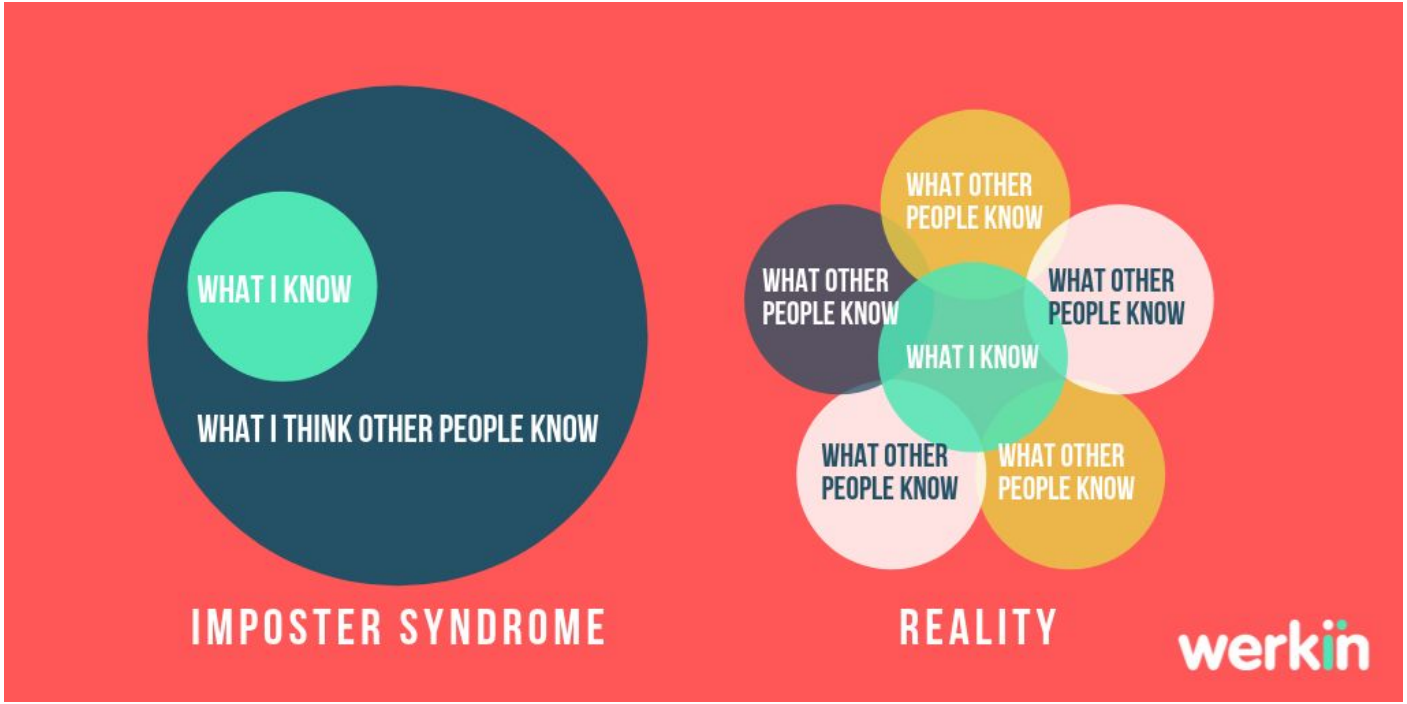
CSE 374: Lecture 22

Memory Management

Notes


WHAT I KNOW
WHAT I THINK OTHER PEOPLE KNOW
IMPOSTER SYNDROME
WHAT OTHER PEOPLE KNOW
WHAT OTHER PEOPLE KNOW
WHAT OTHER PEOPLE KNOW
WHAT I KNOW
WHAT OTHER PEOPLE KNOW
WHAT OTHER PEOPLE KNOW
REALITY
werkin

Review

Test 3

Compile for use with gdb:

```
$ gcc -g -Wall -std=c11 -o mystery5 mystery5.c
```

Homework 5

- GO BACK AND WATCH THE DEMOS
- Copy strings: `strncpy(dest, source, strlen(source)+1);`
- Interpretations of common errors often can be found with a web search
- Its worth fixing the easy stuff if you can't figure out a bug; sometimes it solves your problem.
- Every malloc needs a free.

Notes from Office Hours

- There is a starter code in git repository. Make sure to check those out.
    - Also true for HW6 & HW7)
- It is okay if you want to have extra c files or whatever, but the files in the starter code are sufficient. You are not expected to write extra files.
- Follow the spec to generate messages when there is no more T9noyms.
- Watch and re-watch lecture on linkedlist, read the sample code to get an idea.
- You can test out your Makefile on your machine - no need to submit to gradescope.
    - You can also follow suggestions about how to build a test suite which looks a lot like our autograder
- When running valgrind, make sure to actually pass in the dictionary and do several pattern matches. If you just do ./t9, you will not find memory leaks.

Allocating array memory

An array IS a pointer

A String is an array of char, terminating with \0

strsize returns length of string, minus the final \0 character

Allocate enough space for (strsize+1) chars

```
// copy original string

int strsize = strlen(s)+1;
// result = (char *)malloc(strsize);
result =(char*)malloc(strsize*sizeof(char));
printf ("sizeof char: %d \n", sizeof(char));
strncpy(result, s, strsize);

// from final_reverse.c, lect. 11
```

Using pointer manipulation for good

```
void fill_mem (void* ptr, size) {
  uintptr_t memadd = (uintptr_t) ptr;
  for (int i=0; i<16 && i <size; i++) {
    *((unsigned char*)(memadd+1)) = 0xFE;
  }
}
```

Buffer Overflow

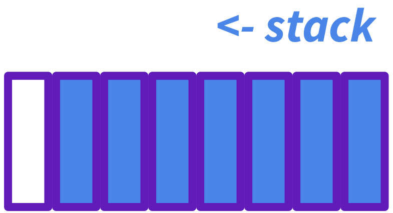
What is buffer overflow?

‘Gets’ doesn’t check for buffer size; if the string is more than 8 characters, it will write onto the memory at the end of buf.

Why is that so bad? (see the stack!)

```
void echo() {
        char buf[8];
        gets(buf);
        puts(buf);
}
```

The stack



Stack stores active functions & local variables

Each function gets a frame, moving down in memory

Last frame is completed, deleted

then the next most recent frame.

(Last in-first out)

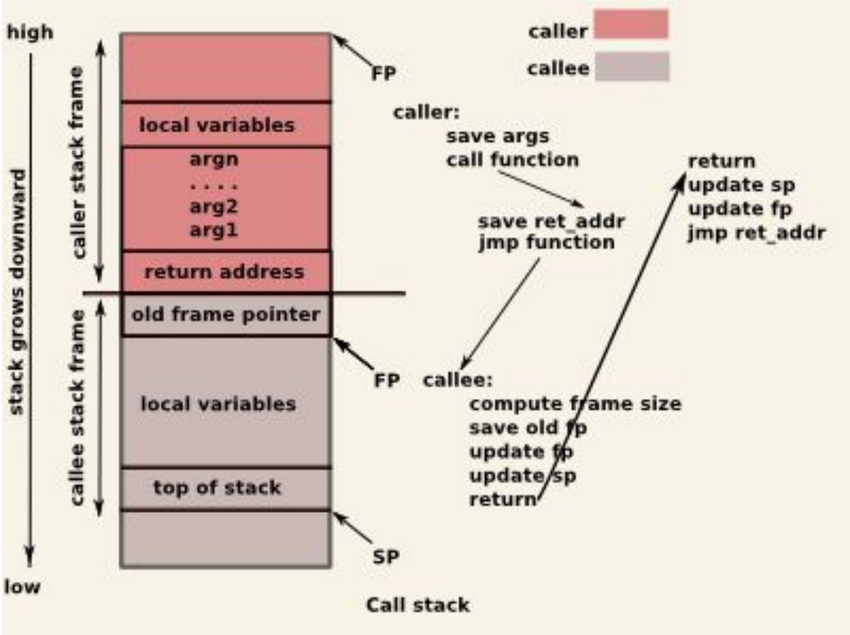
Each function call creates a frame

Containing:

Arguments, return address,

Pointer-to-last-frame,

local variables



Buffer Overflow

Writing past buf may overwrite other data, or the pointer to return to the calling code.

```
void echo() {
      char buf[8];
      gets(buf);
      puts(buf);
}
```

Change return to last frame

```
void bufferplay (int a, int b, int c) {
  char buffer1[5];
  uintptr_t ret;  // holds an address

  // calculate the return address
  // change to be address of return
  ret = (uintptr_t) buffer1 + 0;

  // treat that number like a pointer,
  // and change the value in it
  *((uintptr_t*)ret) += 0;
}

int main(int argc, char** argv) {
  int x = 0;
  bufferplay (1,2,3);
  x = 1;  // want to skip this line
}
```

Use GDB:

```
break bufferplay
x buffer1       // prints the location of buffer1
info frame      // Look at "rip" to get the
    // location of the return address
print <rip-location> - <buffer1-location>
    // prints distance from buffer1 to return
    //  address.

disassemble main  // shows the machine
    // code  and how many bytes each
    // instruction takes up.
```

Replace command at return address

```
int bar(char *arg, char *out) {
  strcpy(out, arg);
  return 0;
}
void foo(char *argv[]) {
  char buf[256];
  bar(argv[1], buf);
}

int main(int argc, char *argv[])
{
  foo(argv);
  return 0;
}
```

Idea:

Pass program a string in argv that contains nefarious code in a string

Take advantage of unprotected strcpy function so the return pointer on the stack is directed at the beginning of buf

When ‘foo’ exits, return ptr actually starts executing code passed in via string.

Defense against the dark-arts

- Avoid vulnerabilities in the first place.
    - Use library functions that limit string lengths
    - fgets instead of gets
    - strncpy instead of strcpy
    - %ns instead of %s in scanf
- System-level protections
    - Make stack non-executable
    - Have compiler insert “stack canaries”
    - Put a special value between buffer and return address
    - Check for corruption before leaving function

HW6

Forming teams NOW (on Canvas)

**** No Late Submissions ****

Project Due March 7

HW6

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

HW6: Approach

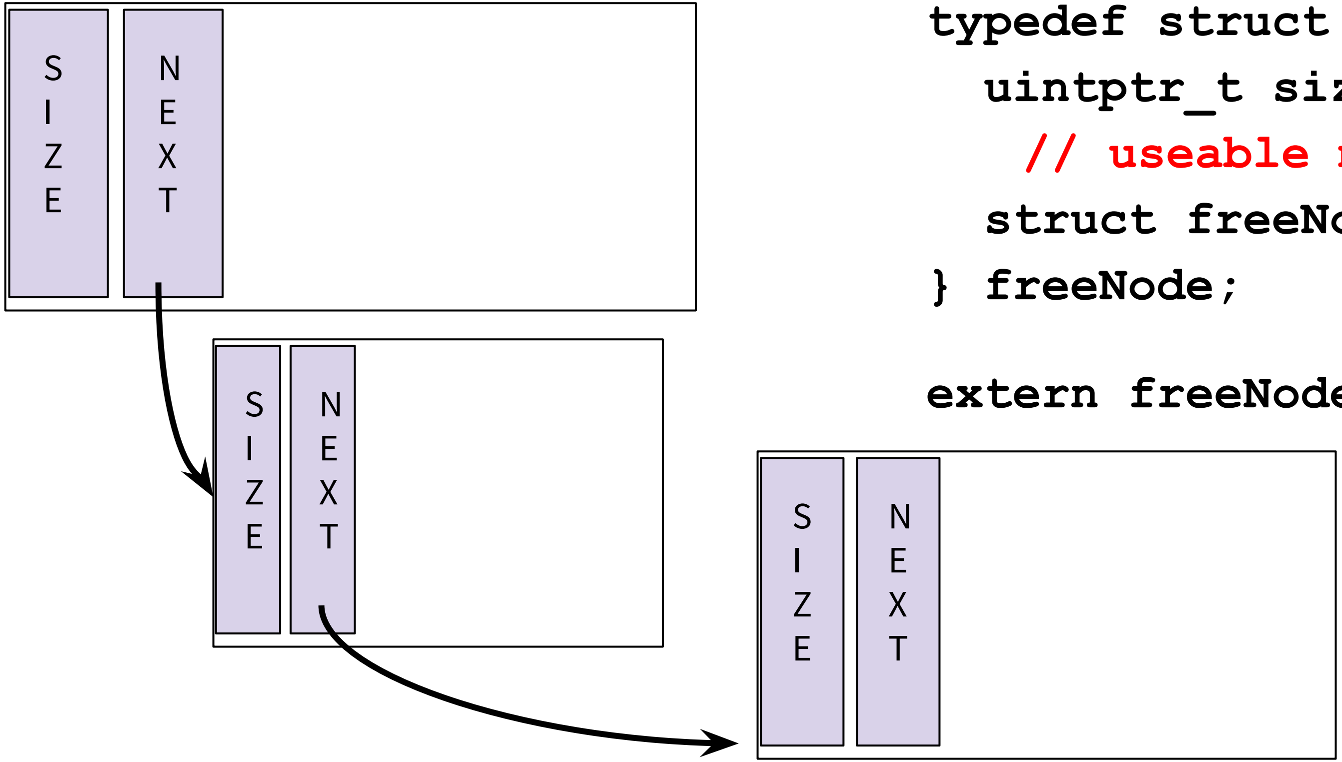
1. We use a system call (aka malloc) to get a big chunk of memory - like 4k-10k bytes.
2. We then parcel out pieces of this chunk to individual calls to getmem and mark them as reserved.
3. When someone calls freemem, we return the chunks to the set of free chunks.
4. How do we keep track of all of the available chunks vs reserved chunks?
   a. Use something called a "free list", which is a linked list of nodes that store information about available chunks.
   b. Shared by both getmem and freemem.
   c. 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.
   d. To help keep data in dynamically allocated blocks properly aligned, we require that all of the 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).

Approach, Cont.

Getmem request? Scan the free list looking for a block of storage that is at least as large as the amount requested, delete that block from the free list, and return a pointer to it to the caller.

Freemem: return the given block to the free list, combining it with any adjacent free blocks if possible to create a single, larger block instead of several smaller ones.

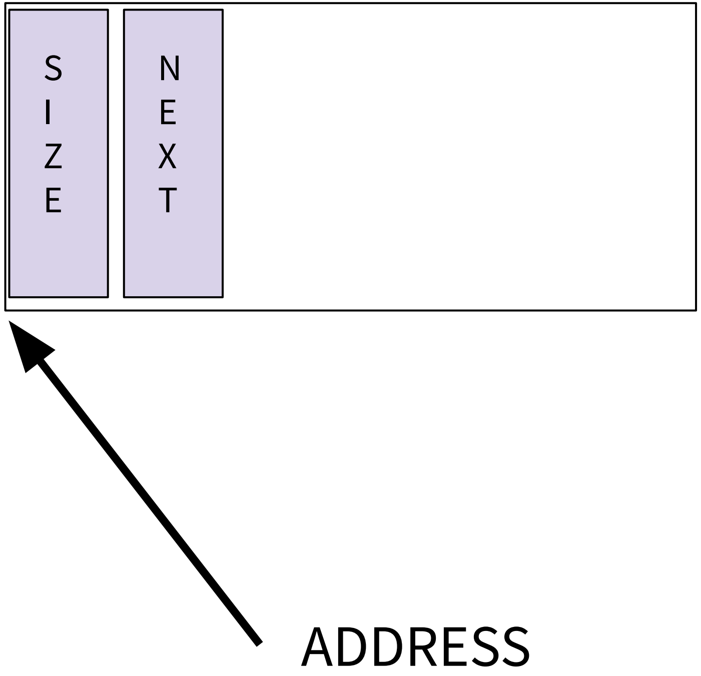
What is a memory frame?



```
typedef struct freeNode {
  uintptr_t size;
   // useable memory
  struct freeNode* next;
} freeNode;

extern freeNode* freelist;
```

Addresses



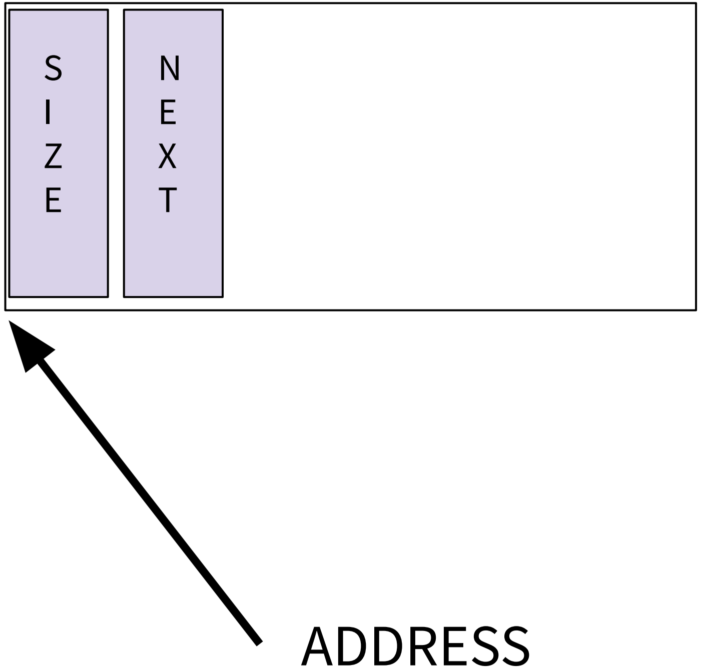
What is the address?

- 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`

getmem

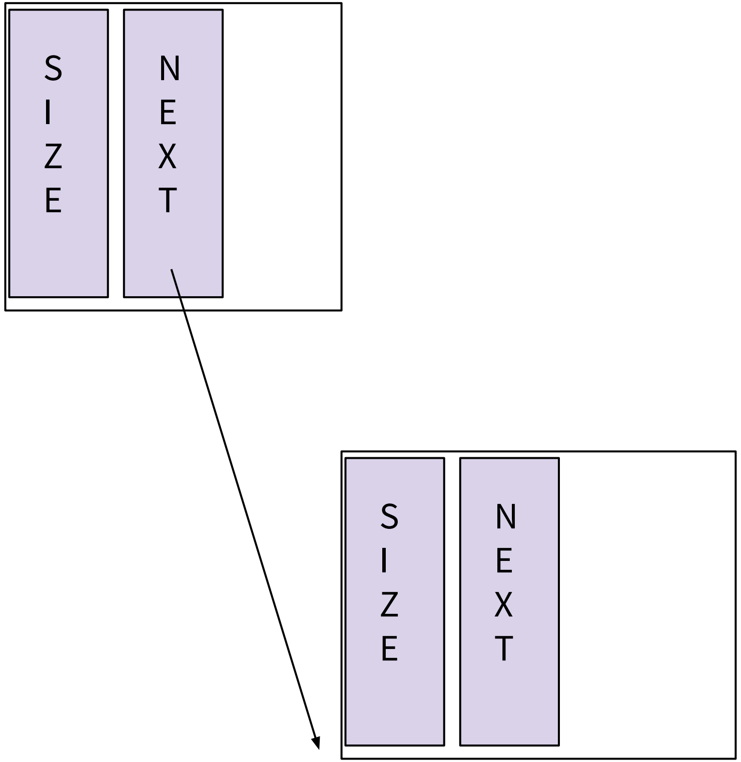


```
freeNode* currentNode = freelist;

get_block (uintptr_t size) {
while(currentNode) {
        if(currentNode->size >= minsize)
        ...
    return(uintptr_t)currentNode;
}

return((void*) block+NODESIZE);
// offset for user's purposes
```

getmem



```
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;

....
```

Approach: getting memory blocks

If, a large enough block exists, ‘getmem’ splits the block into an appropriate sized chunk and pointer to the block

Else, getmem needs to

Get a good-sized block of storage from the underlying system.

Add it to the free list

Split it up, yielding a block that will satisfy the request (‘**if**’ condition)

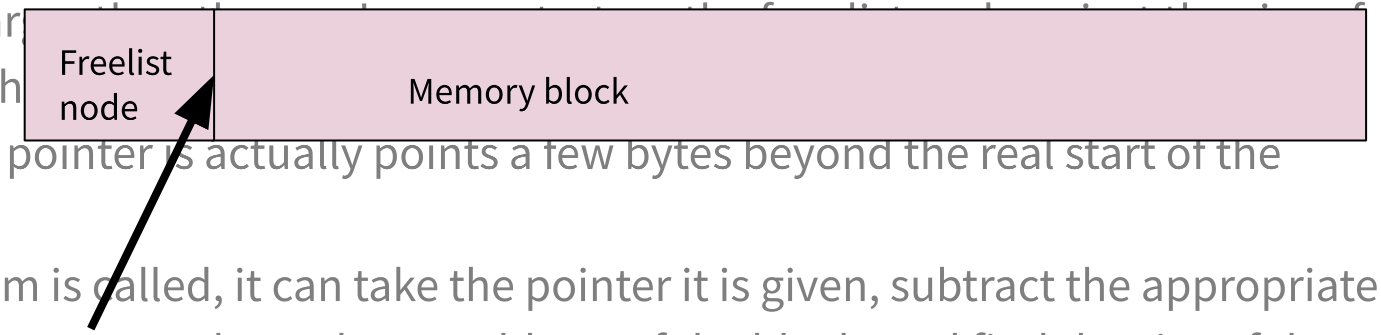
Note, Initial call to getmem finds it with no memory, and results in ‘**else**’ condition.

Approach: returning memory

- Freemem gets a pointer to a block of storage and adds it to the free list, combining it with adjacent blocks on the list.
- Freemem isn't told is how big the block is and must find the size of the block.
- The usual way this is done is to have getmem actually allocate a block of memory that is a bit larger than the user's request, store the free list node or just the size of the block at the beginning of that block.
- The returned pointer is actually points a few bytes beyond the real start of the block.
- When freemem is called, it can take the pointer it is given, subtract the appropriate number of bytes to get the real start address of the block, and find the size of the block there.

Approach: returning memory

- Freemem gets a pointer to a block of storage and adds it to the free list, combining it with adjacent blocks on the list.
- Freemem isn't told is how big the block is and must find the size of the block.
- The usual way this is done is to have getmem actually allocate a block of memory that is a bit lar
  the block at th



- The returned pointer is actually points a few bytes beyond the real start of the block.
- When freemem is called, it can take the pointer it is given, subtract the appropriate number of bytes to get the real start address of the block, and find the size of the block there.   `p-sizeof(freelist_node)->size`

Use ‘assert’ in C: `void check_heap ();`

Check for possible problems with the free list data structure. This function should use `assert`s 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;
   freeNode* currNode = freelist;
   uintptr_t mins= \
currNode->size;

   < …… .>
   assert (mins >= MINSIZE);
}
```

HW6 : using ‘extern’ (a shared global variable)

- Where does the free list head pointer live?
  - Needs to be accessible in both getmem and freemem implementation .c files. (Would normally be in the same module, but divided here for team work.)
- Could put it in a shared header file?
  - But, `int x;` allocates space for ‘x’ which is bad in a header file.
- Can we DECLARE ‘x’, but not DEFINE it?
  - Yes!: `extern int x;`
- Then in a .c file, you can actually define it (only in one file!).

HW6: Bench

Exercises your code: can use it as a test as you build up the other functions

Next up: C++ *(Want to read ahead?)*

Best place to start: C++ Primer, Lippman, Lajoie, Moo, 5th ed., Addison-Wesley, 2013

Every serious C++ programmer should also read: Effective C++, Meyers, 3rd ed., Addison-Wesley, 2005

Best practices for standard C++

Effective Modern C++, Meyers, O’Reilly, 2014

Additional “best practices” for C++11/C++14

Good online source: cplusplus.com

What is C++ ?

A big language - much bigger than C

Conveniences in addition to C (new/delete, function overloading, bigger std library)

Namespaces - similar to Java

Extras (casts, exceptions, templates, lambda functions)

Object Oriented - has classes and objects similar to Java

Why C++ ?

- C++ is C-like in
  - User-managed memory
  - Header files
  - Still use pointers
- C++ is Java like in
  - Object Oriented
  - Modern additions to language
- Knowing C++ may help understand both C & Java better