Memory Manager Project

The goal of your next project is to simulate the C heap manager which is used to allocate and deallocate dynamic memory.

The heap is a large "pool" of memory set aside by the runtime system for a program.

The two main functions are

- **`malloc`**, used to satisfy a request for a specific number of consecutive blocks;

- **`free`**, used to make allocated blocks available for future **`malloc`** requests (i.e., return them to the pool of available memory).

Our simulation uses a large, initially allocated block of unsigned chars as our memory pool; and a doubly-linked list of blocknodes to keep track of allocated and available (free) blocks of unsigned char.

Each blocknode contains the size (number of bytes) of the block, whether it is free, and a pointer to the beginning of the allocated block within our memory pool.

The **`malloc`** algorithm scans the **`blocknode`** list until it finds the first free block whose size is at least the number **`request`** of bytes specified by **`malloc`**.

If the block size is the same as **`request`**, it merely changes the status of the block to "in use" and returns the pointer field of the **`blocknode`**.

If the block is larger than **`request`**, the block is broken up into two blocks.

- The first will consist of the first **`request`** bytes of the free block and **`free`** will be set to **`false`**;
- The second will be a free block of the remaining bytes of the original block.

It should be clear that you want to maximize the size of the free blocks.

Thus, anytime a freed block has a free predecessor or successor, the newly free block should be merged with any adjacent free blocks.

We next look at the main files of this project.

```cpp
// blocknode.h

#include <iostream>

using namespace std;

struct blocknode
{
	unsigned int bsize;
	bool free;
	unsigned char *bptr;
	blocknode *next;
	blocknode *prev;

	blocknode(unsigned int sz,unsigned char *b,bool f=true,
			blocknode *p=0,blocknode *n=0):
		bsize(sz),free(f),bptr(b),prev(p),next(n) {}
};
```

// MemoryManager.h

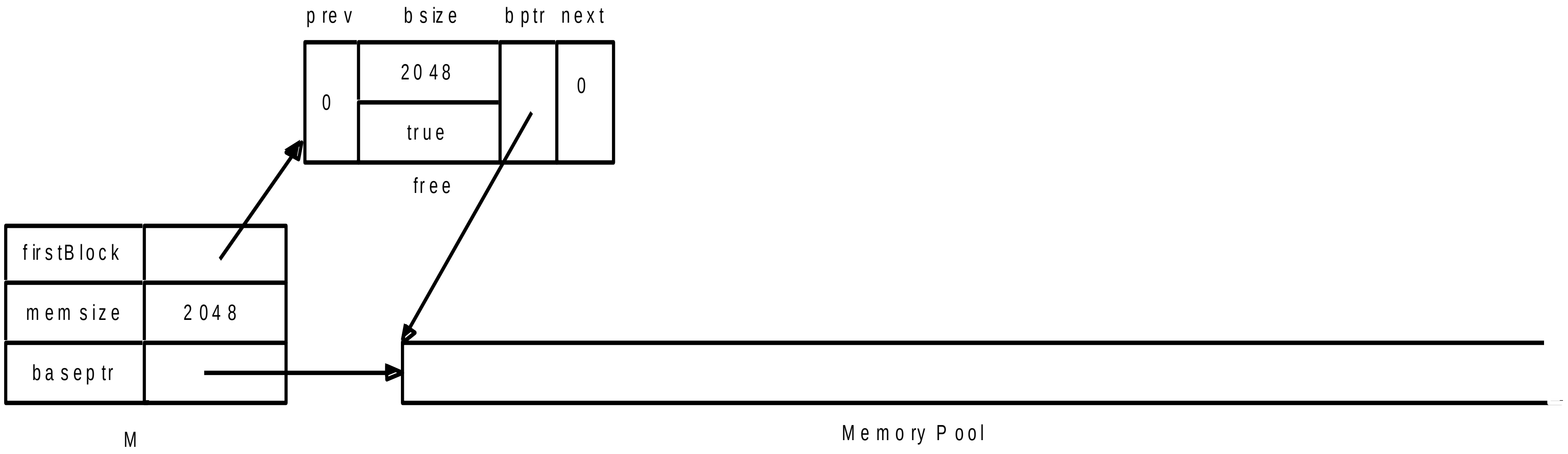
```
#include <iostream>
#include <sstream>
#include "blocknode.h"

using namespace std;

class MemoryManager
{
  public:
   MemoryManager(unsigned int memsize);
   unsigned char * malloc(unsigned int request);
   void free(unsigned char * blockptr);
   friend ostream & operator<<(ostream & out,const MemoryManager &M);

  private:
   unsigned int memsize;
   unsigned char *baseptr;
   blocknode * firstBlock;

   void mergeForward(blocknode *p);
   void splitBlock(blocknode *p,unsigned int chunksize);
};
```

```
// MemoryManager.cpp
#include <cassert>
#include <iostream>
#include "MemoryManager.h"

using namespace std;

ostream & operator<<(ostream & out,const MemoryManager &M)
{
   blocknode *tmp = M.firstBlock;
   assert(tmp);
   while(tmp)
   {
      out << "[" << tmp->bsize << ",";
      if (tmp->free)
               out << "free] ";
      else
               out << "allocated] ";
      if (tmp->next)
               out << " -> ";
      tmp = tmp->next;
   }
   cout << ']';
   return out;
}
```

MemoryManager

```
MemoryManager::MemoryManager(unsigned int memtotal):
       memsize(memtotal)
{
   baseptr = new unsigned char[memsize];
   firstBlock = new blocknode(memsize,baseptr);
}
```

```
  MemoryManager M(2048);
```

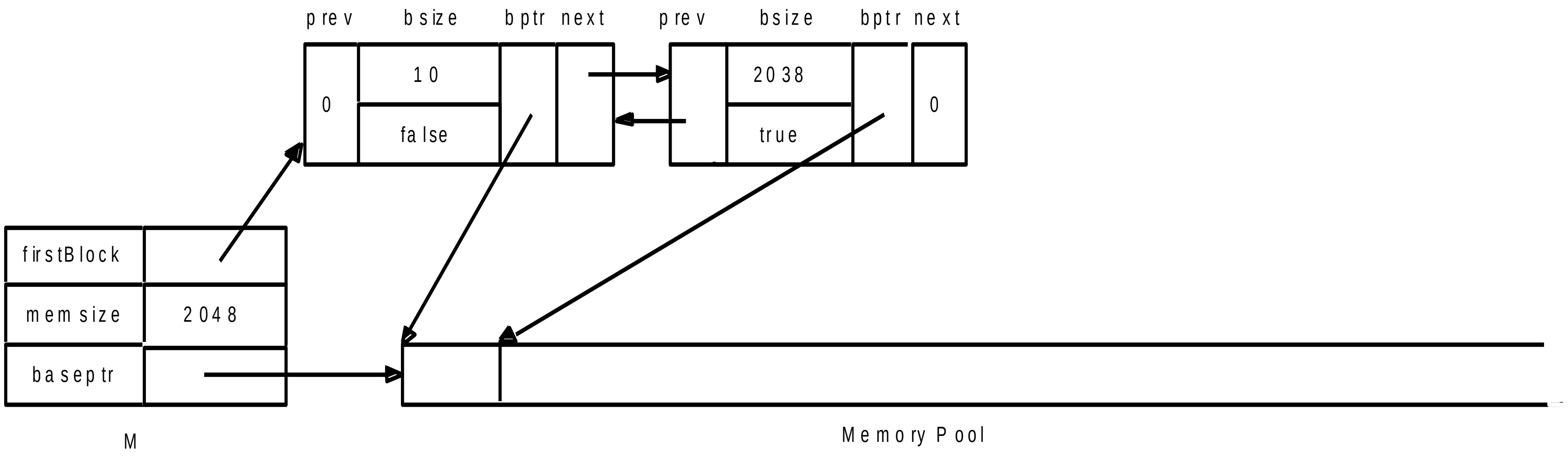


```
MemoryManager::MemoryManager(unsigned int memtotal):
memsize(memtotal)
{
   baseptr = new unsigned char[memsize];
   firstBlock = new blocknode(memsize,baseptr);
}
```

```
unsigned char * MemoryManager::malloc(unsigned int request)
// Finds the first block in the list whose size is >= request
// If the block's size is strictly greater than request
// the block is split, with the newly create block being free.
// It then changes the original block's free status to false
{

}

void MemoryManager::splitBlock(blocknode *p, unsigned int chunksize)
// Utility function. Inserts a block after that represented by p
// changing p's blocksize to chunksize; the new successor node
// will have blocksize the original blocksize of p minus chunksize and
// will represent a free block.
// Preconditions: p represents a free block with block size > chunksize
// and the modified target of p will still be free.

{

}
```

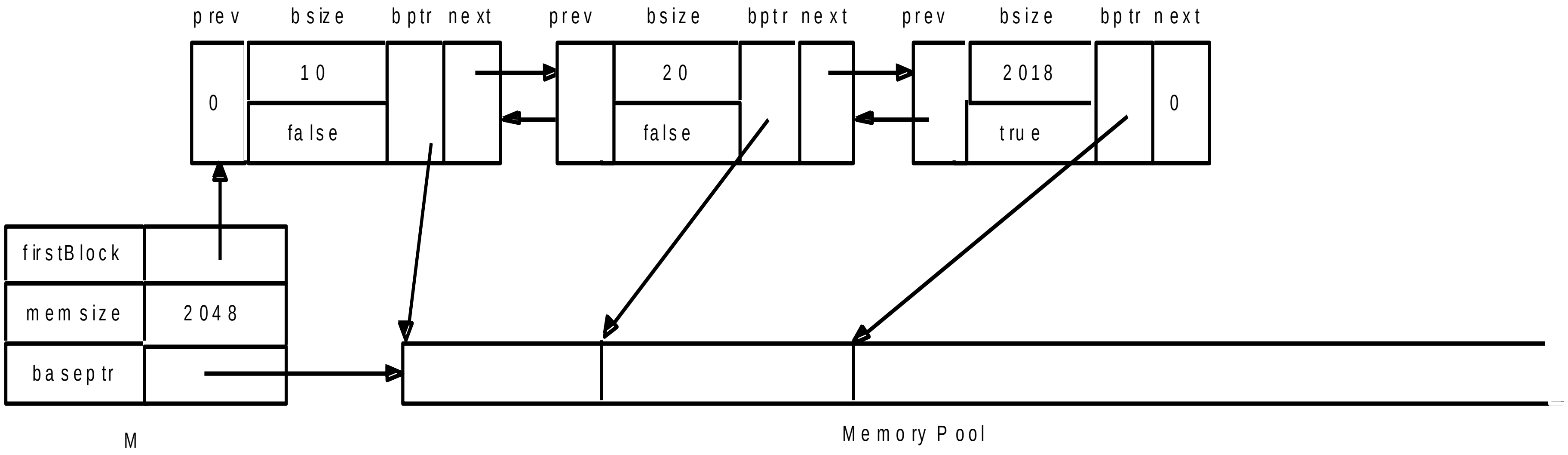
MemoryManager

```
unsigned char *p1 = M.malloc(10);
```



MemoryManager

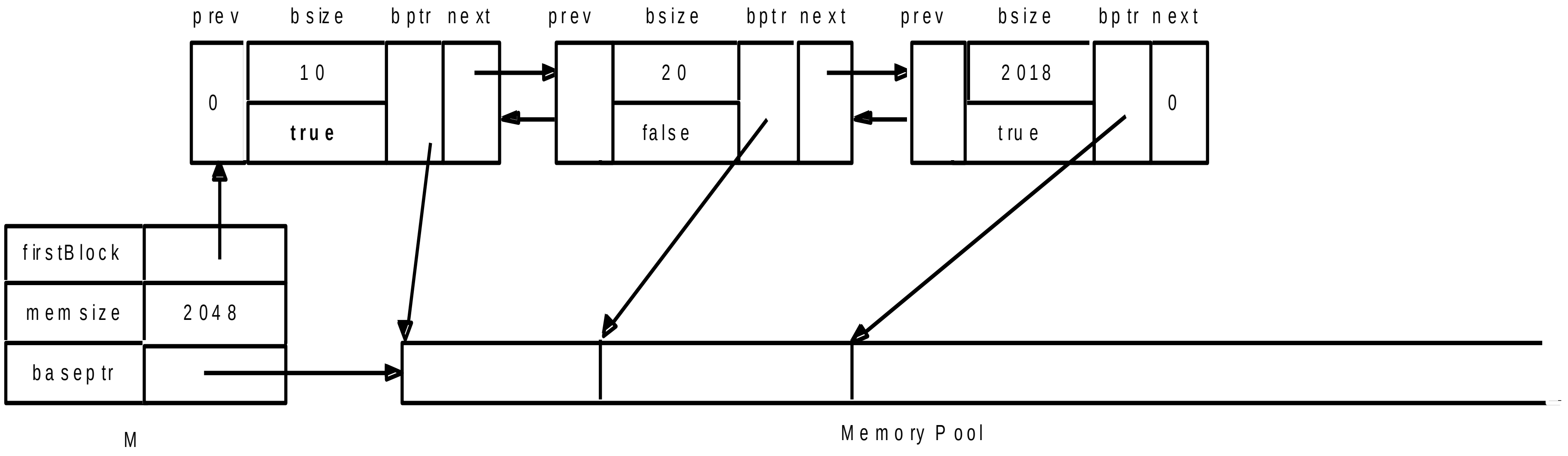
```
unsigned char *p2 = M.malloc(20);
```



```
void MemoryManager::free(unsigned char *blockptr)
// makes the block represented by the blocknode free
// and merges with successor, if it is free; also
// merges with the predecessor, it it is free
{
}

void MemoryManager::mergeForward(blocknode *p)
// merges two consecutive free blocks
// using a pointer to the first blocknode;
// following blocknode is deleted
{

}
```

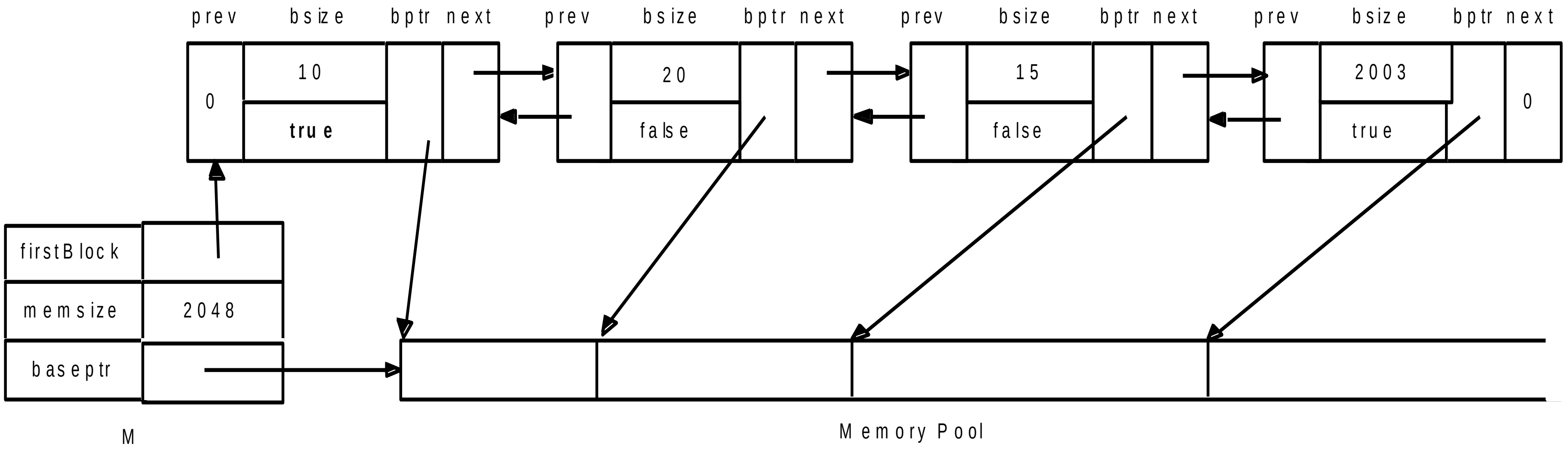
MemoryManager

`M.free(p1);`



MemoryManager

```
p1 = M.malloc(15);
```

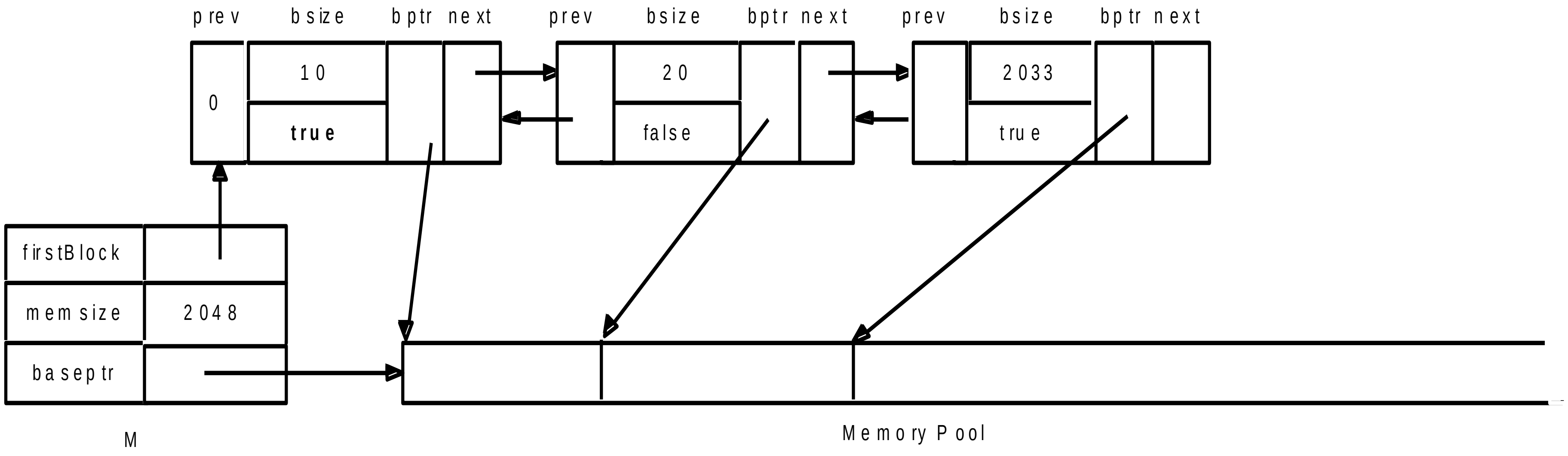


```
#include <iostream>
#include <cassert>
#include "MemoryManager.h"

int main()
{
   MemoryManager heaper(2048);
   cout << "heap initialized\n";
   cout << "Here is the block list\n";
   cout << "\n---------BlockList start--------------\n";
   cout << heaper << endl;
   cout << "-----------BlockList end--------------\n\n";
   cout << "Doing first malloc:\n";
   unsigned char * p1 = heaper.malloc(10);

   cout << "malloc done\n";
   cout << "Here is the block list\n";
   cout << "\n----------BlockList start--------------\n";
   cout << heaper << endl;
   cout << "------------BlockList end--------------\n\n";
cout << "On to the second malloc\n";
unsigned char *p2 = heaper.malloc(20);
cout << "malloc done\n";
cout << "Here is the block list\n";
cout << "\n-------------BlockList start----------------\n";
cout << heaper << endl;
cout << "-------------BlockList end------------------\n\n";

cout << "Next free the first pointer\n";
heaper.free(p1);
cout << "Here is the block list\n";
cout << "\n-------------BlockList start----------------\n";
cout << heaper << endl;
cout << "-------------BlockList end-----------------\n\n";

cout << "Now do a malloc for a block too big for the "
     << "initial open block\n";
p1 = heaper.malloc(15);
cout << "malloc done\n";
cout << "Here is the block list\n";
cout << "\n-------------BlockList start----------------\n";
cout << heaper << endl; n\n";
```

```
   cout << "Next free the most recently allocated pointer\n";
   heaper.free(p1);
   cout << "Here is the block list\n";
   cout << "\n-----------BlockList start-------------\n";
   cout << heaper << endl;
   cout << "------------BlockList end--------------\n\n";

   cout << "Next free the middle pointer\n";
   heaper.free(p2);
   cout << "Here is the block list\n";
   cout << "------------BlockList start--------------\n";
   cout << heaper << endl;
   cout << "\n-----------BlockList end-------------\n\n";

   return 0;
}
```

MemoryManager

M.free(p1);


prev
bsize
bptr
next
10
0
true
prev
bsize
bptr
next
20
false
prev
bsize
bptr
next
2033
true
firstBlock
memsize
2048
baseptr
M
Memory Pool