P2 - Memory Management

COP4600

Slides by: Ernesto A. Mujica

Memory Management

- This project is entirely contained within a single library.
- Other programs should be able to include and link with your library to allocate any memory they might need, instead of using new or delete themselves.
- This project, unsurprisingly, needs you to be comfortable with dynamic memory and pointers.
- If you are at all rusty on dynamic memory, pointers, or pointer arithmetic, please brush up. Those videos should help.

“word” vs byte

- Most functions in this library have units in **words**, not bytes. However, ‘words’ are always going to be a whole number multiple of bytes. The size of a single ‘word’ in your class will be decided at object construction, denoted by the variable **wordSize**, of type ‘unsigned’ (shorthand for unsigned int).
- You will be converting between bytes and words often in this project. If given a number in words, you can convert to bytes as follows:
- numberOfWords * wordSize = numberOfBytes

“word” vs byte (cont.)

- Sometimes there might be a situation in which you may need to fit a certain number of bytes **unevenly** in a number of words. For example:
- wordSize = 4 bytes, and I need enough words to fit 14 bytes.
- 14 mod 4 = 2, meaning you need an extra word for the extra 2 bytes.
- This extra word will technically have 2 unused bytes in it, but that’s alright, and it does **not** count as a hole.

Blocks and Holes

- A **block** is an allocated portion of memory, a **hole** is an unallocated portion of memory.
- Your task in this project is to manage a contiguous, dynamically-allocated portion of memory, and “rent out” portions of that memory by managing the blocks and holes within that memory when allocate() and free() are called.
- On object initialization, you will create a single contiguous array of dynamic memory of size **sizeInWords * wordSize** bytes.

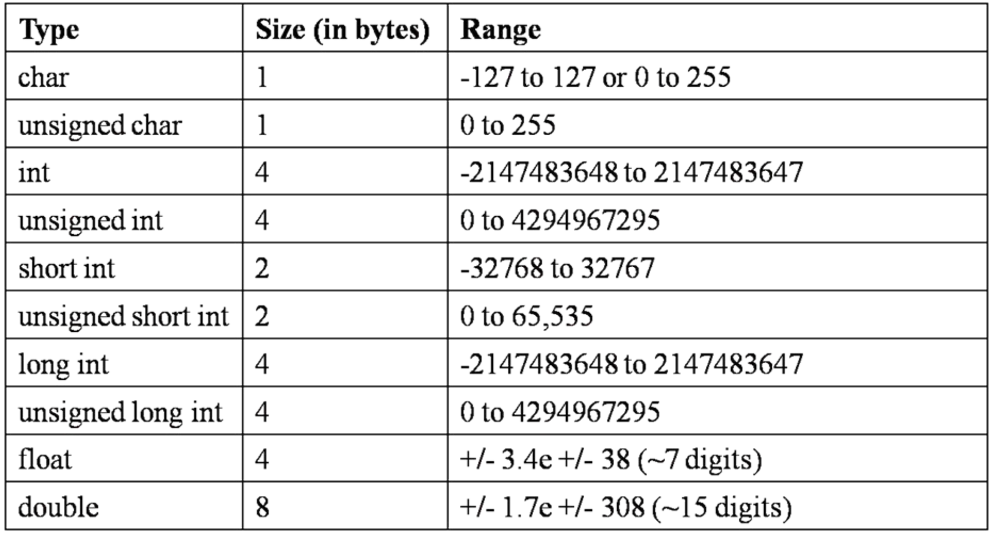
Blocks and Holes (cont.)

- You will need some kind of data structure to keep track of your memory blocks and holes. There are many, many different ways to approach this, and sometimes more than a single data structure may be useful to you (or a combination of them!).
- This project will be completed using **C++**, meaning it would be a good idea to leverage OOP here as well. At a minimum, you will be keeping track of the length and offsets of blocks and/or holes. (Possibly pointers to where they belong in your block as well.)
- It may be wiser to store information in abstracted structs/classes instead of multi-dimensional arrays or std::pair structures.
- Up to you though! We will not be testing you on your design skills, just whether or not it functions correctly when you submit it.

What is a void pointer?

- Several functions in this project either have void* parameters or void* return values.
- ‘void*’ is just a regular pointer. Pointers are memory locations, and pointer types are just there to tell you what they point to.
- If you know what kind of data was put into a particular void*, or what it represents, you can cast to its correct type to be able to use it properly. (Or cast from a different type to void* if necessary.)

C++ type size chart



Type	Size (in bytes)	Range
char	1	-127 to 127 or 0 to 255
unsigned char	1	0 to 255
int	4	-2147483648 to 2147483647
unsigned int	4	0 to 4294967295
short int	2	-32768 to 32767
unsigned short int	2	0 to 65,535
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	+/- 3.4e +/- 38 (~7 digits)
double	8	+/- 1.7e +/- 308 (~15 digits)

- You will need to have specific byte size types in arrays for this project. Look at the pdf and choose the correct type for the correct situation!

- Any of the types above are advisable, but for ease of use I recommend the uintX_t types, which you can use to specify exact size.
- Ex: uint8_t is a 1-byte integer, uint16_t is a 2-byte integer, uint32_t is a 4-byte integer (normal int), etc.
- You will want to instantiate your memory block with a 1 byte type, such as uint8_t or char.

So… what are we doing?

- You are creating a MemoryManager class that can be used as a replacement for the **C++** “**new**” keyword.
- A user of your library will **initialize** a MemoryManager object of a specific size, **sizeInWords**.
- That user can then call **allocate(sizeInBytes)** to request a certain amount of bytes of total space to store something. **allocate** will return a pointer to the start of that space of memory.
- When the user no longer needs that memory anymore, they will call **free(void* address)**, and free the memory they requested from **allocate.**

```
public MemoryManager(unsigned wordSize, std::function<int(int, void *)> allocator)
```

- Store wordSize and the passed in default allocator function as member variables.

```
public ~MemoryManager()
```

- Deletes all heap memory when the object falls out of scope. Never call the destructor directly.

```
public void initialize(size_t sizeInWords)
```

- Instantiates contiguous array of size **(sizeInWords** * **wordSize)** amount of bytes.
- If initialize is called on an already initialized object, call **shutdown** then reinitialize.
- Most of your other functions should not work before this is called.
  They should return the relevant error for the data type, such as void, -1, nullptr, etc.\

```
public void shutdown()
```

- If block is initialized, clear all data. Free any heap memory, clear any relevant data structures, reset member variables.

```
public void *getList()
```

- Returns a pointer to the start of a 2-byte array containing the list of holes in [offset, length] format. The first array element is the number of holes in the list.
- For example, given the following holes: **[0, 10] - [12, 2] - [20, 6]**
- You would return **[3, 0, 10, 12, 2, 20, 6]**

```
public void *allocate(size_t sizeInBytes)
```

- **sizeInBytes** is the amount of bytes to be allocated. May not always be a multiple of wordSize.
  Ex: **wordSize** = 8, and **sizeInBytes** = 26. 8 * 3 = 24, meaning that you need at least 4 words to store 26 bytes. That 4th word is considered to be populated, meaning it is not considered to have a hole.
- Will call your allocator function to find out where best to allocate the memory, meaning you will need to call **getList** too.
- Returns a pointer somewhere in your memory block to the starting location of the newly allocated space.

```
public void free(void *address)
```

- **address** is a pointer somewhere in your memory block to the starting location of the allocated space to be freed. It will be an address that you previously returned from **allocate**.
- Allocated space is getting freed, so remember to either extend your existing holes, or account for new ones.

```
public void setAllocator(std::function<int(int, void *)> allocator)
```

- Just a setter function. Changes your member variable to the new allocator.

```
public int dumpMemoryMap(char *filename)
```

- Prints out current list of holes to a file. You **must** use POSIX calls, you cannot use fstream objects.
- Use **open(filename, O_RDWR | O_CREAT | O_TRUNC, 0777)** to **create, enable read-write, and truncate** the file on creation. Remember to call **close** on the file descriptor before ending the function, or your changes may not save. For further reading: 1, 2.

`public void *getBitmap()`

- Returns a pointer to the start of an array of a 1-byte type that represents the current blocks and holes, using 1’s and 0’s for each word. 0 represents a hole, 1 represents an allocated block. The first two bytes in the bit stream represents the size of the array, in **little-Endian**.
- Given the following list of holes:
  [0,10]-[12,2]-[20,6]
- In words, using 1’s and 0’s, it will look like:
  00000000 00110011 11110000 00
- There are 26 words in this example, but each byte only stores 8 bits, so you need an extra byte to cover those last 2 bits.
  00000000 00110011 11110000 00000000
- You must now **mirror** each byte individually.
  00000000 11001100 00001111 00000000
- Now you add the two size bytes to the front. In this case, we have four bytes above, meaning that the size is 4.
- Representing 4 with two bytes looks like:
  00000000 00000010
- However, because it is in **little-Endian**, we must flip the order of the two bytes, yielding:
  00000010 00000000
- Now, we can complete the array.
  00000010 00000000 00000000 11001100 00001111 00000000
- In integer form, it looks like: [4, 0, 0, 204, 15, 0]. You will return a pointer to the beginning of this array.
-

```
public unsigned getWordSize()
```

- Returns **wordSize** member variable.

```
public void *getMemoryStart()
```

- Returns pointer to the start of your contiguous memory array.

```
public unsigned getMemoryLimit()
```

- Returns **unsigned int** (**unsigned** and **unsigned int** are the same type) that is the total amount of bytes you can store.

```
int bestFit(int sizeInWords, void *list)
```

- Allocator function, can be written inside MemoryManager.cpp but **does not belong to MemoryManager class.**
- **list** will be structured like the output from **getList.**
- Finds a hole in the list that **best fits** the given **sizeInWords**, meaning it selects the smallest possible hole that still fits **sizeInWords**.
- Returns word offset to the start of that hole.

```
int worstFit(int sizeInWords, void *list)
```

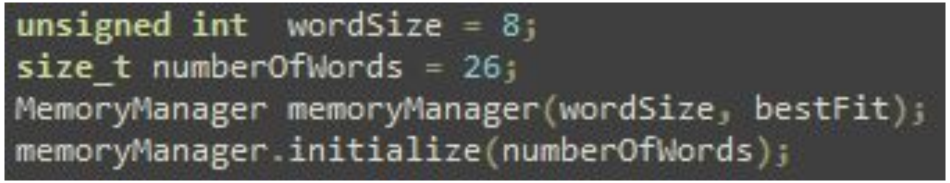
- Same as above, but finds largest possible hole instead.

Tips

- **Start early, and start small.**
- Ignore the complicated printing functions initially.
- Focus on getting the main functions like **initialize, shutdown, allocate, free, getList and the allocators** working first.
- Understand the big picture and plan out your data structure decisions first, before you start programming. Planning is very, very important in this project, so you don’t end up re-coding half of it to fix a bug that occurred because of lack of information/bad data structure management.

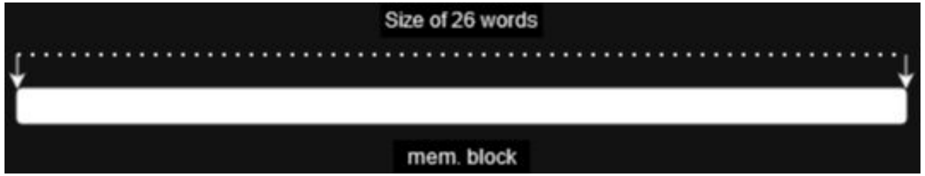
Test Case Example

- Ex: testSimpleFirstFit() from the testing file.



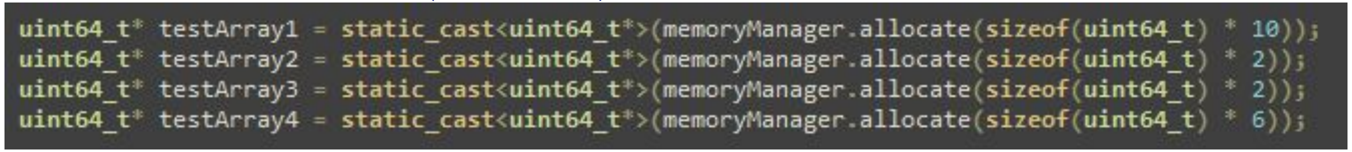
```
unsigned int  wordSize = 8;
size_t numberOfWords = 26;
MemoryManager memoryManager(wordSize, bestFit);
memoryManager.initialize(numberOfWords);
```

- Initializes a memory block of 26 words, each with wordSize of 8, meaning the block is of total size 206 bytes.



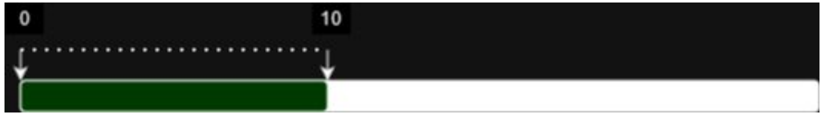
- At initialization, the block is essentially one big hole, since nothing is allocated yet.

Test Case Example (cont.)

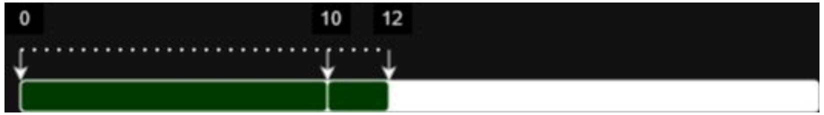


```
uint64_t* testArray1 = static_cast<uint64_t*>(memoryManager.allocate(sizeof(uint64_t) * 10));
uint64_t* testArray2 = static_cast<uint64_t*>(memoryManager.allocate(sizeof(uint64_t) * 2));
uint64_t* testArray3 = static_cast<uint64_t*>(memoryManager.allocate(sizeof(uint64_t) * 2));
uint64_t* testArray4 = static_cast<uint64_t*>(memoryManager.allocate(sizeof(uint64_t) * 6));
```

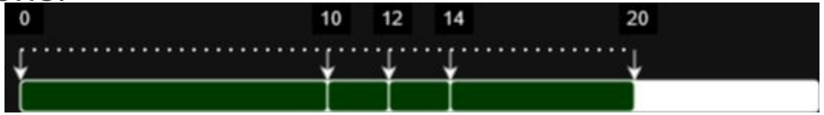
- Allocate is called 4 times. sizeof(uint64_t) is equal to 8, which is the same as wordSize here.
- We first allocate 80 bytes (10 words), then 16 bytes (2 words).. Etc.
- After the testArray1 allocation (numbers 0 and 10 here are in **words**):



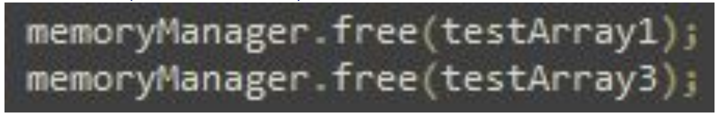
- After the testArray2 allocation:



- After all four allocations:

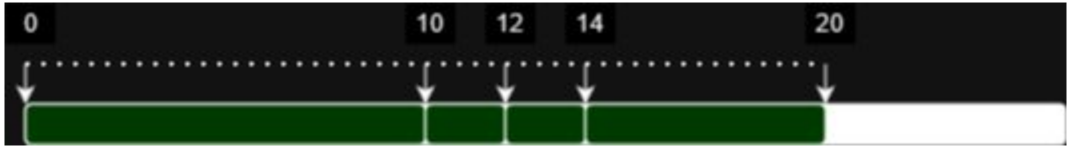


Test Case Example (cont.)

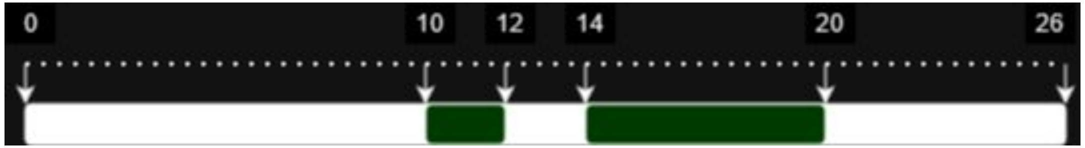


```
memoryManager.free(testArray1);
memoryManager.free(testArray3);
```

- Free is called on the first and third allocations. Therefore the block goes from this:



- To this:



- Meaning that you have 3 holes. Organized in [startingIndex, size] order, the holes are:

[0, 10] - [12, 2] - [20, 6]

- Remember that these units are in **words**.

Testing

- We will provide you with a testing file. It is **highly** recommended that you look at it and understand how it works. Understanding what it wants from you first will make developing your project significantly easier!
- Valgrind will tell you if you leak memory or have memory errors, and is crucial in this project for both ensuring that your submission is not dead on arrival, and for debugging.
- Compile your library (and executable) with debug flags as was taught to you in Ex3, and valgrind will even tell you the line of code that may be causing you memory leaks/errors.

Report/Screencast

- List all files, walk through **all** functions. Explain your data structures, and explain how offsets are calculated, how you use and manipulate pointers, and how blocks and holes are managed, created, and merged.
- Explain your POSIX calls in **dumpMemoryMap**, explain your testing process for memory errors/leaks.
- Demo the testing program in valgrind in your screencast. Go over everything the same way you do in your report, just verbally.

Project PDF

(Go over project PDF, functions, and show functionality with valgrind.)