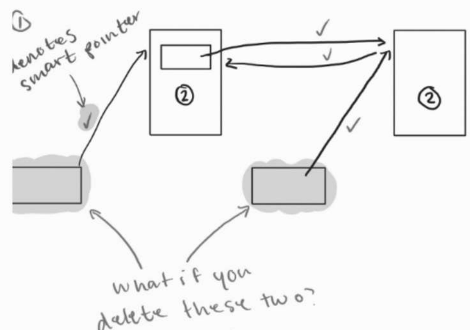
- you should always create your smart pointer like this:

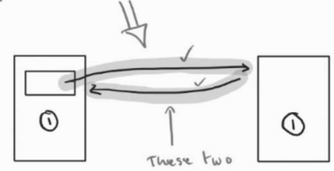
`SmartPointer <int> p {new int(11)};`

instead of

`auto val = 11;` ← you may accidentally use in the future
`SmartPointer <int> p {val};`

2 Failures:



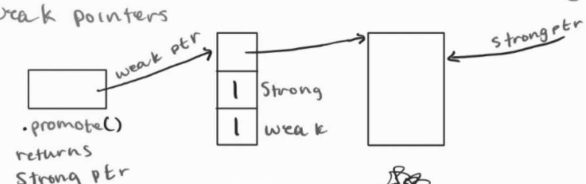


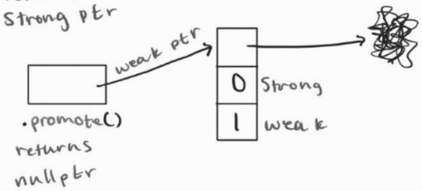
These two pointers keep each other alive but the objects are unreachable

Solution:

Implement a tracing collector AND counting collector
has to run alot less

How to write promote?
change the reference counter to count strong and weak pointers

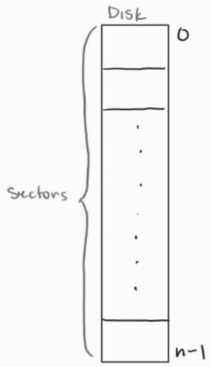




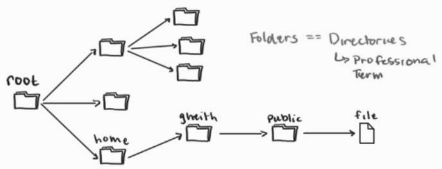
*Delete object when StrongPtr == 0
Delete counter when StrongPtr == 0 and weakptr == 0

How To Build File Systems

Disk:
- A linear list of sectors
  - ↳ Ideally without gaps
- have to move head to go read different sector
- ~10 ms to read something
- we don't typically get to pick sector sizes
  - ↳ most popular: $2^5 = 512\,B$
    $2^{12} = 4\,kB$



What does a File system Look Like?
- Most modern systems look like a tree
  - ↳ Hierarchical system



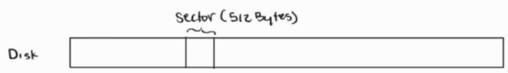
📁 = directory
🗋 = file

metadata:
- sizes
- where is it
- names
- security info
- times
- data

How do we make this look like this?

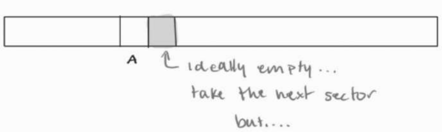
This is what the file system does

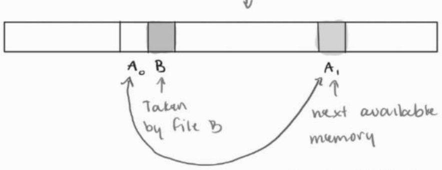
Lets say we have a disk where each sector is 512 bytes



- You can only read memory in units of sectors
  - ↳ EVEN IF you want less

How can you write more efficiently?
What if you start writing an need to get another sector?





how do we connect together?

Note: this fragmentation isn't allowed for malloc because for malloc you pass a pointer for the data to be accessed directly...
with disk you are accessing data through the file system (not directly) So it can be fragmented

ArrayList

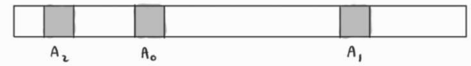
File block	Block #
0	100
1	200
2	50
⋮	

This also needs to be stored on the disk (file system needs to be self contained)

*What if... the table is really long longer than a block?
↳ make a table for the table!
↳ end up with a tree of pointers until one table fits in a block

* THIS IS THE CURRENT DAY SOLUTION!!

How do we glue the sectors together?



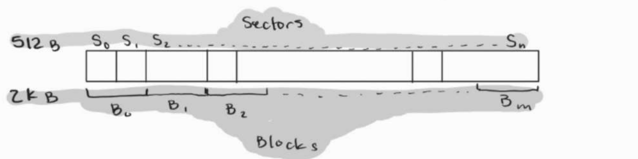
- use metadata!
  $|MD|$ = metadata size   # of sectors = $\frac{\text{File size}}{\text{Sector size}}$

  $|MD|$ proportional to # of sectors
  ↳ inversely proportional to sector size

- To maximize data storage → increase sector size
- But! Larger sector size = Internal Fragmentation
  - ↳ unused space within a sector
  - ↳ Proportional to sector size
- If its uniformly distributed... Internal fragmentation = $\frac{1}{2}$
  - ↳ $\frac{1}{2}$ a sector is Unused

!! Solution (to optimize metadata) !!

- The disk works in sectors but the file system works in Blocks

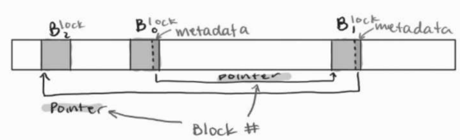


↳ Allows file system to determine optimal block size
↳ the file system will ALWAYS deal with a whole block at a time

*For this class: Block sizes will ALWAYS be MORE than one sector

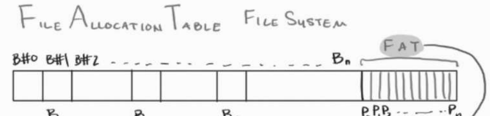
* In modern file systems Blocks have variable size
↳ Not in this class

Now...



- This pointer method is SLOW if we need to append a block
  - ↳ need to traverse the whole linked list
- This only method that uses pointers like this:

File Allocation Table File System



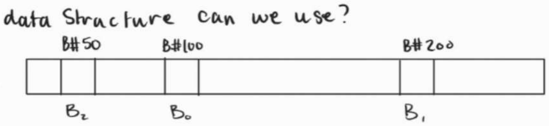
- FAT table was in memory
  ↓
- this was better because you were traversing in memory instead of in the disk
  - ↳ Much faster

contains a table of pointers; B#1's next Block # was in P1 in the FAT

*As disks got larger this method became much less valuable ⇒ FAT becomes too fat

*Never use FAT in larger systems ⇒ OK for very small things (like cameras)

What data Structure can we use?

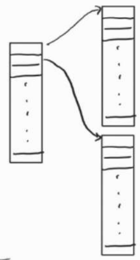


Suppose you create a language where Arrays can only have max 10 items?
Make an Array of Arrays to create longer Arrays!

How would you find item 37?
use base 10!

37

index of outer array — index of inner array



* called a Radix Tree!

* Same idea in file system EXCEPT not base 10 -- base {block_size}

Reminder: |Block|$_B$ ← vertical bars indicate size of something

Suppose:

$|Block|_B = 2KB = 2^{11} B$ (NOTE: 2KB is hypothetical; could be something else)
$|Block Number| = 32b = 4B$ (↑ same with 32b)
↳ the numerical value used to find a block (the "index")

How many Blocks can we store in a block?
How many Block numbers?

$$\frac{|Block|}{|BlockNumber|} = \frac{2^{11}}{4} = 2^9 \text{ entries} = 512 \text{ entries}$$

LBN — Logical Block # (Block # relative to file)
PBN ← Physical Block # (Block # relative to Disk)

LBN	PBN
0	15
1	72
2	12
⋮	⋮
n	

How big can a File be in a 1 Level Tree

$$0 \le |File|_{Blocks} \le 512 \text{ Blocks}$$
$$\le 512 \cdot |Block|_B$$
$$\le 2^9 \cdot 2^{11} B$$
$$\le 2^{20} B$$
$$\le 1MB$$

⇓

$$\text{max \# of Blocks for File} = \left(\frac{|Blocks|_B}{|BlockNumber|_B}\right)^{n}$$

n — Tree Depth

$$\text{max File size (B)} = \left(\frac{|Blocks|_B}{|BlockNumber|_B}\right)^{n} \cdot |Blocks|_B$$

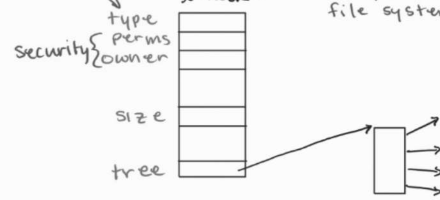
What if the File is VERY Small?
• We don't need to build out the whole tree
⇓
Things can be null ⇒ Build a sparse tree

---

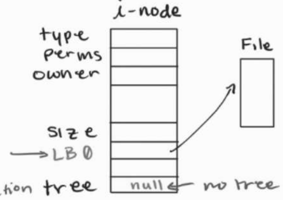
metadata: use i-node to store metadata
ext2...

Can be file, directory, etc → type

i-node ← inode stored in i-table
↳ a specified place in file system



IF... your file is small... you don't need a tree
↳ * Most files are small...



Stores the First Block Location → LB 0
no tree for small file

How Big is the i-node?
• 128 B

All storing Block pointers
• if |Block Number| = 4 then you can store 15 Block pointers
• (Based on some calculations)
12 BP can be used for the first 12 blocks of actual File Data
↳ covers MOST files

3 Level Tree
2 Level Tree
1 Level Tree (starting at LBP # 13

* Note: This method is more efficient than JUST a tree because files often grow and shrink
↳ this method allows you to simply delete or create a Tree instead of restructuring trees to add/delete levels