CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

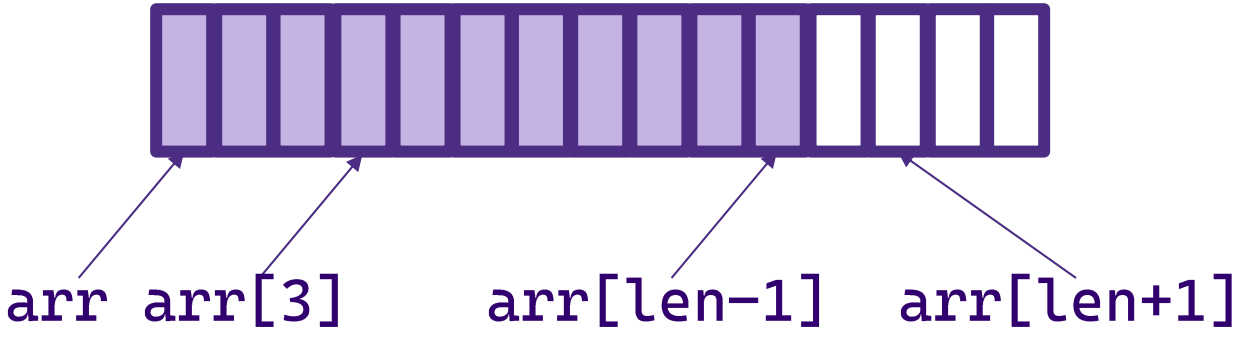
Subject Matter

- Memory manipulation
- Memory homework (6)

Your Goals

- Keep working on
  - Debugging
  - HW5 (due FRIDAY)
  - HW6

Arrays

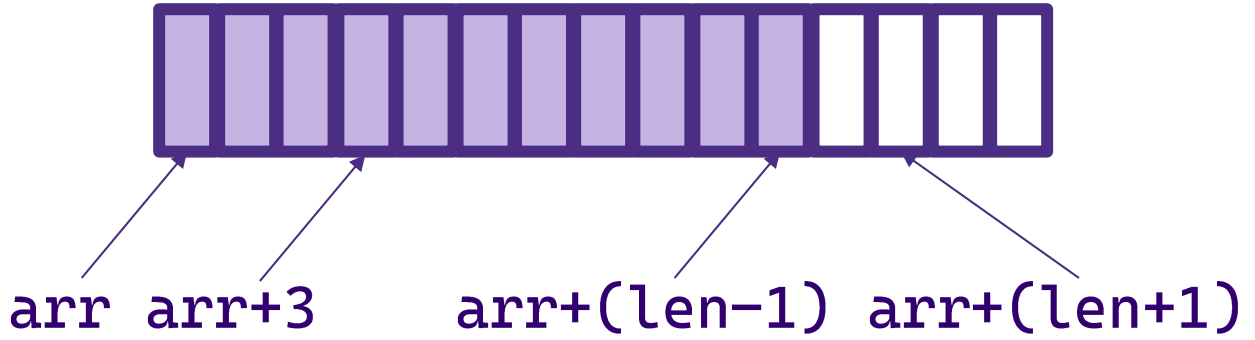


- Arrays are contiguous blocks of memory
  - `Datatype arr[len]`
  - Has type: `Datatype*`
- Why?
  - Array types actually store the address at the beginning of the block of memory

- What happens with arr[len+1]?
- **Best case scenario: you crash.**

The memory at arr[len+1] is not designated for the array. Could be ok, could be used for something else?

Pointer arithmetic

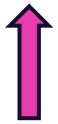


- If `p` has type `T*` or `T[]`
  - `*p` has type `T`
- If `p` points to one item of type `T`, `p+1` points to a place in memory for the next item of type `T`
  - So, `p[0]` is one item of type `T`, `p+i = p[i]`
- `T[]` always has type `T*`, even if it is declared as `T[]`
  - Implicit array promotion
    Result: Arrays are always passed by reference (the address of the data), not by value.

Hexadecimal in C

- There is no unique type for hexadecimal in C. We use '`unsigned int`' or '`unsigned char`'.
  Remember, `sizeof(int) = 4` [bytes]
  and `sizeof(char) = 1` [byte] => **2 hex digits**
- An unsigned char can hold values up to 255 or oxFF (maximum two digit hex value)

```
unsigned char ahexvalue = 0xFE;
uintptr_t mymem = (uintptr_t) malloc (16);
for (int i = 0; i < 16; i++) {
    *((unsigned char*)(mymem+i)) = 0xFE;
}
```



Pointer arithmetic in unsigned integer space

Memory (zoomed in)

- Remember, a string is an array of characters, allocated with a certain amount of space
- Note: With the code to the right we enter dangerous territory because we might over-write the buffer.
- `gets` is deprecated: use `fgets`

```
void echo() {
    char buf[8];
    gets(buf);
    printf("%s", buf);
}
```

Memory – buffer use

- Remember, a string is an array of characters, allocated with a certain amount of space
- With:

```
Enter a string: hello
hello
```

```
int main (int argc, char **argv) {
    printf("Enter a string: ");
    echo();
    return 0;
}
void echo() {
    char buf[8];
    gets(buf);
    printf("%s", buf);
}
```

Memory – buffer use bad

- Remember, a string is an array of characters, allocated with a certain amount of space
- With:

```
Enter a string:
helloagain
```

```
int main (int argc, char **argv) {
    printf("Enter a string: ");
    echo();
    return 0;
}
void echo() {
    char buf[8];
    gets(buf);
    printf("%s", buf);
}
```

Memory – buffers can still break!

- Even if we aren't using a *very bad* function like `gets`, we can break things:

```
[mh75@calgary cse374]$ gcc -o getsbreak
getsbreak.c
getsbreak.c: In function 'echo':
getsbreak.c:14:5: warning: 'fgets' writing
16 bytes into a region of size 8 overflows
the destination [-Wstringop-overflow=]
   14 |     fgets(buf, 16, stdin);
      |     ^~~~~~~~~~~~~~~~~~~~~

[mh75@calgary cse374]$ ./getsbreak
Enter a string: helloagain
helloagain
Segmentation fault (core dumped)
```

```
int main (int argc, char **argv) {
    printf("Enter a string: ");
    echo();
    return 0;
}
void echo() {
    char buf[8];
    fgets(buf, 16, stdin);
    printf("%s", buf);
}
```

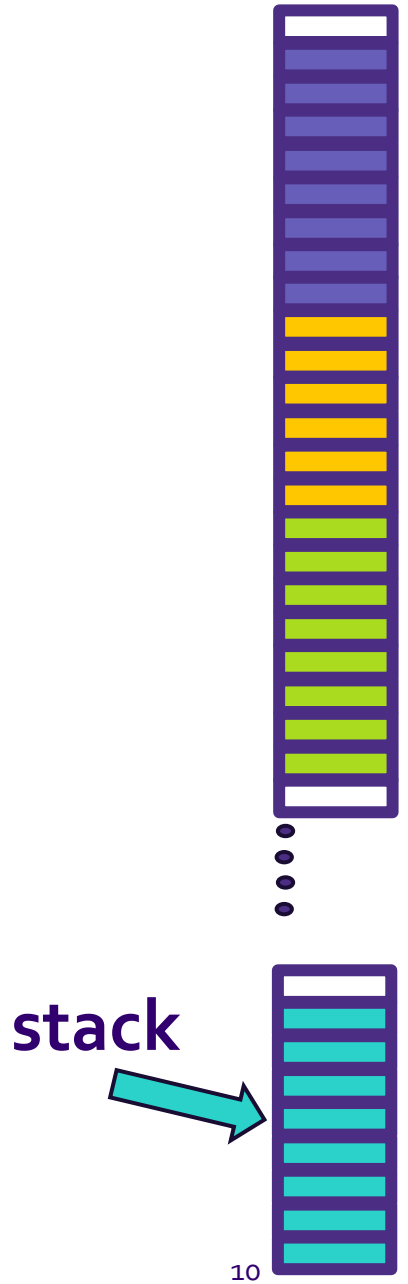
How bad is buffer overflow? Remember: The Stack

rpb (8 bytes)
rip (8 bytes)
Arguments (? Bytes)

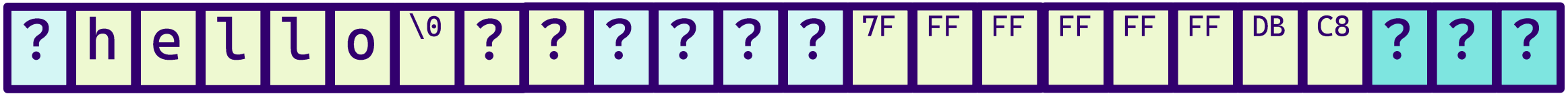
Locals (? Bytes)

Callee frame

Caller frame



Stack smashing



- Stack grows 'down' in memory addresses… callee is at a lower address than caller
- Buffer grows 'up'… later characters are higher addresses

```
int main (int argc, char **argv) {
    printf("Enter a string: ");
    echo();
    return 0;
}
void echo() {
    char buf[8];
    fgets(buf, 16, stdin);
    printf("%s", buf);
}
```

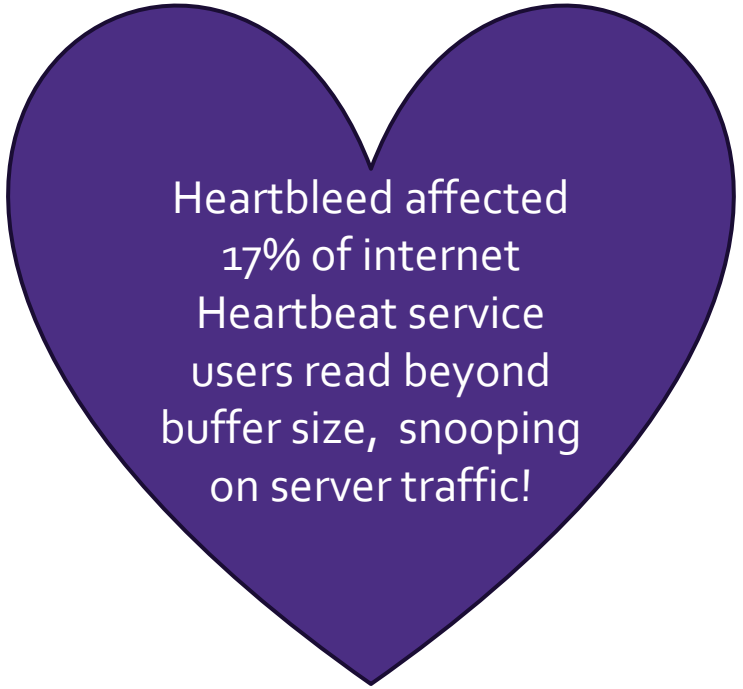
Stack smashing - SMASHED

- Stack grows 'down' in memory addresses… callee is at a lower address than caller
- Buffer grows 'up'… later characters are higher addresses
- If we write past the end of the buffer, we may write over IMPORTANT info.

```
int main (int argc, char **argv) {
    printf("Enter a string: ");
    echo();
    return 0;
}
void echo() {
    char buf[8];
    fgets(buf, 16, stdin);
    printf("%s", buf);
}
```

Exploits based on buffer overflows

- Examples across the decades
- Original "Internet worm" (aka Morris Worm) (1988)
  - Later became one of the founders of YCombinator.
- Heartbleed (2014, affected 17% of servers)
  - Similar issue in Cloudbleed (2017)
- Hacking embedded devices
  - Cars, Smart homes, Planes



Buffer overflow bugs can allow attackers to execute arbitrary code on victim machines

Demo: buffer play

break bufferplay

x buffer1 // prints the location of buffer1

info frame // Look at "rip"

print <rip-location> - <buffer1-location>

// prints distance from buffer1 to rip

disassemble main // shows the machine

// code and how many bytes each

// instruction takes up.

```
void bufferplay (int a, int b, int c) {
  char buffer1[5];
  uintptr_t ret;  // holds an address
  //calculate the address of the return pointer
  ret = (uintptr_t) buffer1 + 0;
  //treat that number like a pointer
  *((uintptr_t*)ret) += 0; // add amount to
  advance
}

int main(int argc, char** argv) {
  int = 0;
  printf("before: %d\n",x);
  bufferplay (1,2,3);
  x = 1;  // want to skip this line
  printf("after: %d\n",x);
  return 0;
}
```

Dealing with buffer overflow

Avoid vulnerabilities in the first place

- Use library functions that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
- Use a programming language that makes them impossible

System-level protections

- Make stack non-executable
- Have compiler insert "stack canaries" (just like a networking header checksum)
- Put a special value between buffer and return address
- Check for corruption before leaving function
- Randomized Stack offsets

WHAT ELSE CAN WE DO WITH MEMORY?

Create our own **`malloc`** and **`free`**!

HW Memory

- In C: malloc and free are wrappers to system calls that reserve space in memory, or cancel the reservation.

  (System calls deal with memory management, I/O stream management, access files, access the network.)

  But malloc and free are more user friendly than the essential system calls.

- Implement equivalents:
- // acts like 'malloc' and returns address in memory

```
void* getmem(uintptr_t size)
```

// acts like 'free' and releases memory

```
void freemem(void* p)
```

Note:
`Uintptr_t` is an integer type that holds a pointer.
`void*` is a pointer to an unspecified type

HWMemory: What we do

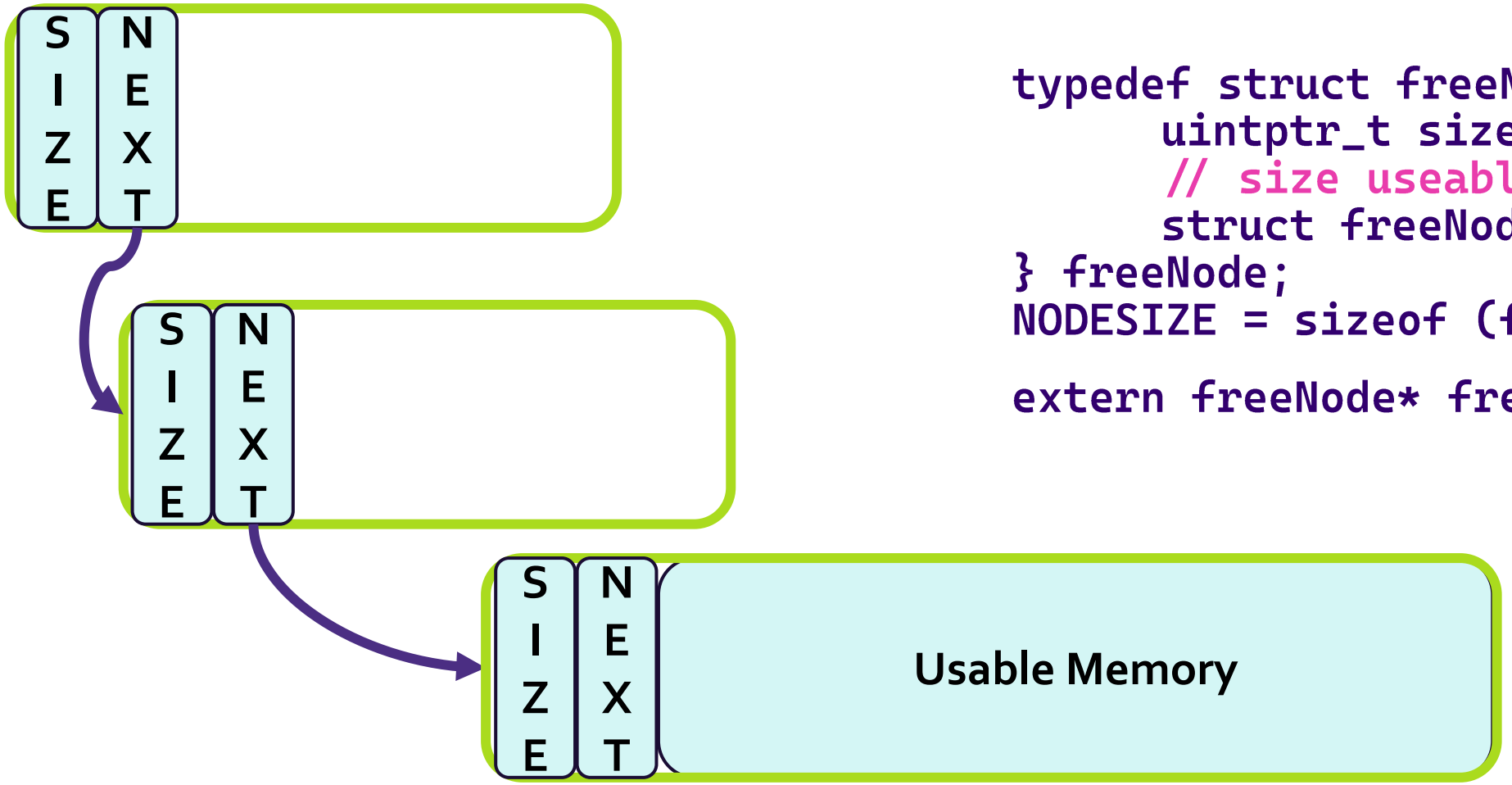
1) We use a system call (aka `malloc`) to get a big chunk of memory - like 4k-10k bytes.
2) We then parcel out pieces of this chunk to individual calls to `getmem` and mark them as reserved.
3) When someone calls `freemem`, we return the chunks to the set of free chunks.
4) We keep track of all of the available memory chunks?

- Use a "free list", which is a linked list of nodes that store information about available chunks.
- `freelist` shared by both `getmem` and `freemem`.
- Each block on the free list starts with an uintptr_t integer that gives its size followed by a pointer to the next block on the free list.
- To keep data in allocated blocks properly aligned, we require that all blocks be a multiple of 16 bytes in size, and that their addresses also be a multiple of 16 (this is the same way that the built-in malloc works).

HWMemory: Using getmem & freemem

- **Getmem request?**
  Scan the free list looking for a block of storage that is at least as large as the amount requested, return a pointer to it's usable memory to the caller and remove that chunk from the freelist
- **Freemem call?**
  Return the given block to the free list, combining it with any adjacent free blocks if possible to create a single, larger block of available memory.

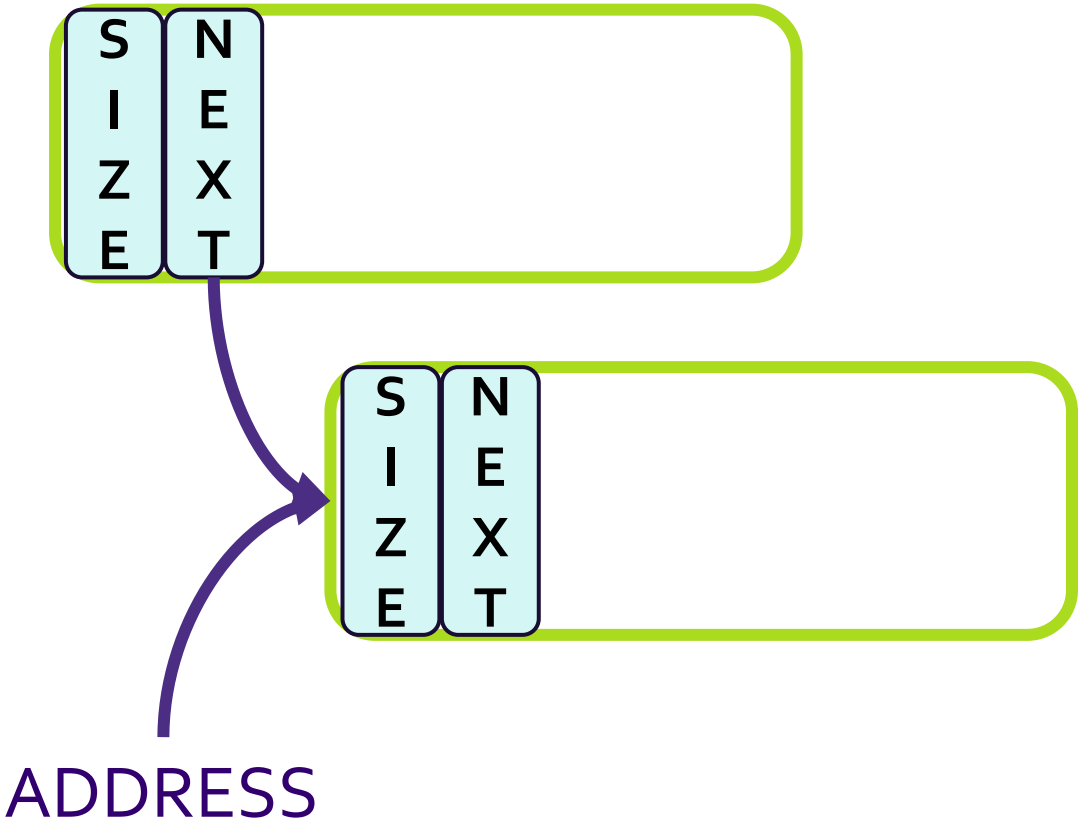
Memory Frame




```
typedef struct freeNode {
      uintptr_t size;
       // size useable memory
      struct freeNode* next;
} freeNode;
NODESIZE = sizeof (freeNode) //16

extern freeNode* freelist;
```

Memory Frame Address



What is the address?

- An integer pointing to the correct byte (`uintptr_t`)
- A pointer to a memory object (`void*`)

What can you do with it?

- Math - add or subtract an integer to go forward or backwards
- Cast between integer and (`T*`)
- If cast to `(freeNode*)` - access data of that type `freeNode→size, freeNode->next`

HWMemory: Using getmem

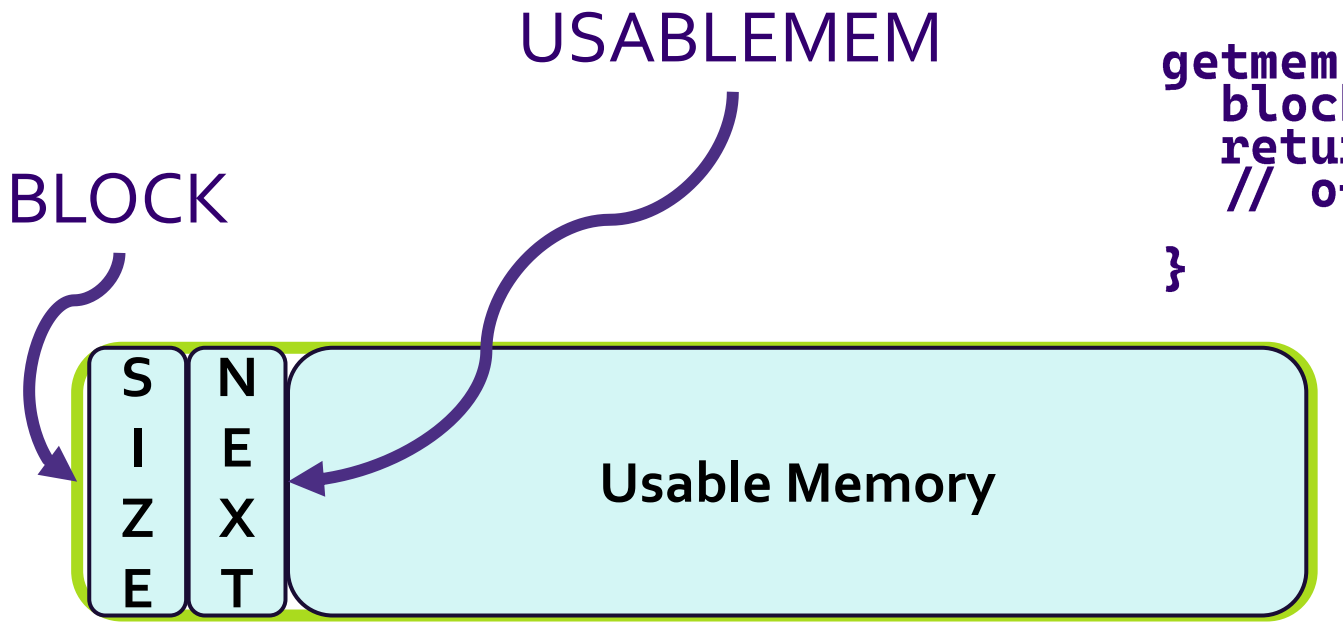
- **Getmem request?**
  Scan the free list looking for a block of storage that is at least as large as the amount requested, return a pointer to it's usable memory to the caller and remove that chunk from the freelist
  - If the block is large enough, split it into two chunks
    - One block is the right size to meet the request – return that one
    - Keep the remaining block in the free list
  - `else` if no block is large enough
    - Use malloc to get another large block & add it to the freelist
      - Go to case A
    - FIRST call to getmem results in `else` case

Get Block

```
get_block (uintptr_t size) {
  freeNode* currentNode = freelist;
  while(currentNode) {
    if(currentNode→size ≥ minsize)
      ...
    return(uintptr_t)currentNode;
}

getmem (uintptr_t size) {
  block =get_block (uintptr_t size);
  return((void*) block+NODESIZE);
  // offset for user's purposes

}
```

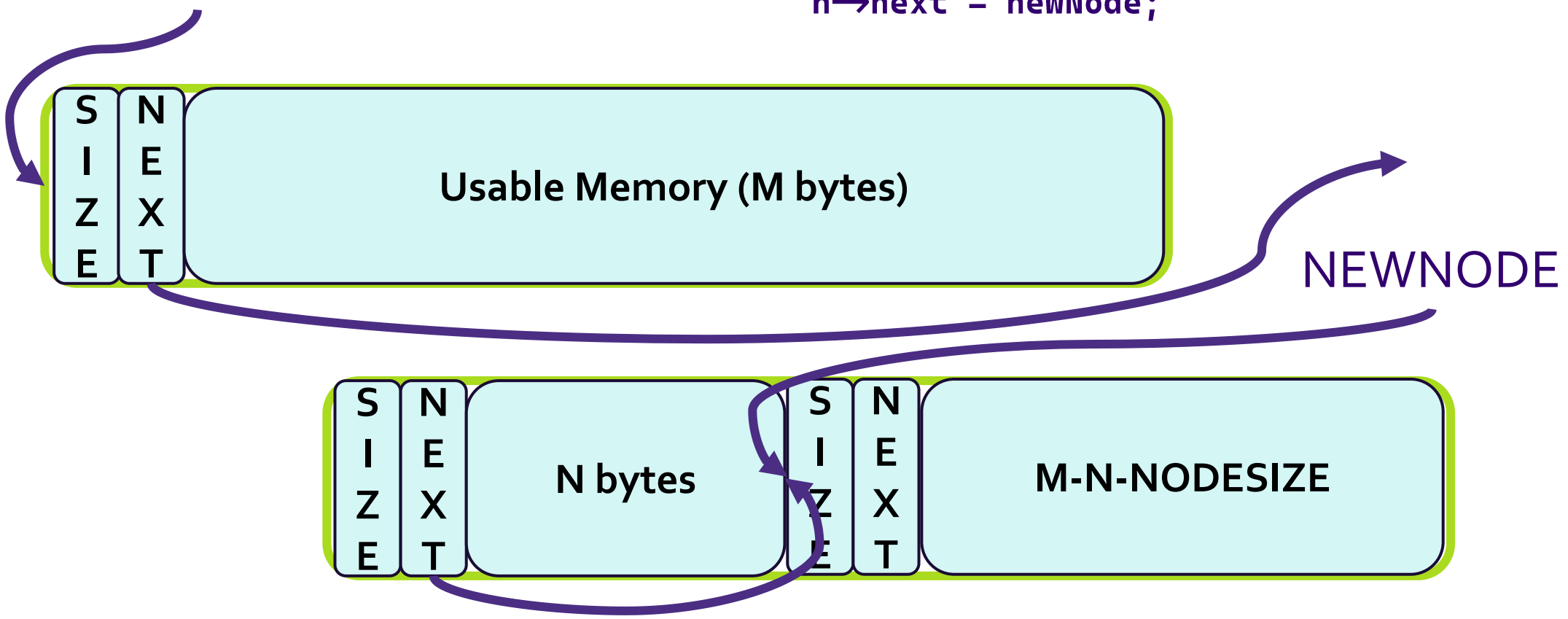


Split Frame

```
void split_node(freeNode* n, uintptr_t size) {
  freeNode* newNode =
   (freeNode*)((uintptr_t)(n) + size+NODESIZE);

  newNode→size = n→size - size - NODESIZE;
  newNode→next = n→next;

  n→size = size;
  n→next = newNode;
```

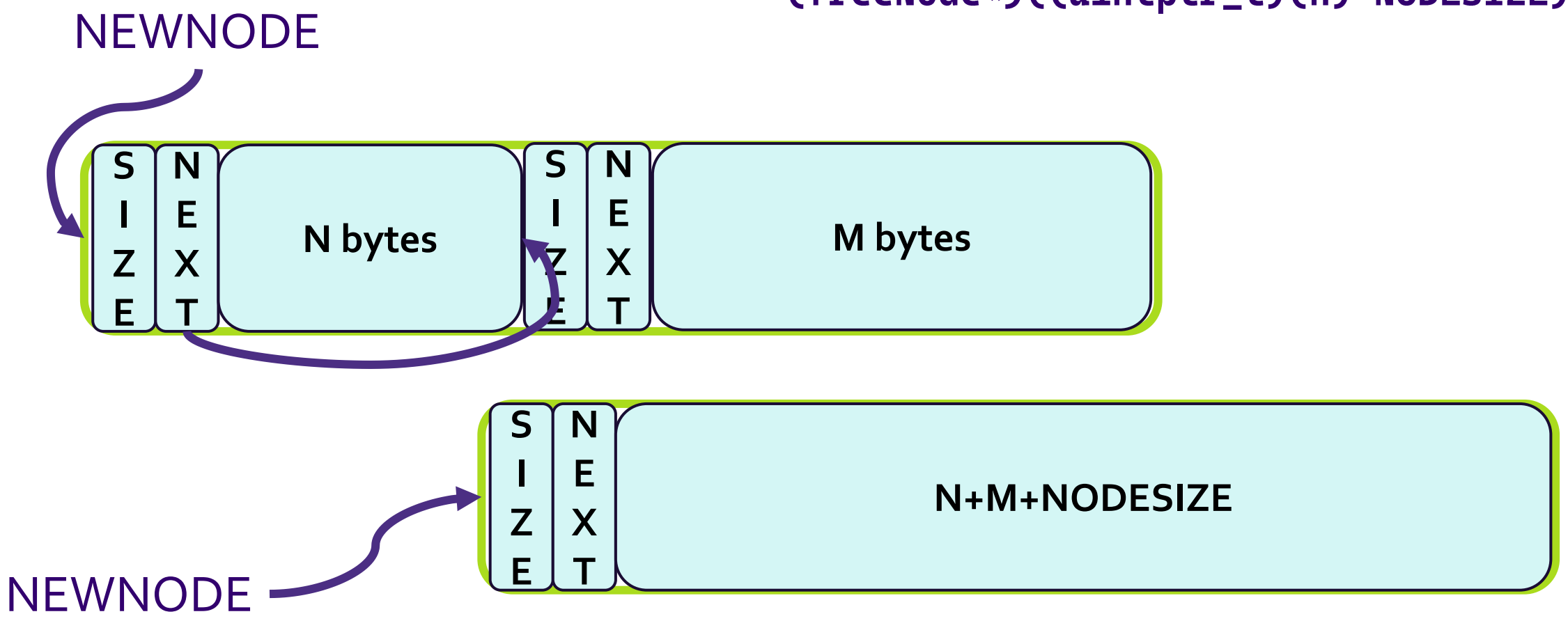


HWMemory: Using freemem

- **Freemem call?**
  Return the given block to the free list, combining it with any adjacent free blocks if possible
- The value passed to freemem is the address of the usable memory
- As usual, we have a freeNode stored right before the usable memory
  - Can calculate the location of the freeNode using pointer arithmetic
  - And then use it as a freeNode to retrieved the size.
- You'll want to insert into the freelist. You may want to merge with an adjacent node, undoing what you do when you split a node

Return Node

```
void freemem(void* n) {
  freeNode* newNode =
   (freeNode*)((uintptr_t)(n)-NODESIZE);
```

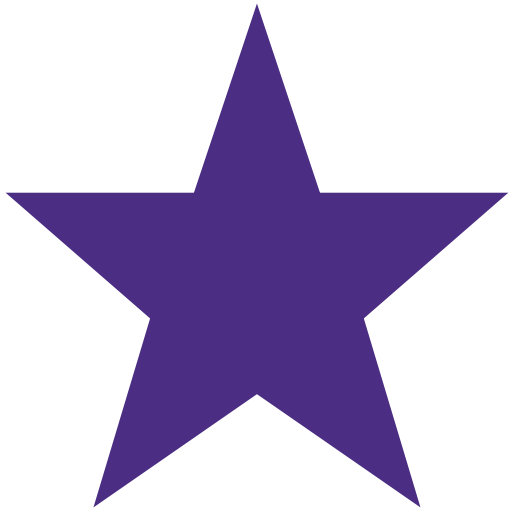


Freelist invariants

Check for possible problems with the free list data structure. This function should use asserts to verify that:

- Blocks are ordered with increasing memory addresses
- Block sizes are positive numbers and no smaller than whatever minimum size you are using
- Blocks do not overlap (the start + length of a block is not an address in the middle of a later block on the list)
- Blocks are not touching (the start + length of a block should not be the address of the next block on the list)
- If no errors are detected, this function should return silently after performing these tests. If an error is detected, then an assert should fail and cause the program to terminate at that point.

```
void check_heap() {
  if (!freelist) return;
  < …….>
  assert (mins ≥ MINSIZE);
}
```



EXTRAS

Memory Model

Code Globals Heap-> <-Stack

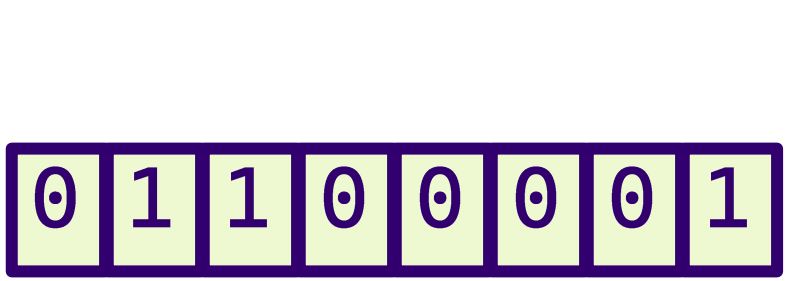
- Each byte of memory has an address
- Lowest memory stores program instructions, then global variables (static constants, string literals)
- 'Heap' holds dynamically allocated variables ('new' or 'malloc' variables)
- 'Stack' holds current instructions, each function in a frame
  - 'Stack' memory implies that a frame is added, and then the last frame added is removed first

Memory (zoomed in) (bkup)

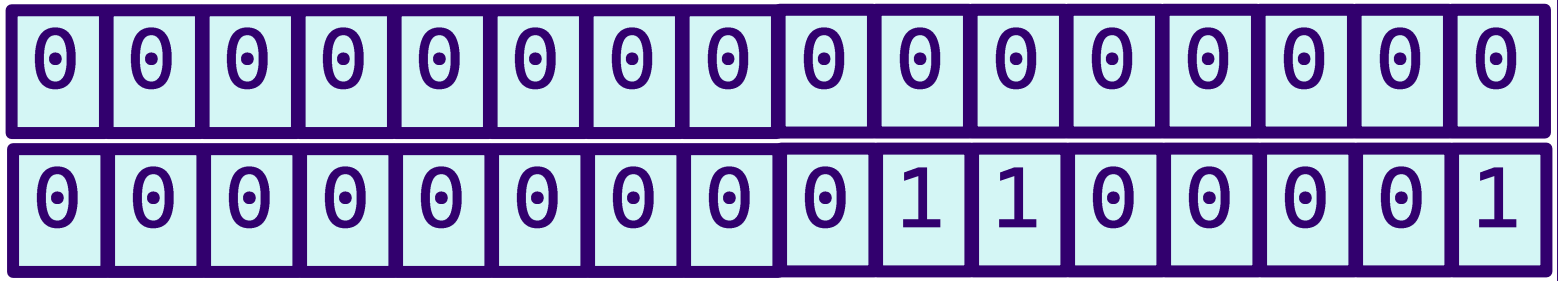
char – 1 byte **int – 4 bytes**

- Each byte of memory has an address
- Different data types need different amount of memory to store them
- All data types are stored as binary code: each of the 8 bits in a byte is a zero or one.

Memory (zoomed in even more)



char – 'a' in binary



int – 97 in binary

- All data types are stored as binary code: each of the 8 bits in a byte is a zero or one.

Remember Hexadecimal?

Base 10

- (10 fingers)
- 234
- $2x100 + 3x10 + 4x1$
- $2x10^2 + 3x10^1 + 4x10^0$

Base 2

- (binary/digital)
- 234 = 0b11101010
- $1x2^7 + 1x2^6 + 1x2^5 + 0x2^4\ 1x2^3 + 0x2^2 + 1x2^1 + 0x2^0$

3 digits in base 10, 8 digits in base 2, 2 digits in base 16

Base 16

- (hexadecimal)
- 234 = 0xEA
- $14x16^1 + 10x16^0$
- Need 16 digits, so use [0-9A-F]