CS 61C (Clancy) **Lab assignment for week 3**
Spring 2002

Goals

This lab will give you practice working with a simulated version of the boundary tags method of storage allocation.

Reading

Section 8.7 in K&R. The code you will work with in this assignment differs in a number of respects from the K&R code:

- memory addresses are simulated by index values into the memory "pool";
- free blocks are not sorted by memory address;
- there's a "trailer" pointer at the end of each block, implementing a doubly linked list of free blocks and allowing a block to be freed without search;
- the size of a block precedes the link to the next free block;
- when a too-big block is found, the first part of it is returned to the user rather than the tail end;
- there is no interface to the operating system to get more memory when the allocated memory pool runs out.

Working with partners

Complete this lab in partnership with someone in your section with whom you have not yet worked. Make sure that you and your partner both understand all aspects of your solutions. Where you split work among partnership members, expect your lab t.a. or lab assistant to ask you to explain the work of your partner.

Running the program

The boundary tags code is found in ~cs61c/labs/lab3; a listing appears on the following pages. To compile it (after making the changes in exercise 1), type the command

```
gcc -Wall -g stgmain.c stgalloc.c
```

An example run for the completed program appears below, with user input in boldface and annotations following //. "Memory" addresses are simulated by integers in the range 1 to POOLSIZE (20 in the provided code). (0 is not used as a memory address so that it can represent a failed allocation.) The command "a" means "allocate"; its argument is the size in "bytes" of the requested memory segment. The command "f" means "free"; its argument is the "address" of the previously allocated block that is now to be freed.

```
18 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 18
Free blocks:                // The pool of allocatable memory consists of 20 locations.
        Header at 1; size = 18; prev at 0; next at 0

Command? a 2
-2 0 0 -2 14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 14
Free blocks:      // There is now an allocated block of size 2 and a free block of size 14.
        Header at 5; size = 14; prev at 0; next at 0
"address" of allocated block = 2

Command? a 14
-2 -2 -2 -2 -14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -14
Free blocks:       // The extra –2's simulate a user writing into the allocated memory.
"address" of allocated block = 6

Command? f 2
2 0 0 2 -14 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -14
Free blocks:
        Header at 1; size = 2; prev at 0; next at 0

Command? f 6
18 0 0 2 14 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 18
Free blocks:       // All allocated storage has been freed, though not reinitialized to 0.
        Header at 1; size = 18; prev at 0; next at 0

Command? a 4
-4 0 0 2 14 -4 12 0 0 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 12
Free blocks:
        Header at 7; size = 12; prev at 0; next at 0
"address" of allocated block = 2

Command? a 10
-4 -2 -2 -2 -2 -4 -12 0 0 -6 -6 -6 -6 -6 -6 -6 -6 -6 -6 -12
Free blocks:                    // 12 is allocated to avoid too small a leftover block
"address" of allocated block = 8

Command? f 8
-4 -2 -2 -2 -2 -4 12 0 0 -8 -8 -8 -8 -8 -8 -8 -8 -6 -6 12
Free blocks:
        Header at 7; size = 12; prev at 0; next at 0
```

```
Command? f 2
18 0 0 -2 -2 -4 12 0 0 -8 -8 -8 -8 -8 -8 -8 -8 -6 -6 18
Free blocks:
        Header at 1; size = 18; prev at 0; next at 0

Command? a 2
-2 0 0 -2 14 0 0 0 0 -8 -8 -8 -8 -8 -8 -8 -8 -6 -6 14
Free blocks:
        Header at 5; size = 14; prev at 0; next at 0
"address" of allocated block = 2

Command? a 3
-2 -2 -2 -2 -3 0 0 0 -3 9 0 0 -8 -8 -8 -8 -8 -6 -6 9
Free blocks:
        Header at 10; size = 9; prev at 0; next at 0
"address" of allocated block = 6

Command? a 10
Can't allocate!

Command? a 4
-2 -2 -2 -2 -3 -6 -6 -6 -3 -4 0 0 -8 -8 -4 3 0 0 -6 3
Free blocks:
        Header at 16; size = 3; prev at 0; next at 0
"address" of allocated block = 11

Command? a 1
-2 -2 -2 -2 -3 -6 -6 -6 -3 -4 -11 -11 -11 -11 -4 -3 0 0 -6 -3
Free blocks:                    // 3 is allocated to avoid too small a leftover block
"address" of allocated block = 17

Command? f 6
-2 -2 -2 -2 3 0 0 -6 3 -4 -11 -11 -11 -11 -4 -3 -17 0 -6 -3
Free blocks:
        Header at 5; size = 3; prev at 0; next at 0

Command? f 17
-2 -2 -2 -2 3 16 0 -6 3 -4 -11 -11 -11 -11 -4 3 0 5 -6 3
Free blocks:
        Header at 16; size = 3; prev at 0; next at 5
        Header at 5; size = 3; prev at 16; next at 0

Command? f 11
-2 -2 -2 -2 14 0 0 -6 3 4 -11 -11 -11 -11 -4 3 0 5 -6 14
Free blocks:
        Header at 5; size = 14; prev at 0; next at 0

Command? f 2
18 0 0 -2 14 0 0 -6 3 4 -11 -11 -11 -11 -4 3 0 5 -6 18
Free blocks:
        Header at 1; size = 18; prev at 0; next at 0
```

Exercise 1 (1 checkoff point)

The allocate and deallocate functions each require that you supply code that computes the size of a newly created block. Complete and test these functions. A diagram may be useful to convince your lab t.a. that the code you supply is correct.

Exercise 2 (1 checkoff point)

Supply a sequence of "a" and "f" commands that creates the following memory layout.

```
-5 -2 -2 -2 -2 -2 -5 3 0 17 -9 3 -2 -14 -14 -2 2 8 0 2
```

Exercise 3 (1 checkoff point)

Modify the allocate function to apply the "best fit" strategy for free block selection rather than the "first fit" strategy. Test your code.

Then find a sequence of allocate and free requests for which the first-fit strategy can successfully satisfy all the requests but the best-fit strategy cannot, and find another sequence of allocate and free requests for which the best-fit strategy can successfully satisfy all the requests but the first-fit strategy cannot. Display both sequences to your t.a. for checkoff.

Exercise 4 (1 checkoff point)

Add an entry to your lab.summary file that describes what you learned about the boundary tags method of storage allocation from this lab—in particular, what misconceptions you had that the lab activities cleared up—along with aspects of this topics that you are still unsure about. Your lab t.a. or lab assistant will want to see both your file and your partner's to award you this checkoff point.

stgmain.c

```
#include <stdio.h>
#include "stgalloc.h"

extern Block* memory;  /* the memory pool */

int main () {
   char cmd;
   int arg;
   int ptr;
   int k;

   initMemory ( );
   while (1) {
      /* Read commands of the form "a #" (in order to allocate the given number of units),
       * or "f ptr" (in order to free the space pointed to by the given "pointer".
       * A "pointer" here is either 0 (null) or a pool array index.
       */
      printf ("Command? ");
      scanf ("%1s%d", &cmd, &arg);
      switch (cmd) {
      case 'a':
         ptr = allocate (arg);
         if (ptr == 0) {
            printf ("Can't allocate!\n");
         } else {
            /* Write a bunch of identifying stuff to the memory just acquired. */
            for (k=ptr; k<ptr+arg; k++) {
               memory[k] = -ptr;
            }
            printf ("\"address\" of allocated block = %d\n", ptr);
         }
         break;
      case 'f':
         deallocate (arg);                        /* "free" is already used in C */
         break;
      case 'q':
         return 0;
      default:
         printf ("ASCII %x is not a legal command.\n", cmd);
      }
   }
}
```

stgalloc.h

```
typedef int Block;                  /* Block is an index into memory. */

void initMemory ( );
int allocate (int);
void deallocate (Block);            /* "free" is already used in C */
```

stgalloc.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stgalloc.h"

typedef int boolean;
#define TRUE 1
#define FALSE 0
boolean DEBUGGING = TRUE;

Block* memory;                      /* the memory pool */

Block firstFreeBlock;

int size (Block b);                 /* size of the given block */
void setSize (Block b, int newSize);

Block prev (Block b);               /* free block preceding the given one */
void setPrev (Block b, Block newPrev);
Block next (Block b);               /* free block following the given one */
void setNext (Block b, Block newNext);

int trailer (Block b);              /* trailer info for the given block */
void setTrailer (Block b, int value);
Block trailerLoc (Block b);         /* index of trailer */

Block leftBlockLoc (Block b);       /* index of block adjacent on the left */
Block rightBlockLoc (Block b);      /* index of block adjacent on the right */
boolean leftBlockIsFree (Block b);
boolean rightBlockIsFree (Block b);

void unlink (Block b);              /* unlink a block from the free list */

void debugPrint ( );
void printFreeBlockInfo ( );

/*
    Header information for each block consists of three pieces of info:
        int bSize, < 0 if block is allocated and > 0 if it's free;
        Block prev, index of the header info of the previous free block;
        Block next, index of the header info of the next free block.
    prev and next are only used for free blocks; otherwise they're
      fair game for use by the user.
    Trailer information for each block consists of one piece of info:
        int bSize, < 0 if block is allocated and > 0 if it's free.

    A value of type Block is the index into memory of the start of a legal block of storage.
    For a Block value b, memory[b] is the size of the block; for a Block value b representing
    a free block, memory[b+1] and memory[b+2] are the prev and next links, respectively.

    An implicit REQUIRES condition of everything in this program except initMemory is that
    memory is correctly set up as just described.
*/

int POOLSIZE = 20;                  /* size of the memory pool */
int HEADERSIZE = 3;
int TRAILERSIZE = 1;
```

```
/*
   REQUIRES: b represents a legal block of storage.
   EFFECTS: Return the index of the trailer for the given block.
*/
Block trailerLoc (Block b) {
   return b+abs(size(b))+1;
}

/*
   REQUIRES: b represents a legal block of storage.
   EFFECTS: Return the size of the given block.
*/
int size (Block b) {
   return memory[b];
}

/*
   REQUIRES: b represents a legal block of storage.
      newSize > 1, newSize ≤ the size of the largest possible block.
   EFFECTS: Set the size of the given block to the given value.
*/
void setSize (Block b, int newSize) {
   memory[b] = newSize;
}

/*
   REQUIRES: b represents a legal free block of storage.
   EFFECTS: Return the index of the free block preceding this one.
*/
Block prev (Block b) {
   return memory[b+1];
}

/*
   REQUIRES: b and newPrev represent legal free blocks of storage.
   EFFECTS: Point b's prev at newPrev.
*/
void setPrev (Block b, Block newPrev) {
   memory[b+1] = newPrev;
}

/*
   REQUIRES: b represents a legal free block of storage.
   EFFECTS: Return the index of the free block following this one.
*/
Block next (Block b) {
   return memory[b+2];
}

/*
   REQUIRES: b and newNext represent legal free blocks of storage.
   EFFECTS: Point b's next at newNext.
*/
void setNext (Block b, Block newNext) {
   memory[b+2] = newNext;
}
```

```
/*
   REQUIRES: b represents a legal block of storage.
   EFFECTS: Return the trailer information (i.e. the size) of the given block.
*/
int trailer (Block b) {
   return memory[trailerLoc(b)];
}

/*
   REQUIRES: b represents a legal block of storage;
      value > 1, value ≤ the size of the largest possible block.
   EFFECTS: Set the trailer information of the given block to the given value.
*/
void setTrailer (Block b, int value) {
   memory[trailerLoc(b)] = value;
}

/*
   REQUIRES: b represents a legal block of storage.
   EFFECTS: Return the index of the block adjacent on the left.
*/
Block leftBlockLoc (Block b) {
   if (b <= 1) {
      return 0;
   } else {
      return b - memory[b-1] - 2;
   }
}

/*
   REQUIRES: b represents a legal block of storage.
   EFFECTS: Return true if the block adjacent on the left is free.
*/
boolean leftBlockIsFree (Block b) {
   if (b <= 1) {
      return FALSE;
   } else {
      return memory[b-1] > 0;
   }
}

/*
   REQUIRES: b represents a legal block of storage.
   EFFECTS: Return the index of the block adjacent on the right.
*/
Block rightBlockLoc (Block b) {
   if (trailerLoc(b) >= POOLSIZE) {
      return 0;
   } else {
      return trailerLoc(b)+1;
   }
}
```

```
/*
   REQUIRES: b represents a legal block of storage.
   EFFECTS: Return true if the block adjacent on the right is free.
*/
boolean rightBlockIsFree (Block b) {
   if (trailerLoc(b) >= POOLSIZE) {
      return FALSE;
   } else {
      return memory[trailerLoc(b)+1] > 0;
   }
}

/*
   EFFECTS: Initialize the pool of allocatable memory as a single free block.
*/
void initMemory ( ) {
   memory = (Block *) malloc (sizeof(int) * (POOLSIZE+1));
   /* memory is treated as an array indexed starting at 1. */
   firstFreeBlock = 1;
   /* Usable memory in a block is all but one unit at each end. */
   setSize (1, POOLSIZE-2);
   setPrev (1, 0);                    /* no previous block */
   setNext (1, 0);                    /* no next block */
   setTrailer (1, POOLSIZE-2);
   debugPrint ();
}

/*
   REQUIRES: b represents a legal free block of storage.
   EFFECTS: Unlink the block starting at the given index from the free list.
*/
void unlink (Block b) {
   if (prev(b) != 0) {
      setNext (prev(b), next(b));
   } else {
      firstFreeBlock = next(b);
   }
   if (next(b) != 0) {
      setPrev (next(b), prev(b));
   }
}

/*
   EFFECTS: Print memory.
*/
void debugPrint ( ) {
   int k;
   if (DEBUGGING) {
      for (k=1; k<=POOLSIZE; k++) {
         printf ("%d ", memory[k]);
      }
      printf ("\n");
      printFreeBlockInfo ();
   }
}
```

```
/*
   EFFECTS:  Print information about the free blocks.
*/
void printFreeBlockInfo () {
   Block b;
   printf ("Free blocks:\n");
   for (b=firstFreeBlock; b; b=next(b)) {
      printf ("\tHeader at %d; size = %d; prev at %d; next at %d\n",
         b, size(b), prev(b), next(b));
      if (size(b) <= 0) {
         printf ("*** Block size is invalid!\n");
      }
      if (size(b) != trailer(b)) {
         printf ("*** Block sizes are inconsistent!\n");
      }
      if (prev(b)<0 || prev(b)>POOLSIZE) {
         printf ("*** Invalid prev!\n");
      }
      if (next(b)<0 || next(b)>POOLSIZE) {
         printf ("*** Invalid next!\n");
      }
   }
}
/*
   EFFECTS:  We have a request for n units.
      Return a pointer to n free units of storage from the pool—the pointer value, minus 1,
      represents a legal allocated block—choosing it using the first-fit strategy.
*/
int allocate (int n) {
   Block b;
   int oldBlockSize;                                  /* size of the block we choose */
   int allocBlockSize;                            /* size of the portion of it we allocate */
   if (n<=0) {                                  /* Make sure this is a reasonable request. */
      return 0;
   }
   if (n==1) {                                   /* A block of size 1 is too small to handle. */
      n++;
   }
   /* Search for a free block at least as large as desired. */
   for (b=firstFreeBlock; b && size(b)<n; b=next(b)) {
   }
   if (b == 0) {                                                    /* couldn't find one */
      return 0;
   }
```

```
   /*
      We have a free block. First find out exactly how much of the free block is to be used.
      If the requested size doesn't leave any space in the free block for another free block's header
      and trailer, return the whole block.
   */
   oldBlockSize = size(b);
   if (n <= size(b)-HEADERSIZE-TRAILERSIZE) {
      allocBlockSize = n;
   } else {
      allocBlockSize = oldBlockSize;
   }
   if (oldBlockSize == allocBlockSize) {
      /* The entire free block is used. */
      /* Merely detach it from the free list and return. */
      unlink (b);

      /* Update the free block's size too (to show that it's allocated). */
      setSize (b, -allocBlockSize);
      setTrailer (b, -allocBlockSize);
   } else {
      /* A new, smaller free block is to be created. Insert it into the free list. */
      Block newBlockLoc;

      setSize (b, -allocBlockSize);
      setTrailer (b, -allocBlockSize);
      newBlockLoc = rightBlockLoc (b);
      setSize (newBlockLoc, _____________________________ );
      setTrailer (newBlockLoc, _____________________________ );

      /* Link new block into the free list. */
      setPrev (newBlockLoc, prev(b));
      if (prev(b) != 0) {
         setNext (prev(b), newBlockLoc);
      } else {
         firstFreeBlock = newBlockLoc;
      }
      setNext (newBlockLoc, next(b));
      if (next(b) != 0) {
         setPrev (next(b), newBlockLoc);
      }
   }
   debugPrint ();
   return b+1;
}
```

```
/*
   REQUIRES: ptr represents a legal allocated block.
   EFFECTS:  We have a request to free a block.
      The argument points to the array cell immediately after the cell that starts the header.
*/
void deallocate (Block ptr) {
   Block b = ptr - 1;
   Block blockSize;
   boolean blockIsEnlarged = FALSE;
   setSize (b, abs(size(b)));       /* Indicate that the block is now free. */
   /* See if previous adjacent block is free, and combine if so. */
   if (leftBlockIsFree (b)) {
      blockSize = _____________________________________ ;
      b = leftBlockLoc(b);
      setSize (b, blockSize);
      /* See if next adjacent block is free, and combine if so. */
      if (rightBlockIsFree (b)) {
         int adjBlock = rightBlockLoc(b);
         blockSize = _____________________________________ ;
         setSize (b, blockSize);
         unlink (adjBlock);        /* Unlink the old block. */
      }
      blockIsEnlarged = TRUE;
   } else if (rightBlockIsFree (b)) {
      /* See if next adjacent block is free, and combine if so. */
      Block adjBlock = rightBlockLoc(b);
      blockSize = _____________________________________ ;
      setSize (b, blockSize);
      setPrev (b, prev(adjBlock));
      if (prev(b) != 0) {
         setNext (prev(b), b);
      } else {
         firstFreeBlock = b;
      }
      setNext (b, next(adjBlock));
      if (next(b) != 0) {
         setPrev (next(b), b);
      }
      blockIsEnlarged = TRUE;
   }
   setTrailer (b, size(b));
   /* If the new free block isn't already in the free list, add it to the start. */
   if (blockIsEnlarged) {
      debugPrint (); return;
   }
   if (firstFreeBlock != 0) {
      setPrev (firstFreeBlock, b);
   }
   setNext (b, firstFreeBlock);
   setPrev (b, 0);
   firstFreeBlock = b;
   debugPrint ();
}
```