Custom Memory Allocators

ITP 435
Week 5, Lecture 1

Concerns about the default heap allocator

Design Decision	Concern
A general-purpose allocator to handle a large variety of allocations	It can't be optimized for specific usage cases
Must be thread-safe	Pay for thread safety even when not needed
No way to track memory usage of specific systems	We may want this!
Checking for memory leaks requires external tools	We may want to internally test for this.

These are all reasons why you may implement a custom memory allocator!

Some Background: Dynamic Memory in C

- In C, there is no `new`/`delete`

- Dynamic allocation is done via ***malloc***:
- `void* malloc(size_t size)`
  - Takes in size of allocation
  - Returns void* pointer to memory address

- Example:

```
// Allocate a single integer
int* p1 = malloc(sizeof(int));
```

Some Background: Dynamic Memory in C

- In C, there is no `new`/`delete`
- Dynamic allocation is done via ***malloc***:
- `void* malloc(size_t size)`
  - Takes in size of allocation
  - Returns void* pointer to memory address

`void*` means it's a memory address, but we make no guarantees about what type we will store at this address

- Example:

```
// Allocate a single integer
int* p1 = malloc(sizeof(int));
```

More Malloc Examples

```
// Allocate an array of 10 integers
int* array = malloc(sizeof(int) * 10);

// Allocate a struct
struct Test { int a; int b; };
Test* test = malloc(sizeof(Test));
```

- Notably, ***these do not initialize the memory*** to anything!

Initializing After Allocation

Outside of manually setting members of the struct, the only way to initialize is to use ***memset***:

```
// Allocate a struct
struct Test { int a; int b; };
Test* test = malloc(sizeof(Test));

// Use memset to set the memory of the struct to 0
// aka "zero out" the memory
// Parameters are (address, value, number of bytes)
memset(test, 0, sizeof(Test));
```

What does new do?

- In C++, the default `new` has two separate responsibilities:
    1. Dynamically allocate memory
    2. Call the constructor for the object(s), if they exist

- We want to customize the ***first*** part, but not the second (incidentally, there's no way to customize the second part)

Mixing malloc and C++

- We can allocate memory for a class using malloc:

```
class Test {
public:
   Test() { std::cout << "Constructor!" << std::endl; }
};
// Allocate memory for a Test instance
Test* myTest = static_cast<Test*>(malloc(sizeof(Test)));
```

- To construct it, we have to use a special syntax of new called ***placement new***:

```
// Construct the object using "placement" new
new(myTest) Test();
```

- The default placement new will construct the object in the specified memory address

Placement new

- More generally, a ***placement new*** just means it has additional parameters

- Placement new where you pass in a pointer constructs the object in the memory you provide:

```
// This will construct my class inside buffer
char buffer[1024];
MyClass* p = new (buffer) MyClass();
```

- “nothrow” placement new tells the program you don’t want it throwing an exception if it fails:

```
// This will return nullptr if it fails
char* c = new (std::nothrow) char[1024];
```

Freeing Memory in C

- To free memory allocated with malloc, use ***free***:
- `void free(void* ptr)`
  - Takes in the pointer to be freed

Mixing malloc/free with C++

- Have to manually call the destructor (which is not pretty)...

```
class Test {
public:
   Test() { std::cout << "Constructor!" << std::endl; }
   ~Test() { std::cout << "Destructor!" << std::endl; }
};

// Allocate memory for a Test instance
Test* myTest = static_cast<Test*>(malloc(sizeof(Test)));
// Construct the object using "placement" new
new(myTest) Test();
// Destruct the object
myTest->~Test();
// Free the memory
free(myTest);
```

A Better Solution: Override new and delete

- You can override new/delete for a particular class type:

```
class Test {
public:
   Test() { std::cout << "Constructor!" << std::endl; }
   ~Test() { std::cout << "Destructor!" << std::endl; }
   // Override operator "new" to allocate with malloc
   static void* operator new(size_t size) {
      std::cout << "Custom new!" << std::endl;
      return malloc(size);
   }
   // Override operator "delete" to free with free
   static void operator delete(void* ptr) {
      std::cout << "Custom delete!" << std::endl;
      free(ptr);
   }
   // NOTE: Should also overload new[] and delete[]
};
```

Custom new/delete in Action

```cpp
Test* myTest = new Test();
delete myTest;
```

C:\WINDOWS\system32\cmd.exe

```
Custom new!
Constructor!
Destructor!
Custom delete!
Press any key to continue . . .
```

Remember that custom new/delete only overrides the allocation behavior, not the constructor/ destructor behavior!

Fitting in our custom allocators

```
// Override operator "new" to allocate with malloc
static void* operator new(size_t size) {
    std::cout << "Custom new!" << std::endl;
    return malloc(size);
}
// Override operator "delete" to free with free
static void operator delete(void* ptr) {
    std::cout << "Custom delete!" << std::endl;
    free(ptr);
}
```

- We need to replace “malloc” and “free” with our own custom allocator functions (as malloc/free still uses the default heap)

Note on overriding new/delete

- It is possible to globally override, but not recommended!

- Also technically, when we override new we should throw `std::bad_alloc` if it fails to allocate, and a couple of other things

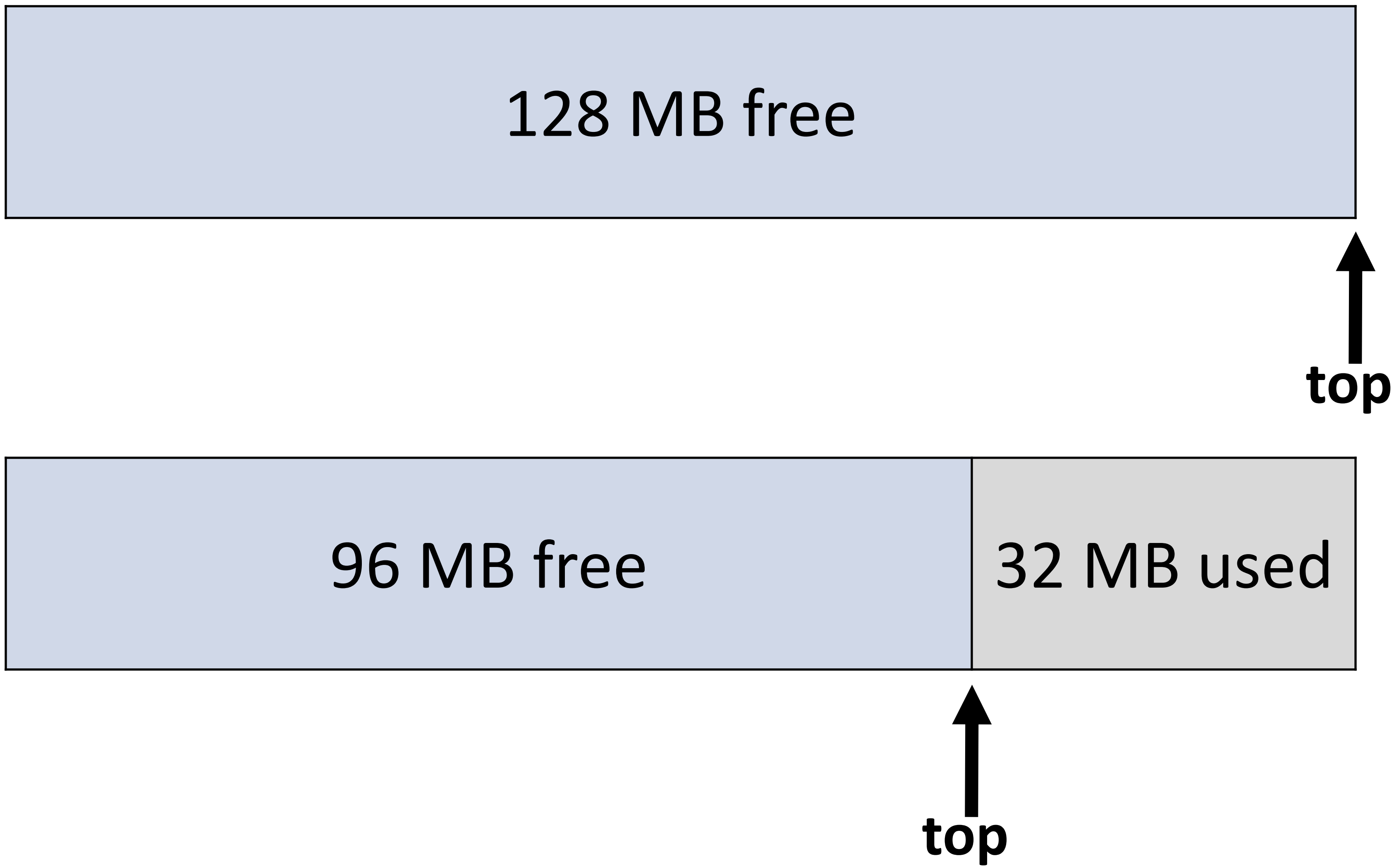
- See the relevant Effective C++ sections for further information

Types of Custom Memory Allocators

- We'll cover three varieties:
  - ***Stack allocator*** (aka arena allocator) – Good for lots of temporary allocations that you can throw away at once
  - ***Pool allocator*** – Good for many same-sized (typically small) allocations
  - ***Boundary tag allocator*** (aka slab allocator) – A more general-purpose allocator

Stack Allocator

Stack Allocator



Basic Stack Allocator - Declaration

```
class BasicStackAlloc
{
public:
   // Constructs the stack allocator to requested size
   BasicStackAlloc(size_t size);
   // Destructor
   ~BasicStackAlloc();
   // Gets how many bytes are left in the stack
   size_t GetBytesRemaining() const;
   // Allocates the specified number of bytes, if available,
   // and returns a pointer to this allocation
   // Returns nullptr if not enough bytes are left
   void* Allocate(size_t size);
   // Disallow copy construction, assignment, and moves
   BasicStackAlloc(const BasicStackAlloc&) = delete;
   BasicStackAlloc& operator=(const BasicStackAlloc&) = delete;
   BasicStackAlloc(BasicStackAlloc&&) = delete;
   BasicStackAlloc& operator=(BasicStackAlloc&&) = delete;
private:
   // Pointer to the beginning of the usable memory area of the stack
   char* mMemoryBuffer;
   // Pointer to top of stack
   char* mTop;
};
```

Basic Stack Allocator – Constructor/Destructor

```
BasicStackAlloc::BasicStackAlloc(size_t size)
	:mMemoryBuffer(nullptr)
	,mTop(nullptr)
{
	// Allocate specified bytes
	mMemoryBuffer = new char[size];
	// The "top" should be at the end of the buffer
	// (pointer arithmetic)
	mTop = mMemoryBuffer + size;
}

BasicStackAlloc::~BasicStackAlloc()
{
	delete[] mMemoryBuffer;
	mMemoryBuffer = nullptr;
	mTop = nullptr;
}
```

Basic Stack Allocator – GetBytes…/Allocate

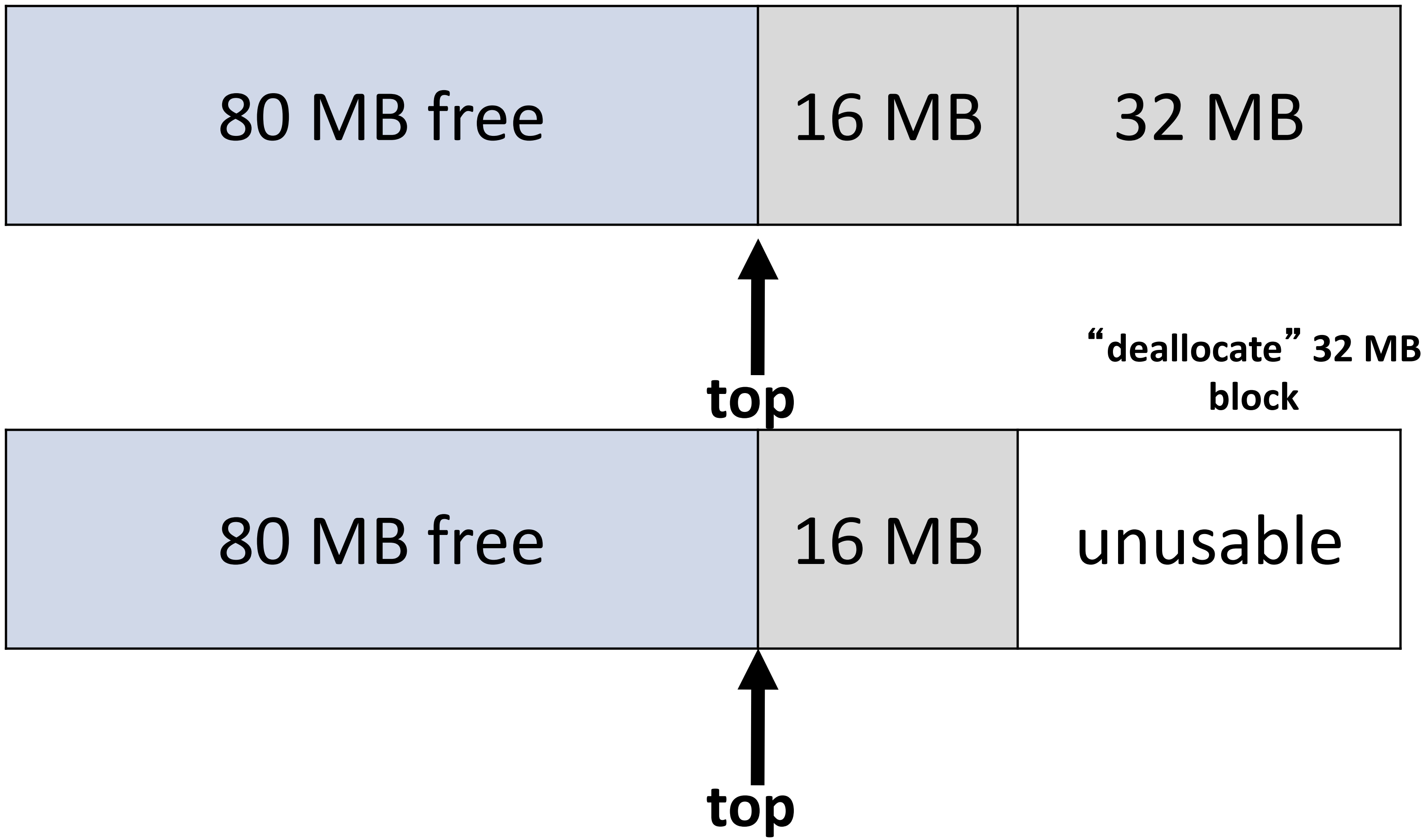
```
size_t BasicStackAlloc::GetBytesRemaining() const {
   return (mTop - mMemoryBuffer);
}

void* BasicStackAlloc::Allocate(size_t size) {
   // If we have enough space, move the top by the size
   // and return this address
   if (GetBytesRemaining() >= size) {
      mTop -= size;
      return mTop;
   } else {
      // Not enough space, so return null
      return nullptr;
   }
}
```

"Freeing" Memory with Stack Allocator

- There is no way to free individual allocations – you can only free the entire stack!

- This is why a stack allocator is good for lots of temporary allocations that you know you can throw away at a fixed point

Stack Fragmentation



Pool Allocator

Pool Allocator

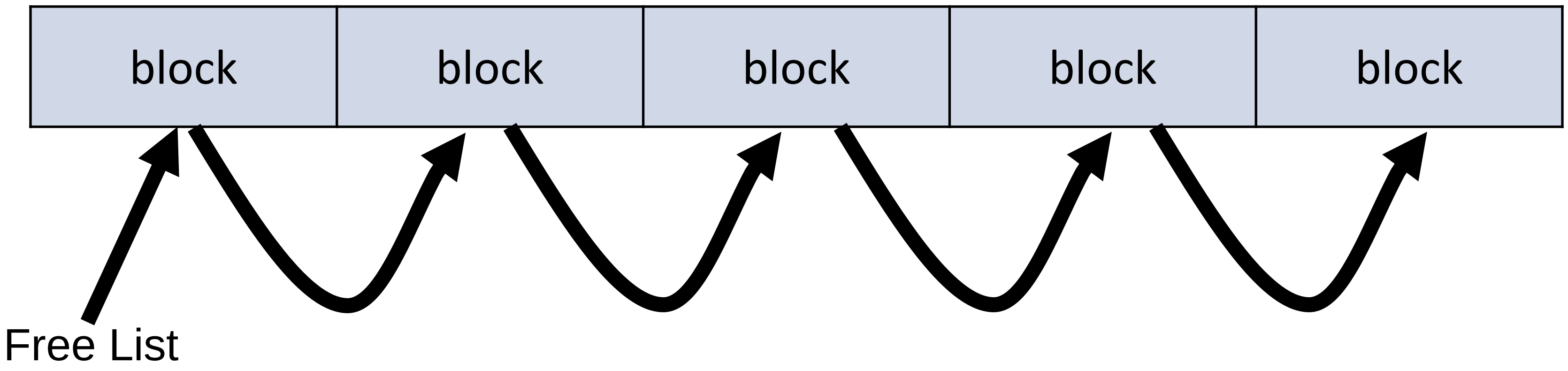
- Have several identical-sized blocks
- Allocation requests will return a free block from the pool, if one's available

64B	64B	64B	64B	64B	64B	64B	64B	64B	64B

- Good for when you know you will have several allocations of approximately the same size

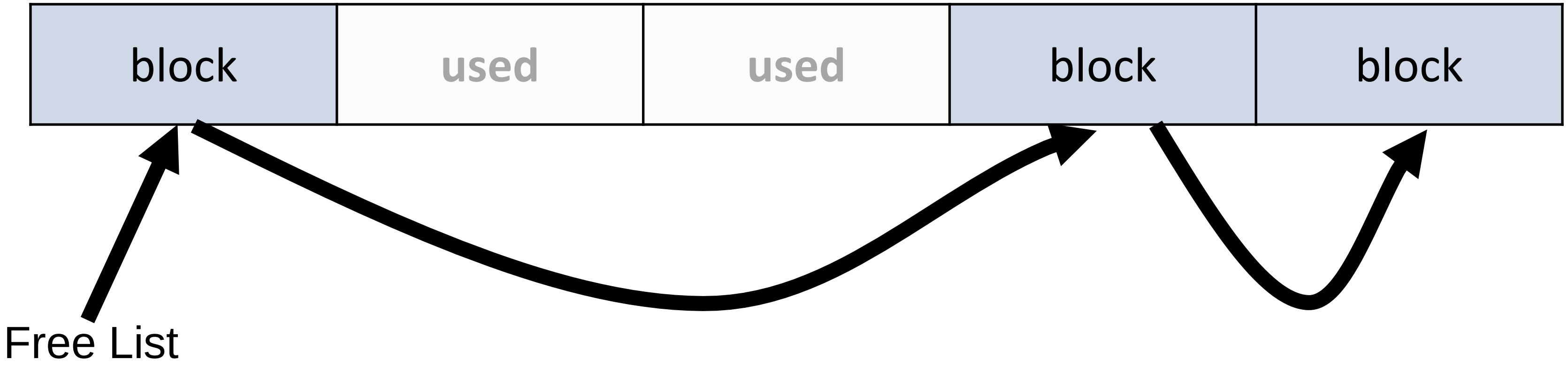
The Free List

- A pool allocator has a ***free list***
  - Linked list of blocks that are available for allocation
- ***O(1)*** constant time to find an available block, if one is available
- Initially, every block is on the free list:



The Free List

- Over time, as we allocate/deallocate blocks in any arbitrary order, the free list will still give ***O(1)*** access to a free block



Anatomy of a Block

- Observation: either a block is available (in the free list), or it's not (its memory is in use)
- This means we have mutually exclusive data – either memory for the block or memory for the next pointer (to maintain the list)
- Use a ***union*** for the block!

```
template <size_t blockSize>
union PoolBlock
{
   // Usable memory of the block
   char mMemory[blockSize];
   // Pointer to next block that's free
   PoolBlock* mNext;
};
```

Pool Allocator - Declaration

```
template <size_t blockSize>
class PoolAlloc {
public:
    // Construct Pool Allocator with requested number of blocks
    PoolAlloc(size_t numBlocks);
    // Destruct the Pool Allocator
    ~PoolAlloc();
    // Allocates a block, if possible
    void* Allocate(size_t size);
    // Puts a block back into the free list
    void Free(void* ptr);
    // Disallow copy construction, assignment, and moves
    // ...
private:
    union PoolBlock { /* From previous slide... */ };
    PoolBlock* mHead; // Pointer to head of free list
    PoolBlock* mAllBlocks; // Pointer to array of all blocks
    size_t mFreeBlocks; // Tracks the number of free blocks
    size_t mTotalBlocks; // Tracks the total number of blocks
};
```

Pool Allocator – Constructor

```
template <size_t blockSize>
PoolAlloc<blockSize>::PoolAlloc(size_t numBlocks)
   :mHead(nullptr)
   ,mAllBlocks(nullptr)
   ,mFreeBlocks(numBlocks)
   ,mTotalBlocks(numBlocks)
{
   // First, allocate "all blocks" as an array of numBlocks blocks
   mAllBlocks = new PoolBlock[numBlocks];

   // Set the "next pointer" for all blocks
   // (Initially, this is just to the block at the next index)
   for (unsigned i = 0; i < mTotalBlocks - 1; i++) {
      mAllBlocks[i].mNext = &mAllBlocks[i + 1];
   }

   // The next for the last block is null (end of list)
   mAllBlocks[mTotalBlocks - 1].mNext = nullptr;
   // The head of the free list is just the first block in "all blocks"
   mHead = &mAllBlocks[0];
}
```

Pool Allocator – Allocate

```
template <size_t blockSize>
void* PoolAlloc<blockSize>::Allocate(size_t size)
{
   // Only return a block if the alloc request fits,
   // and there are blocks available
   if (mFreeBlocks > 0 && size <= blockSize) {
      // Save the address of the head
      PoolBlock* oldHead = mHead;
      // Advance the head
      mHead = mHead->mNext;
      // One less block is free
      mFreeBlocks--;
      // Return the old head block
      return oldHead;
   } else {
      return nullptr;
   }
}
```

Pool Allocator – Free

```
template <size_t blockSize>
void PoolAlloc<blockSize>::Free(void* ptr)
{
   // NOTE: In a professional-grade allocator, we should verify that
   // ptr is actually a block in this pool!

   // Cast to the block pointer type
   PoolBlock* reclaimed = static_cast<PoolBlock*>(ptr);
   // Make the reclaimed block the head of the free list
   reclaimed->mNext = mHead;
   mHead = reclaimed;
   // One more block is free
   mFreeBlocks++;
}
```

Pool Allocator – Destructor

```
template <size_t blockSize>
PoolAlloc<blockSize>::~PoolAlloc()
{
   // Delete all the blocks
   delete[] mAllBlocks;
   // Clean up member variables, just in case
   mHead = nullptr;
   mAllBlocks = nullptr;
   mFreeBlocks = 0;
   mTotalBlocks = 0;
}
```

Using the Pool Allocator with a class

```
class MyClass {
public:
   // Any members of MyClass
   // ...

   // Override operator "new" to allocate from pool
   static void* operator new(size_t size) {
      return sTestPool.Allocate(size);
   }
   // Override operator "delete" to free back to pool
   static void operator delete(void* ptr) {
      sTestPool.Free(ptr);
   }
   // Note: Should also overload new[] and delete[], which are similar
private:
   // Static PoolAllocator that's shared by all instances of MyClass!
   static PoolAlloc<sizeof(MyClass)> sTestPool;
};
// Allows a max of 1000 MyClass objects
PoolAlloc<sizeof(MyClass)> MyClass::sTestPool(1000);
```

Pool Allocator

- Problem: We may have wildly different allocation sizes.
- Solution: Have one pool per size type, and pick best fit at runtime (this makes the code a bit more complex, of course)

512B pool

256B pool

128B pool

64B pool

In-class activity

Boundary Tag Allocator

Boundary Tag Allocator

- A more general purpose-allocator
- Can allocate/free in an arbitrary order with a variety of sizes
- Attempts to minimize waste

Sample block layout

Size	4 bytes
Next Free Ptr	8 bytes
Data	>= 32 bytes
Prev Free Ptr	8 bytes
Size	4 bytes

Problem: Since data is a variable size, how do we mark the "tags"?

Initial Creation

- Instantiate it to a large array of chars (like in a stack)
- Write the "tags" at the start and end of the block

- Ex: Instantiate to 2048 bytes
  - 4 bytes will store size at start
  - 8 bytes will store pointer to next free block
  - 8 bytes will store pointer to next prev free block
  - 4 bytes will store size at end
  - So 2048 – 4 * 2 – 8 * 2= 2024 bytes block size
  - (Still also would need pointers to head/tail for free list)

2048 byte allocator (initial state)

Head/
Tail

Size	2024
Next Free Ptr	nullptr
Data	2024 bytes... (available)
Prev Free Ptr	nullptr
Size	2024

Memory request

- Suppose we get a request for 256 bytes

- The current block has 2024 bytes available…do we really want to waste ~1700 bytes?

Request for 256 bytes

- Split the big block into two, and one of them is free still

Head/
Tail

Size	1744
Next Free Ptr	nullptr
Data	1744 bytes... (available)
Prev Free Ptr	nullptr
Size	1744
Size	256
Next Free Ptr	0x1
Data	256 bytes... (in use)
Prev Free ptr	0x1
Size	256

Request for 256 bytes

- Split the big block into two, and one of them is free still

Head/
Tail

Size	1744
Next Free Ptr	nullptr
Data	1744 bytes... (available)
Prev Free Ptr	nullptr
Size	1744
Size	256
Next Free Ptr	0x1
Data	256 bytes... (in use)
Prev Free ptr	0x1
Size	256

0x1 is a magic number to denote the block is in use

Request for 128 bytes

- The top block has to be split again

Head / Tail

Size	1592
Next Free Ptr	nullptr
Data	1592 bytes...(available)
Prev Free Ptr	nullptr
Size	1592
Size	128
Next Free Ptr	0x1
Data	128 bytes...(in use)
Prev Free Ptr	0x1
Size	128
Size	256
Next Free Ptr	0x1
Data	256 bytes...(in use)
Prev Free Ptr	0x1
Size	256

"Freeing" a block

- "Is the block below me free?"
  - If so, combine with the below block
- "Is the block above me free?"
  - If so, combine with the above block

- If neither are free, we need to linearly search either up or down for the closest free block and add ourselves to the free list properly

- Either way, might need to fix up free list pointers

Freeing

- Suppose we want to free the 256 byte block

Head / Tail

Size	1592
Next Free Ptr	nullptr
Data	1592 bytes...(available)
Prev Free Ptr	nullptr
Size	1592
Size	128
Next Free Ptr	0x1
Data	128 bytes...(in use)
Prev Free Ptr	0x1
Size	128
Size	256
Next Free Ptr	0x1
Data	256 bytes...(in use)
Prev Free Ptr	0x1
Size	256

Freeing 256 block

- There's no adjacent block below, and the one above is not free

Head / Tail

Size	1592
Next Free Ptr	nullptr
Data	1592 bytes...(available)
Prev Free Ptr	nullptr
Size	1592
Size	128
Next Free Ptr	0x1
Data	128 bytes...(in use)
Prev Free Ptr	0x1
Size	128
Size	256
Next Free Ptr	0x1
Data	256 bytes...(in use)
Prev Free Ptr	0x1
Size	256

Freeing 256 block

- We search upwards and find that the 1592 size block is free

Head / Tail

Size	1592
Next Free Ptr	nullptr
Data	1592 bytes...(available)
Prev Free Ptr	nullptr
Size	1592
Size	128
Next Free Ptr	0x1
Data	128 bytes...(in use)
Prev Free Ptr	0x1
Size	128
Size	256
Next Free Ptr	0x1
Data	256 bytes...(in use)
Prev Free Ptr	0x1
Size	256

Freeing 256 block

- We can then add the 256 byte block into the free list!

Head

Size	1592
Next Free Ptr	&(256 block)
Data	1592 bytes...(available)
Prev Free Ptr	nullptr
Size	1592
Size	128
Next Free Ptr	0x1
Data	128 bytes...(in use)
Prev Free Ptr	0x1
Size	128
Size	256
Next Free Ptr	nullptr
Data	256 bytes...(available)
Prev Free Ptr	&(1592 block)
Size	256

Tail

Freeing 128 block

- If we freed the 128 block, we could coalesce back into one big block!!!!

Head

Tail

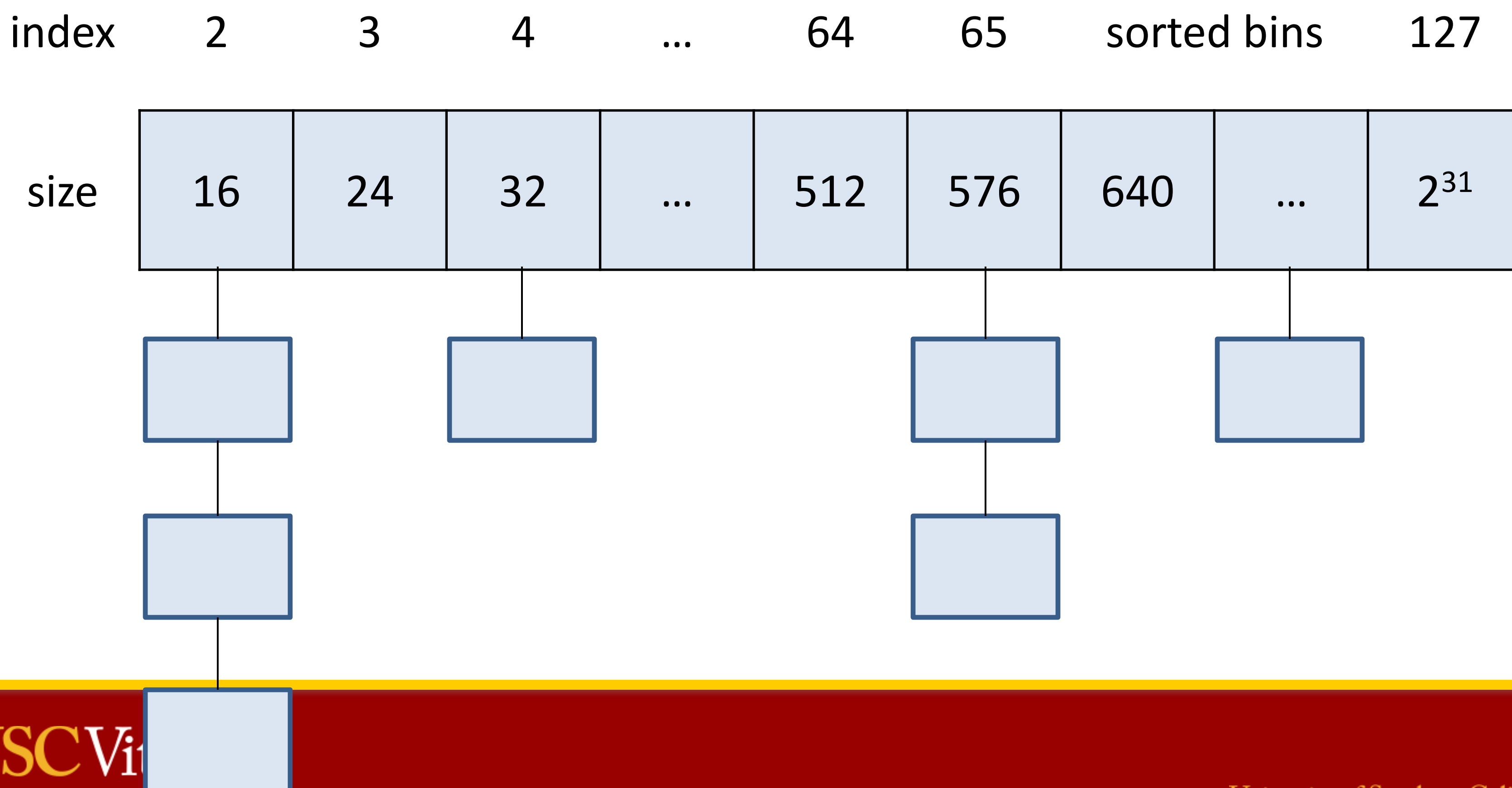
Size	1592
Next Free Ptr	&(256 block)
Data	1592 bytes...(available)
Prev Free Ptr	nullptr
Size	1592
Size	128
Next Free Ptr	0x1
Data	128 bytes...(in use)
Prev Free Ptr	0x1
Size	128
Size	256
Next Free Ptr	nullptr
Data	256 bytes...(available)
Prev Free Ptr	&(1592 block)
Size	256

dlmalloc – Example of boundary tag allocator

Allocated chunk	size,status=inuse
	… user data …
	size
Free chunk	size,status=free
	pointer to next free block
	pointer to previous free block
	… unused space …
	size
Allocated chunk	size,status=inuse
	user data
	size
Other chunk…	…
Allocated chunk	size,status=free
	…
	size

dlmalloc - Binning

- Free blocks are organized into buckets.
- So when you need a block of a particular size, you can just grab one from the appropriate bucket.



STL and Allocators

STL containers and allocators

- Every STL container has an optional allocator template parameter:

```
template <class T, class Allocator = allocator<T>> class list;
```

- If you want to overload STL container allocations, you could create an Allocator class and pass it to the template…

- (But this feature is somewhat unpopular…)

STL Allocator

```
template <class T> class allocator
{
public:
    // Required data definitions removed...

    allocator() noexcept;
    allocator(const allocator&) noexcept;
    template <class U> allocator(const allocator<U>&) noexcept;
    ~allocator() noexcept;

    pointer address(reference x) const;
    const_pointer address(const_reference x) const;

    pointer allocate(size_type,
                     allocator<void>::const_pointer hint = 0);
    void deallocate(pointer p, size_type n);
    size_type max_size() const noexcept;

    void construct(pointer p, const T& val);
    void destroy(pointer p);
};
```

Memory Resources (C++17)

- These are STL-provided implementations of commonly used allocators
- Sample: https://github.com/chalonverse/CPPBeyondSamples/blob/master/Ex04/Main.cpp
- The "stack allocator" is `montonic_buffer_resource`
- There are two types of "pool" allocators, one that's thread-safe and one that isn't:
  - `synchronized_pool_resource`
  - `unsynchronized_pool_resource`

Stack Allocator Example

```
// A monotonic_buffer_resource is like the "stack allocator"
std::pmr::monotonic_buffer_resource stackAlloc;

// Allocate 50 bytes
void* ptr2 = stackAlloc.allocate(50);

// Deallocate won't do anything in this case!
stackAlloc.deallocate(ptr2, 50);

// To release all the memory use the release function
// (Also happens when the destructor is called)
stackAlloc.release();
```

Pool Allocator Example

```
// Use pool_options to configure the pool
std::pmr::pool_options options;
// Maximum of 1024 blocks
options.max_blocks_per_chunk = 1024;
// Largest block size required in bytes
options.largest_required_pool_block = 16;

// This is a non-thread safe pool allocator
std::pmr::unsynchronized_pool_resource poolAlloc(options);
// Allocate 12 bytes
void* ptr = poolAlloc.allocate(12);
// Deallocate the memory
poolAlloc.deallocate(ptr, 12);
```