Warm-Up Problem

Please do your Teaching Evaluations! Go to https://perceptions.uwaterloo.ca

CS 241 Lecture 23

Heap Management

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Heap Management

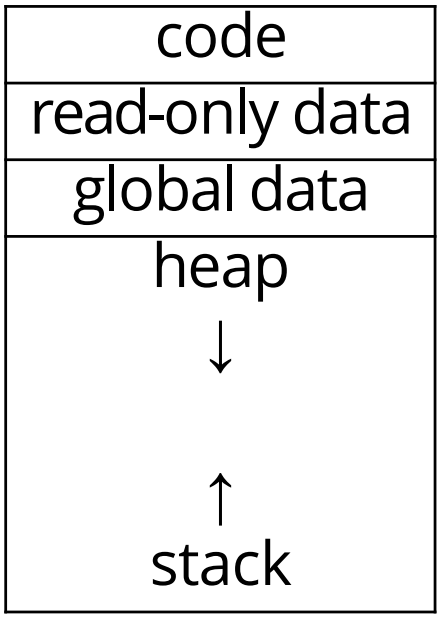
- In our course, we gave you a library to deal with all of memory management features which included init, new and delete.
- This allows for data to exist in memory that is out of scope, that is, out of the boundaries of your stack frame.
- How do we manage this memory?

Heap Management

- The memory not on the stack is either in code (but this is static and not allocated later) or *on the heap*. The init procedure initializes a heap for us to use.
- Much more problematic than a stack to take care of. Stacks are nice and ordered, and, well, stack-like. Calls can be made to heaps using delete or new in arbitrary orders, so we can't simply push and pop memory.

Heap Management

Our World:



Note that our stack contains a pointer to where the heap is.

Aside: Read-Only/Global Data?

- We just added an extra bit to the usual diagram…
- But we didn't actually add anything. Remember, code is also just data. You can store data in your code.
- In C, it's common to separate out static, global information from code, but it's all just "the stuff before the dynamic heap"

Heaps

- How exactly does a heap work?
- We have a variety of implementations that we will discuss ranging in practicality and utility.
- We'll start with a simple problem and make things more complex as we go
- Our first example makes use of the fact that heap management is easy (indeed, trivial) if you never allow for delete

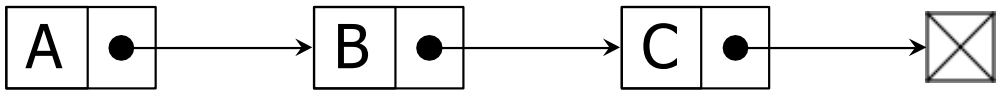
Example 1: No Reclamation of Memory and Fixed Blocks

- After init, you get two pointers, one to the start of memory on the heap and one at the end.
- Initialization is O(1).
- Allocation is also O(1).
- Never delete so this is fine.
- Clearly not our best choice; we run out of memory quickly if we don't reuse reclaimed memory. (Think: actual garbage waste)

Example 2: Explicit Reclamation and Linked List of Fixed-Sized Blocks

- We can keep the fixed size idea but this time, we keep track of a free list (a linked list of free memory blocks) and we can allocate from this linked list.

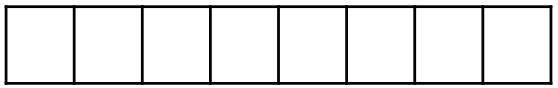
where *A*, *B*, and *C* are the starting memory addresses of the free blocks and then the associated linked list:



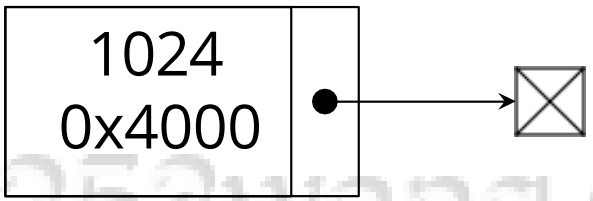
- Works only because you allow a single block of memory each time. You keep track of the memory in the free list and allocate from the free list first if memory is being requested.

Example 3: Variable-Sized Blocks

- Idea: We once again used a linked list but here our linked list will store a number of bytes, where those bytes can be found and the next node.
- Init: Start with the entire heap being free. As an example, suppose we have 1024 bytes:
  - Memory (first address is at 0x4000):

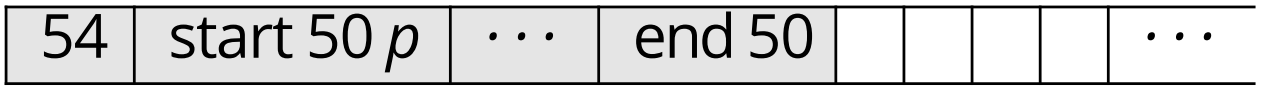


  - Free Linked List:

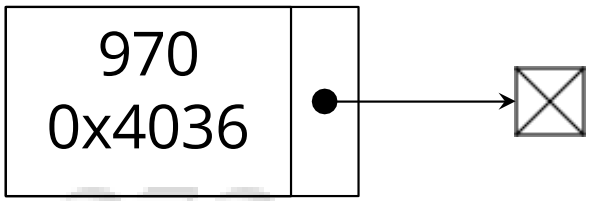


Next

- Let's say we want to allocate 50 bytes. What we will do is allocate 54 bytes
  - The first 4 bytes are the size of the block (an integer), and the rest is the requested bytes. We need this bookkeeping because delete doesn't take a size! We return a pointer to the start of the 50 bytes.
- Memory:

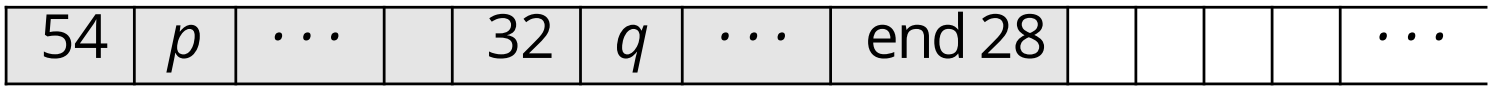


- Free List:

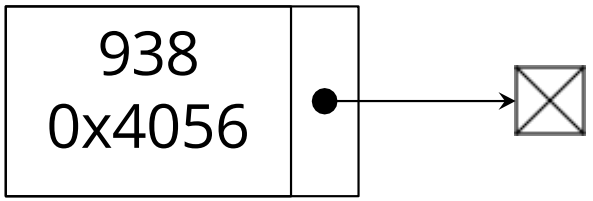


Next

- Next, we want to allocate 28 bytes. What we will do is allocate 32 bytes. We return a pointer to the start of the 28 bytes.
- Memory:

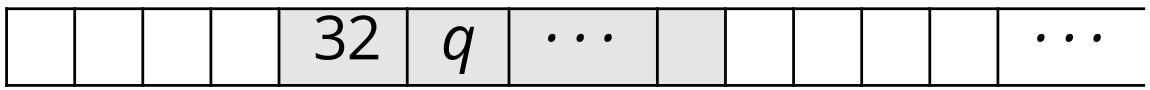


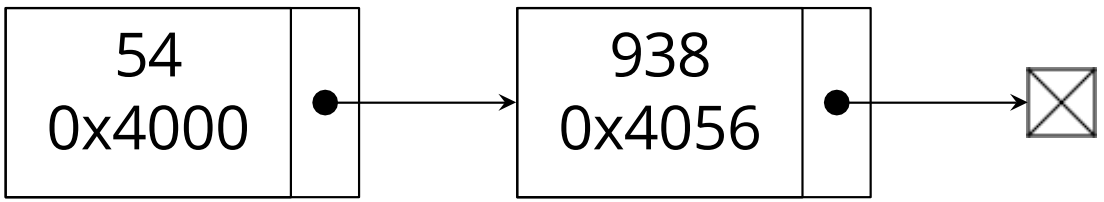
- Free List:



Free

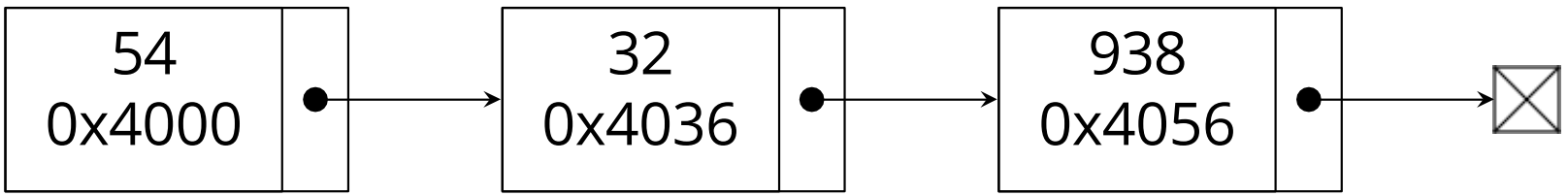
- Next, we free the 50 bytes!





Free

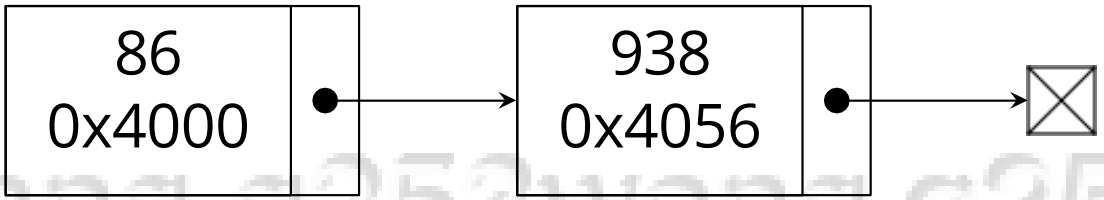
- Lastly, we free the other 32 bytes



Consolidate

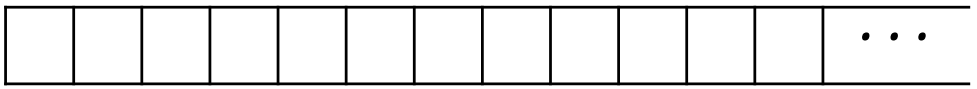
- Now, we can do some consolidation. Notice that
- 54 + 0x4000 = 0x4036 and so the first two nodes in our linked list can collapse to a single node:
- Memory:

- Free List:

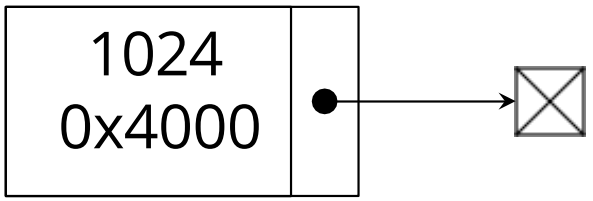


Consolidate

- Again, we can consolidate. Notice that 86 + 0x4000 = 0x4056 and so the first two nodes in our linked list can collapse to a single node:
- Memory:



- Free List:



Issues

The biggest issue with this approach is fragmentation. Suppose we have 48 bytes and we make the following calls:

Allocate 12

Allocate 20

Allocate 4

Free 20

Allocate 8

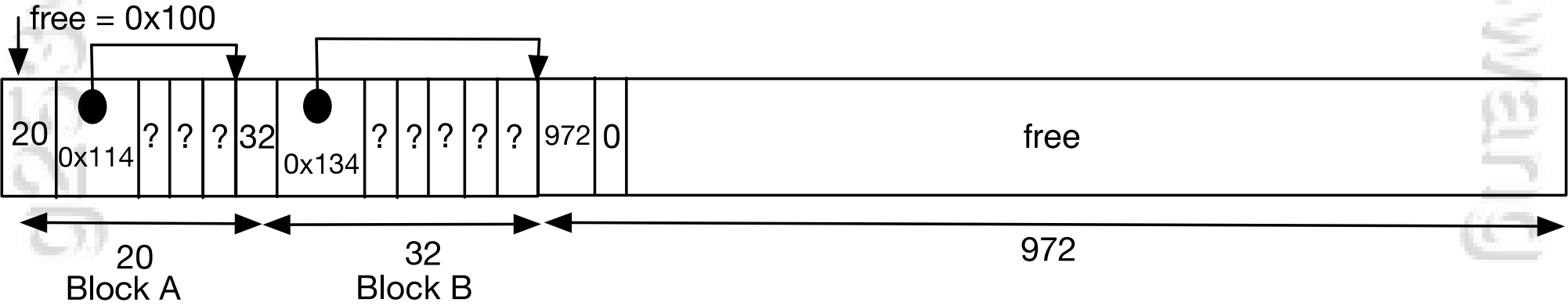
Cannot allocate 16, despite having 24 bytes free!

Free List?

- We've been showing the free list as a separate data structure...
- but, we're the ones allocating data structures...
- so, how do we allocate space for the free list itself?

Free List

- The free list goes *in the space itself*. Remember, it's free, so the user doesn't care what we put there!



- Let's do a full example (the fragmentation example) on the board

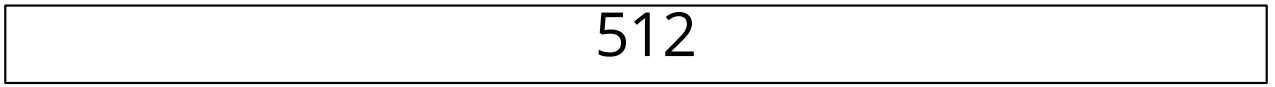
Dealing With Fragmentation

Heuristics:

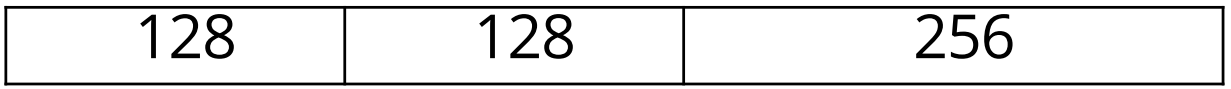
- First fit: Put memory in the first available spot
- Best fit: Put the block in an exact match (or as close to it so there is less waste)
- Worst fit: Exact match if possible, otherwise put the block in the largest available space
- Problem: Best- and worst-fit involve looking over the whole free list, so are slow!
- Other ideas include dmalloc and the *binary buddy system* (or buddy memory allocation). We discuss the latter.

Binary Buddy System

Idea: Start off with (e.g.) 512 bytes of heap memory:



Suppose we try to allocate 19 bytes. We would need need an extra one for bookkeeping (so 20 total). This fits in a block of size $2^5 = 32$. We split our memory until we find such a block and reserve the entire block.



Continuing

64	64	128	256

and finally:

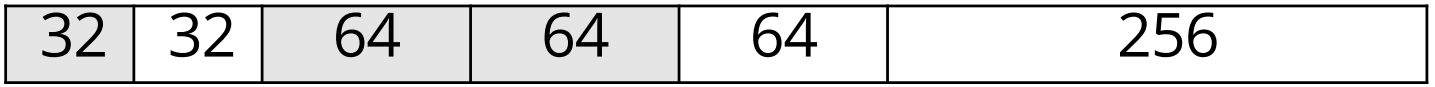
32	32	64	128	256

Now, we request 63 bytes (so we need 64 bytes of space) we have an exact place to put this so we allocate there:

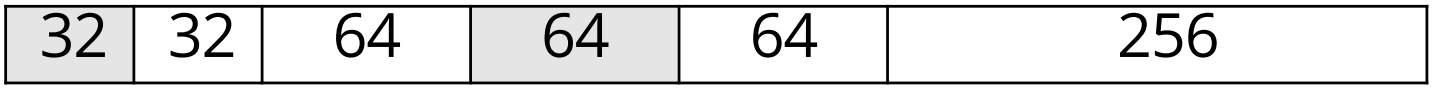
32	32	64	128	256

Continuing

Suppose now we need 40 bytes (so actually 41). We need a 64-byte block we don't have. We split the 128 block.



Now let's say we free the allocated 63-byte block:



Continuing

Further, let's say we also free the allocated 19-byte block:

32	32	64	64	64	256

Notice that 32 and its neighboring buddy are both free, so we collapse!

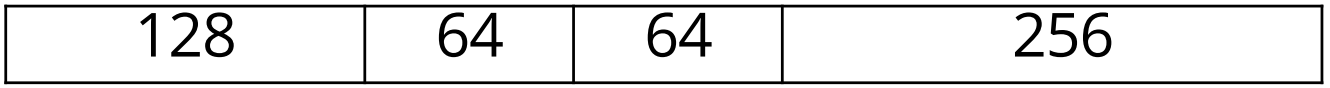
64	64	64	64	256

We can collapse again!

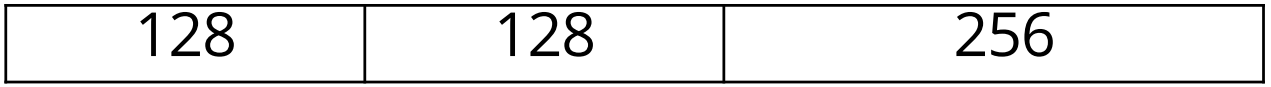
128	64	64	256

Continuing

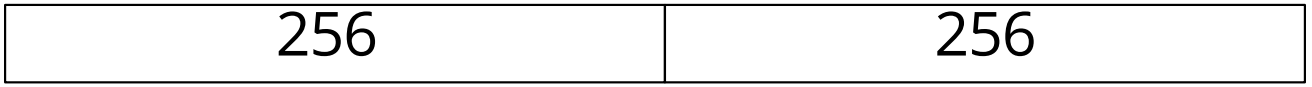
Lastly, we free the allocated 40-byte block:



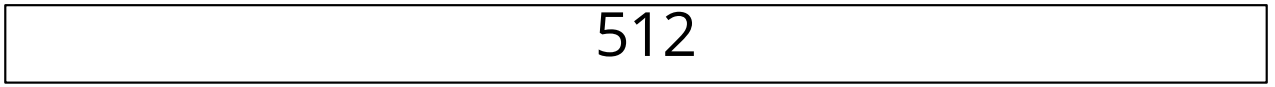
and we collapse:



and again:



and again:



Why?

- Binary buddy still creates fragmentation, just of a different sort.
- If most allocations are of nice powers of two, it defragments well.
- More importantly: Both the *size* and the *location* of any block can be encoded using a list of “left” and “right actions!

Binary Buddy Code

- Start with a 1. The code for the whole of the heap is just 1.
- For each block division, append a 0 if the left block is selected, or a 1 if the right block is selected.
- Since the first bit is always 1, we can tell the length of the code by looking for the first 1.
- Since each bit represents a power of two, very short codes represent a lot of information.
- One word is plenty for a code.

Demonstration

- Let’s demonstrate binary buddy using the same sequence of allocations and deletions as the fragmentation example

Allocate 12

Allocate 20

Allocate 4

Free 20

Allocate 8

Automatic Memory Management

A category of languages informally described as "languages that are not terrible" will take care of the deallocation for you, so that you don't need to delete when you are done using memory. Here are some examples from Java:

```
int f(){
   MyClass ob = new MyClass(); // Java
   ...
} //ob no longer accessible
// garbage collector reclaims.
```

Automatic Memory Management

Second example:

```
int f(){
   MyClass ob2 = null;
   if(x == y){
      MyClass ob1 = new MyClass();
      ob2 = ob1;
   }//ob1 goes out of scope;
   //BUT ob2 still  holds the object
   //so not reclaimed
   ...
} //ob2 no longer accessible;
//no one else holds the obj so reclaimed
```

Automatic Memory Management

- In order for automatic memory management to happen, the compiler and the allocator need to coordinate in some way
- At a minimum, the compiler needs to tell the allocator when something goes out of scope
  - But, scope isn't actually good enough…

Technique 1: Reference Counting

- For each heap block, keep track of the number of pointers that point to it.
- Must watch every pointer and update reference counts each time a pointer is reassigned (decrement the old one and increment the new one).
  - That is, the compiler needs to call some procedure provided by the allocator every time a pointer goes into or out of scope
- If a block's reference count reaches 0, then reclaim it.

Aside: It was never really scope

- Consider:

```
struct List {
    List *next;
    int val;
};

l = new List;
l->next = new List;
// l->next is not a variable in
// scope, it's a field of an object,
// and there is a reference to that
// object in scope
```

Technique 1: Reference Counting

- What issues are there to this?
- If a block points to another block and vice versa (and nothing points to the cluster) then the cluster is unreachable and should be cleaned…
- But both will retain a reference to each other!
- Let’s see this in practice with C++ shared_ptrs (which is reference counting)

Solutions?

- If reference counting is so bad, why do people use it?
  - ~~Because they're idiots~~
  - It's very straightforward to implement, so it's an easy way to get *some* automatic memory management easily
  - But, it's expensive (always twiddling counts) and flawed, so should only be used if you have a very good reason

Fixes?

- Is it possible to fix this issue with reference counting?
  - Short answer: No
  - Long answer: Nooooooooooooooooooo
  - Longer answer: What if we delayed counting references, so that we would never get to these problematic clusters in the first place...

Garbage Collection

- The remaining techniques we will describe are classed as *garbage collection*
- GC requires* that the language be designed with GC in mind, so we couldn't bolt it onto WLP4

* there are some exceptions called *conservative garbage collectors*, but that's way beyond the scope of this course

Technique 2: Mark and Sweep

- Scan the entire stack [and global variables] and search for pointers.
- Mark the heap blocks that have pointers to them. If these contain pointers, continue following etc.
- Then scan the heap: reclaim any blocks that aren't marked.
- Boils down to a graph traversal problem.

Mark and Sweep

- The compiler needs to tell the allocator
  - Where pointers are in the stack
  - Where pointers are in the heap
- This means that the compiler and allocator need to agree on both stack layout and object layout
- This is a much greater degree of co-design than is needed for reference counting, so mark and sweep is rarely bolted onto a language post hoc

Mark and Sweep

- Let's demonstrate this on the board with a binary tree type

```
class BinTree {
    BinTree left, right; // references (Java)
    int val;
}
```

Technique 3: Copying Collector

- Idea: Split the heap into two halves, say H_1 and H_2
- Allocate memory in H_1. When, perform a *mark-and-copy*: trace like in mark-and-sweep, but mark by copying objects from H_1 into H_2
- After the copy, H_2 has all living objects stored contiguously.
- Once finished copying, begin allocation to H_2 (that is, reverse the roles of H_1 and H_2)

Technique 3: Copying the Collector

- "Copy? Isn't that slow?"
- Three major benefits:
  - Allocation is always extremely fast, because we simply fill a heap half from beginning to end
  - The "sweep" phase is literally free
  - We have no fragmentation in our memory; new and delete are very quick
- If most objects aren't long-lived, then we don't actually copy much

More Co-Design!

- Since copying an object from one heap to an object is *moving* objects, the compiler has to be prepared for the allocator to actually *modify active memory!*
- The location of any object changes over time!

Demo

- Let’s demonstrate on the board again with our tree type

Generalizing

- In practice, most objects are short-lived (good for copy!), but those objects that aren't short lived are nearly immortal
- Thus, most practical garbage collectors are *generational* collectors: they use copying from H_1 to H_2, but then H_2 is mark-and-sweep (or a related technique, mark-and-compact).
- With generational, we benefit from the free sweep for young objects, and the lack of copying for old objects