CSE 374: Lecture 23

Introduction to C++

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

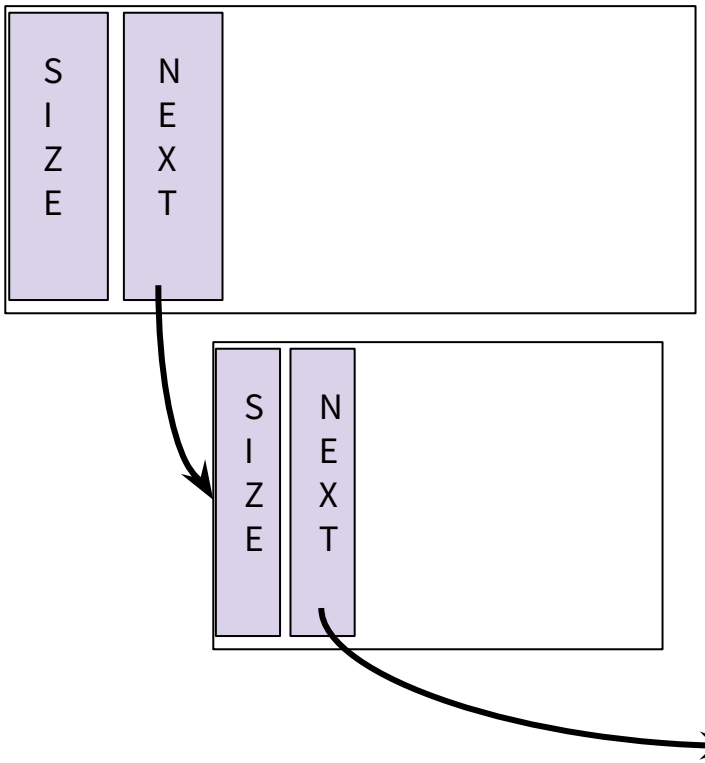
1. We use a system call (aka malloc) to get a big chunk of memory - like 4k-10k bytes.
2. We then parcel out pieces of this chunk to individual calls to getmem and mark them as reserved.
3. When someone calls freemem, we return the chunks to the set of free chunks.
4. How do we keep track of all of the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

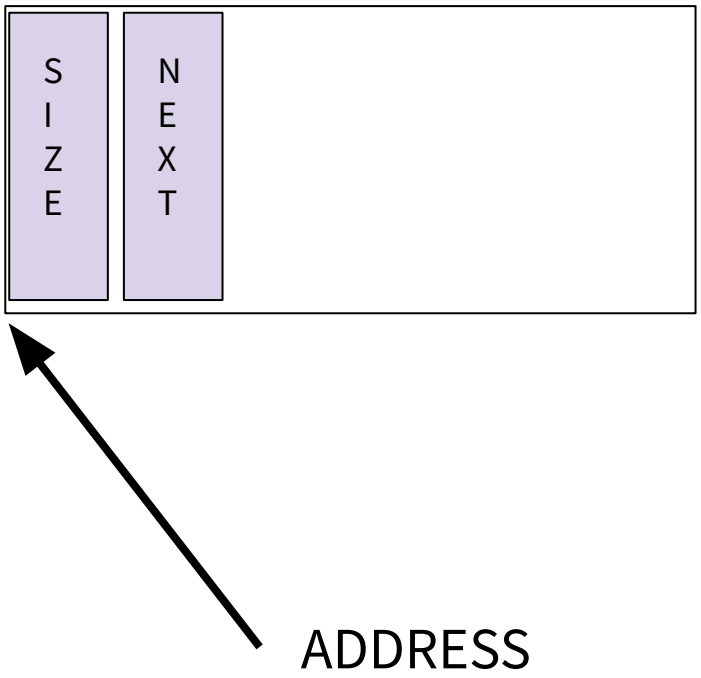
What is a memory frame?




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

extern freeNode* freelist;
```

Addresses



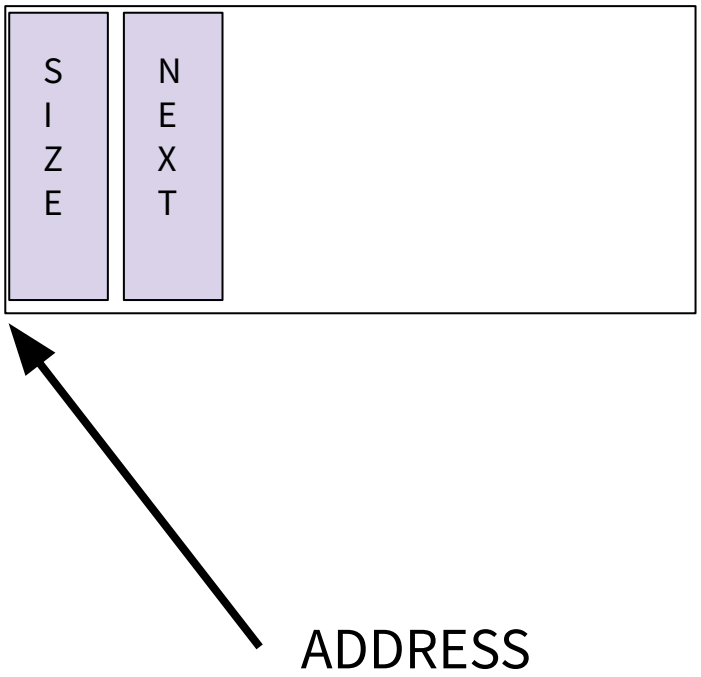
What is the address?

- 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

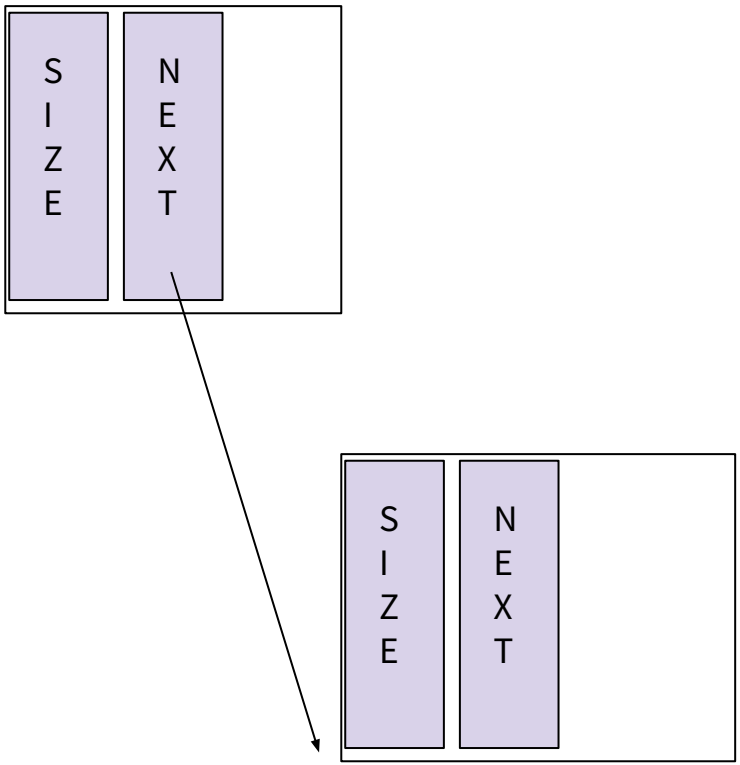



```
freeNode* currentNode = freelist;

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)

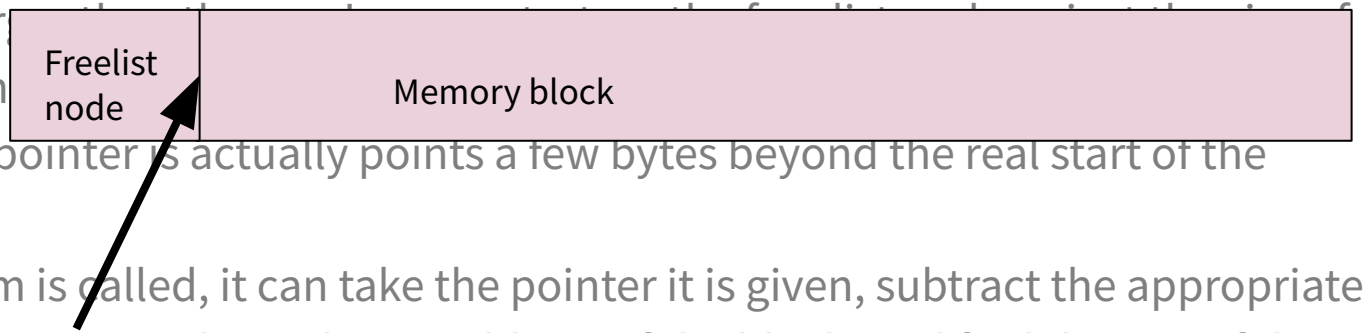
Note, Initial call to getmem finds it with no memory, and results in ‘**else**’ condition.

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

Object Oriented Programming

- **Encapsulation**
  - Discrete portions of code keep state and implementation private while providing public interfaces
- **Abstraction**
  - The high-level interface is exposed to users without detailing underlying code.
- **Inheritance**
  - Classes can be derived from other classes allowing for shared code.
- **Polymorphism**
  - Subclasses implement methods of superclasses to allow for a consistent interface.

C++ Hello, World!

```
#include <cstdlib>
#include <iostream>

const int CURRENT_YEAR = 2019;

using namespace std;

// REFERENCE
void pig(string& s) {
  char first = s[0];
  s = s.substr(1);
  s += first;
  s += "ay";
}
```

```
int main() {
  // stack-allocated array: int arr[100];
  // C++ style heap allocation:
  int* arr = new int[100];

  // C++ style array deletion:
  delete [] arr;
  // Use "delete x;" for things non-arrays.

  cout << "What is your name? ";
  string name;
  cin >> name;
  pig(name);
  cout << "What year were you born? ";
  int year;
  cin >> year;
  const int age = CURRENT_YEAR - year;
  cout << "Hello, " << name << "!" << endl;
  cout <<"You're "<<age<< " years old"<< endl;
  return EXIT_SUCCESS;
}
```

So, what different with C++?

- File Names (instead of *.c)
  - *.cc or *.cpp or *.cxx
- Compiler (instead of gcc)
  - $g++
- Preprocessor (still uses C preprocessor)
  - But #include <cstdlib>

- Still use *.h for header files

- Basically does the same thing as <stdlib.h>

Namespaces

- Group code logically
- Can re-use names for each namespace
- Can nest namespaces
- Disambiguate with :: syntax
- Can avoid using the prefix with `using namespace foo`
  `doSomething(3)`
- If you are using a namespace in a header, you must also use the namespace in the source code (.cpp)

```
namespace foo {
    int doSomething(int x);
}

namespace bar {
    int doSomething(int x);
}

int main() {
    foo::doSomething(3);
    bar::doSomething(3);
}
```

I/O in CPP

Std library include a `cout` and a `cin` function

Operators ‘>>’ and ‘<<’ act like shell redirection

Operators ‘>>’ and ‘<<’ take left and right operands and return a stream

Use namespace std or

use `std::cout` & `std::cin`

```
using namespace std

cout << "What is your name? ";
string name;
cin >> name;

cout << "When were you born? ";
int year;
cin >> year;
```

Pass by reference

- In C: all function arguments are copies
  - Pointer arguments pass a copy of the address value
- In C++: Can do the above
  - but can also use a “reference parameter” (& character before var name)
  - As though the calling line wrote pig(&name) and in ‘pig’ every ‘s’ is a ‘*s’

```
void pig(string& s) {
  char first = s[0];
  s = s.substr(1);
  s += first;
  s += "ay";
}
```

```
string name;
cin >> name;
pig(name);
```

Const

In C++ we also have the new "const" keyword, which says "this thing must not change". We can use this to declare global constants:

```
const int CURRENT_YEAR = 2018;
```

Global constants look a lot like global variables, but they are OK stylistically whereas regular global variables are not because the "const" keyword says that this value CANNOT CHANGE.

```
// This won't compile.
CURRENT_YEAR = 2038;
```

New / delete

In C:

```
int* arr = (int*) malloc(sizeof(int) * 100);
free(arr);
```

In C++, we have a nicer syntax for this that does the same thing:

```
int* arr = new int[100];
delete [] arr;
```

We can also do this for non-array types:

```
int* x = new int(4);        // x stores the value 4.
delete x;
```

Arrays

- Create a heap-allocated array of objects: `new A[10];`
  - Calls default (zero-argument) constructor for each element
  - Convenient if there's a good default initialization
- Create heap-allocated array of pointers to objects: `new A*[10];`
  - More like Java (but not initialized?)
- As in C, `new A()` and `new A[10]` have type `A*`
- `new A*` and `new A*[10]` both have type `A**`
- Unlike C, to delete a non-array, you must write `delete e`
- Unlike C, to delete an array, you must write `delete [] e`

Resources

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