Paris Diderot University

Engineering School Denis Diderot

Dynamic Memory Allocation in C

second year internship report

Author:
Mamadou Yéro Baldé

Supervisor:
Mihaela Sighireanu

February 25, 2019

Les allocateurs de memoire dynamique gèrent le tas (*heap*) d'un program. Ils sont indispensables à tout langage de programmation moderne. En C, l'allocateur de memoire est inclu dans la bibliothèque standard; en Java, il fait partie de l'environnement d'exécution (*runtime*). Le code de ces allocateurs est assez petit, mais il est notoirement difficile à mettre au point car il doit utiliser des primitives de programmation complexes: arithmetique des pointeurs, operations bit à bit, appels système. De plus, les structures de données que ce code utilise sont assez complexes: liste-tas (champs suivant obtenu avec l'arithmetique des pointeurs), listes doublement chainées, listes circulaires, arbres binaires de recherche, tables de hachage.

Pour mettre au point ces programmes, des techniques de verification formelles ont été developpées par l'équipe "Modélisation et Vérification" de l'IRIF, où mon stage a eu lieu. Ces techniques sont basées sur des logiques de programmes. Mon travail a été de fournir des programmes de test pour ces techniques. J'ai donc mis au point quatre allocateurs en utilisant des techniques de test unitaire et en s'appuyant sur la bibliotheque CUNIT.

Contents

1 Introduction

This report presents the work I have completed during my summer internship within the IRIF research institute at Paris Diderot University, under the supervision of Mihaela Sighireanu. The work performed concerns the study of *dynamic memory allocators* and the coding of some of them.

1.1 Presentation of IRIF

IRIF (abbreviation for the french translation of *Research Institute on the Foundations of Informatics*) is a research unit co-founded by the CNRS and the University Paris-Diderot, as UMR 8243. IRIF hosts two INRIA project-teams. IRIF is also member of the Foundation of Mathematical Sciences of Paris (FSMP) and of the Federation of Research in Mathematics of Paris[1].

The scientific objectives of IRIF are at the core of computer science and, in particular, they focus on the conception, analysis, proof, and verification of algorithms, programs, and programming languages. They are built upon fundamental research activities developed at IRIF on combinatorics, graphs, logics, automata, type, semantics, and algebras.

IRIF hosts currently (in January 2018) a hundred permanent members, including roughly 54 faculty members, 30 CNRS researchers, 6 INRIA researchers and 5 administrative and technical staffs. The total number of IRIF members, including PhD students, post-docs, and long-term visitors amounts for about two hundred.

IRIF is structured in nine thematic teams, grouped into three research poles. I've done my internship the the "Modeling and verification" team of the pole "Automata, structures and verification". The team studies and develops methods and tools for the software verification.

[1]See `www.irif.fr`

1.2 Motivation

Dynamic memory allocation (DMA for short) has been a fundamental part of most computer systems since roughly 1960, see for example Knuth's book [?], since it is an important part of the mechanisms allowing to manage program's memory. DMA are in charge of the dynamic memory of a program, also called the heap, but they do not belong in general to the low level operating system primitives. Indeed, the performances in time and memory consumption of the DMA depend on their usage. Therefore, there are different DMA implementations which are specialized for low level programming languages (like C/C++), high level programming languages (like Java) or specific usages (for real-time systems, for embedded systems, etc.).

The algorithms for such DMA and their performances are well studied, see for example [?, ?]. Their correctness is difficult to establish because the DMA code generally uses complex programming primitives: pointer arithmetics, bitwise operations, system calls (for the expansion of process memory). Some work exists on formal verification of such code, i.e., proving the correctness of DMA code using mathematics. These techniques are based on (i) logics specifying programs' configurations, also called program or Hoare's logics, and (ii) tools to manipulate proofs in these logics, e.g., theorem provers or static analysis.

The research team in which I've did my internship developed such formal verification tools are need to test them on DMA code. My work was to provide such examples of DMA code. The examples may contain flaws or may pass a test suite checking some elementary properties of DMA. I also had to build such a test suite for the code I've developed.

1.3 Overview

This document is organized as follows. Section 2 shortly introduces the organization of program's memory and the operating system's primitives allowing to manipulate this memory. These primitives are used by the DMA code, whose interface and properties are presented in Section 3. The internals and the main algorithms used by DMA are presented in Section 5. Section **??** presents the allocators I've fixed or coded during my internship. The test suite used for these allocators is presented in Section 6. I conclude this work in Section **??**. The code that I've developed or fixed is given in the appendixes.

2 Heap and System Calls

In the order to understand dynamic memory allocation, we need to understand how the memory of a process executing a program is handled in most operating systems. We will keep an abstract point of view for that part, since many details are operating system and hardware dependent.

2.1 The Process's Memory

Every process has its own virtual address space which is dynamically translated into physical memory address space by the memory management unit (MMU) and the kernel. This memory space may be seen as a contiguous array of bites, like in Figure 1. It is divided in several parts, mainly:

- the text part containing the code executed,
- the data part space for constant and global variables,
- the stack where local variables of functions called during the process execution are stored, and
- the program's data created during the execution, called the *heap*.

The DMA manages the heap. The heap is a continuous (in terms of virtual addresses) space of memory blocks with three bounds (see Figure 1): a starting point, a maximum limit (managed through `sys/resource.h`'s functions `getrlimit()` and `setrlimit()`) and an end point called the break. The break marks the end of the used memory space, that is, the part of the heap that is managed by the dynamic allocation [?].

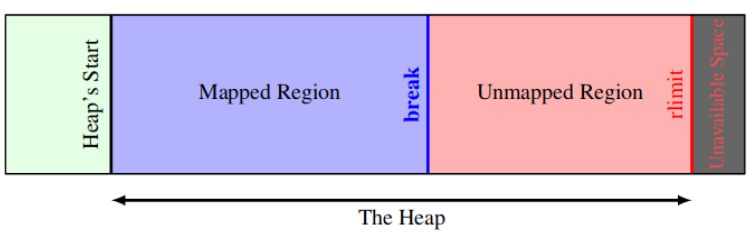

Figure 1: Memory organization

2.2 System Calls

To manage the heap, the DMA calls low level primitives of the operating system in order to obtain the starting point of the heap and the break point position; it also needs to be able to move the break point. For the Unix systems, these primitives are `brk()` and `sbrk()`, whose signature is:

```c
int brk(const void *addr);
void* sbrk(intptr_t incr);
```

As specified by the Unix manual, `brk()` and `sbrk()` change the location of the break point. Increasing the break point has the effect of allocating heap memory to the process; decreasing the break deallocates memory.

`brk()` sets the break point to the value specified by `addr`, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum more limit.

`sbrk()` increments the break point by `incr` bytes, with the same limitations like above. Calling `sbrk()` with an increment of 0 can be used to find the current location of the program break.

On success, `brk()` returns zero. On error, `-1` is returned. On success, `sbrk()` returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, `(void *) -1` is returned.

2.3 Heap Initialization

Using these system calls, the DMA starts from a initial size of the heap and moves the break depending on the client's needs. The initial configuration of the heap region may be fixed to some `size` using the following code, where the global variables `hit` and `hli` denote the start respectively the limit (first after the last) address of the memory region managed:

```c
void minit(int size)
{ hst=sbrk(size); hli=sbrk(0); }
```

Figure 2 illustrates the heap region obtained with the above code. In this region, the DMA manages its own data and the data allocated for the process.

3 Dynamic Memory Allocation

We focus on this work on the DMA for the C language.

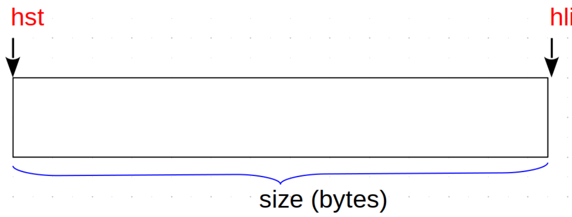

Figure 2: Initial configuration of the heap

3.1 User interface

Usually, programs use the dynamic memory allocation to add a node to a data structure whose size is not known. In object oriented languages, dynamic memory allocation is used to get the memory for a new object, for example the primitive `new` in Java. The program may ask for memory of different sizes at different execution points.

Memory allocated may be returned whenever it is no longer needed. Memory can be returned in any order without any relation to the order in which it was allocated.

Two functions make it possible to reserve and release dynamically an area of memory: `malloc()` for the reservation and `free()` for the liberation of previously allocated memory via `malloc()`.

3.1.1 Memory allocation

`malloc` is a C standard library function which is used to allocate a block of memory on the heap. The program accesses this block of memory via a pointer that `malloc` returns. The function signature is:

```
void* malloc(size_t size);
```

The only parameter to pass to malloc is `size`, the *number of bytes* to allocate. The returned value is the address of the first byte of the allocated memory area. If the allocation could not be realized (due to lack of free memory), the return value is the `NULL` constant.

3.1.2 Memory release

When memory returned by `malloc()` is no longer useful, it doesn't get freed on its own. In the order to release the space allocated before, the program should explicitly use another C standard library function: `free()`. The function signature is:

```
void free(void *ptr);
```

`free()` releases the memory space pointed by `ptr`, which shall be a pointer obtained on a previous call to `malloc()`. If `ptr` has already been released, the behavior of `free()` is undetermined and it is considered a programming flaw. If `ptr` is `NULL`, no attempt to release takes place.

3.1.3 Memory fragmentation

The heap region managed by the DMA (see Figure 2) is initially considered free and may be allocated entirely by a call to `malloc` of the size nearly equal to `size`. However, programs are usually asking for smaller blocks, which are reserved by splitting the heap region managed by the DMA in several *chunks*. I will present the precise organization of a chunk in the next section.

What is important here is to know that a chunk stores the block of memory required by the program and some internal data used by the DMA to manage the chunk.

With this splitting in chunks, the heap region may develop "holes" where previously allocated memory has been returned between blocks of memory still in use. A new request for memory might return a range of addresses out of one of the holes. But it might not use up all the hole, so further dynamic requests might be satisfied out of the original hole.

If too many small holes develop, memory is wasted because the total memory used by the holes may be large, but the holes cannot be used to satisfy dynamic requests. This situation is called *memory fragmentation* [?].

3.2 Properties of good allocators

An allocator must keep tracking chunks which are in use and free. The main goals of a good allocator are [?]:

- Maximizing compatibility: The implemented allocator must be compatible with ANSI/POSIX conventions.
- Maximizing portability: The allocator must cover as many systems as possible.
- Minimizing space: The allocator shouldn't waste space and track contiguous chunk to minimize fragmentation.

- Minimizing time: The time for allocation and release of memory should be as short as possible.
- Maximizing tunability: Optional features and behavior should be controllable by users either statically (via `#define` and the like) or dynamically (via control commands such as `mallopt`).
- Maximizing locality: Allocate chunks of memory that are typically used together near each other. This helps minimize page and cache misses during program execution.
- Maximizing error detection: Ensure that the use of the allocator is safe by checking the parameters received. Also, it does not seem possible for a general-purpose allocator to also serve as general-purpose memory error testing tool such as Purify. However, allocators should provide some means for detecting corruption due to overwriting memory, multiple frees, and so on.

4 Implementing an Allocator

Allocators are categorized by the mechanisms they use to keep track of free chunks and to coalesce neighboring free chunks. In what follows, I will explain step by step the design principles of a class of allocators that manages free chunks using a list.

4.1 Chunk Information

At the beginning of every chunk, the DMA stores extra-informations, called meta-data (see Figure 3), about the size of the chunk, a flag to mark free chunks (free or busy), the pointer to the next or previous free chunk. the pointer returned by `malloc` is the address in the chunk after this meta-data, that we call *chunk header* in the following.

There is a balance to find between the size of information stored in the meta-data and the memory consumption. More information may lead to faster algorithms, but also reduces the available memory. Let us present several ways of defining chunk header.

A chunk header storing the full information can be defining using the following C structure:

```c
typedef struct header
{
  size_t size;
```

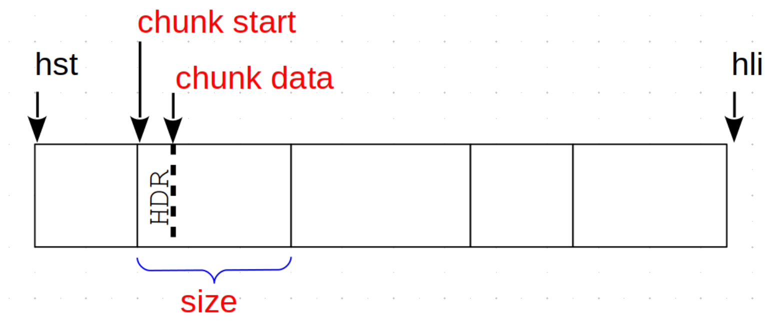

Figure 3: Chunk's internals

```
  bool isfree;
  struct header *fpr;
  struct header *fnx;
} header;
```

where the field **size** stores the size in bytes of the full chunk (including the header), **is free** is the flag storing the status of the chunk, and **fpr** resp. **fnx** are used to implement the list of free chunks (and they are valid only if **isfree** is true).

A more compact solution for the header may be obtained if the flag for the chunk's status is stored with the size information. This is possible if the DMA maintains only chunks of even size. Therefore, the least significative bit of the size is always zero and may be used to store the flag on the status of the chunk. Also, the list of free chunks may be only singly linked, therefore the **fpr** field may be eliminated. The following definition implements this more compact solution:

```
typedef struct header
{
  size_t size;
  struct header *fnx;
} header;
```

and it will be used in the next section in the code I've implemented. Notice that we can get the chunk size by doing a bitwise operation. The following chunk in the heap region is obtained by summing the address of the current chunk and the size of the chunk. The following macro-definitions get or set the information of the header defined above.

```
// methods to get header information
#define HDR_GET_SIZE(p)     (p->size & (~1))
```

```
#define HDR_GET_STATUS(p)   (p->size & 1)
#define HDR_GET_NEXT(p)     ((header*) p + HDR_GET_SIZE(p))
#define HDR_SET_SIZE(p,nh)  p->size = (((nh + 1) >> 1) << 1) &
    (~HDR_GET_STATUS(p))
#define HDR_SET_STATUS(p)   p->size = p->size | 1
#define HDR_UNSET_STATUS(p) p->size = p->size & (~1)
```

The most compact solution is the one storing only an integer (4 bytes) as header. This integer may store the size of the chunk and the flag (as explained above), or the next pointer and the flag (if the addresses of chunks are always even). The free list is not kept, which means that the DMA has to scan all the chunks to find the free ones and therefore is slower. The following definition implements this very compact solution:

```
typedef int header;
```

4.2 Allocation Algorithms

One of the important part of `malloc` is the way that free chunks are found and allocated. In what follows, we will discuss some policies and algorithms used to perform a dynamic allocation when the free chunks are stored in a list. Figure 4 illustrates a heap region managed by the DMA with free list.

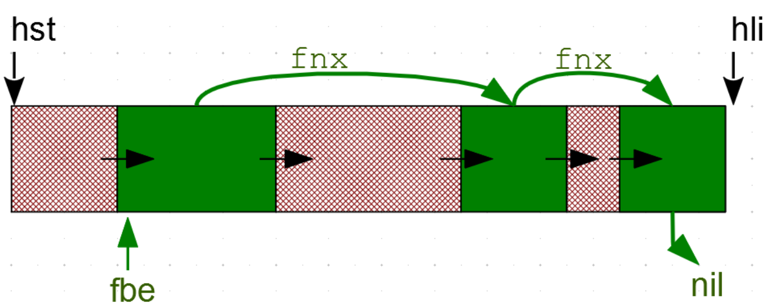

Figure 4: Heap with free list in green

4.2.1 First-fit method

The first-fit method allocates the first free chunk in the free list which has sufficient size to satisfy the request. If such a chunk does not exist, there are several choices. In lazy allocators, a coalescing of neighboring free chunks is done, and then the DMA tries again to find a suitable free chunk. Otherwise, if the break point does not reached the heap limit, the DMA has the choice to increase the size of the heap region managed using `sbrk()`, and then try the allocation again. If this fails, the allocation also fails.

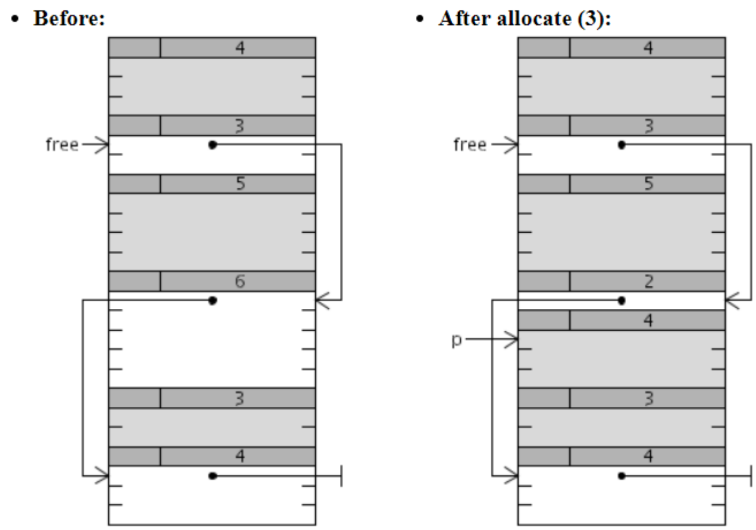

Figure 5: First fit execution for a request of size 3

Algorithm 1 Algorithm for malloc(n)

Require: $n \geq 0$

  $rsize \leftarrow n + size(header)$
  Scan free list for first chunk with size $\geq rsize$
  if chunk not found **then**
    Failure (time for coalescing!)
  else if free chunk size is $k \geq rsize + size(header) + 1$ **then**
    Split chunk into a free chunk and a busy chunk of size $rsize$
    Free chunk size $\leftarrow k - rsize$
    Busy chunk size $\leftarrow rsize$
    Return pointer to the data part of the Busy chunk
  else
    Unlink chunk from free list
    Return pointer to the data part of the chunk
  end if

The main advantage of this method is the fastest search. The disadvantage, studied in [**?**], is the localization of the in-use chunks at the start of the heap region.

4.2.2 Best-fit method

The best fit method allocates the free chunk which has the smallest sufficient size. If such a chunk does not exist, the best fit proceeds like in the failure case of the first-fit method: first it tries the coalescing of neighboring free chunks, then it increases the size of the heap region managed.

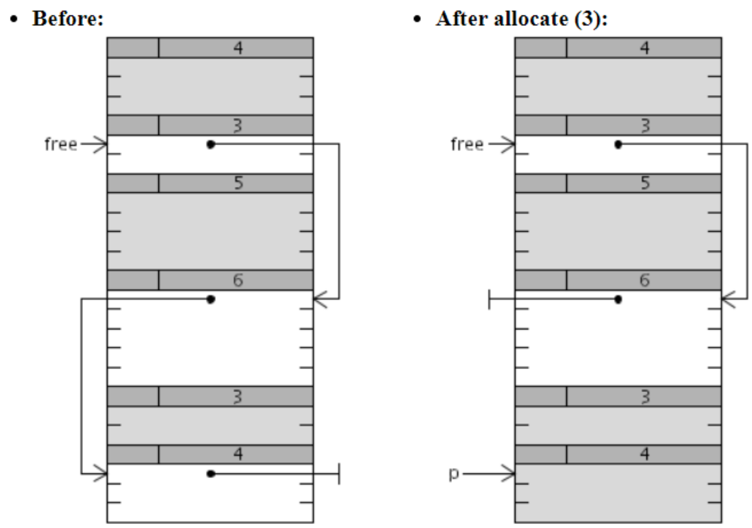

Figure 6: Best fit execution for a request of size 3

Although best fit minimizes the wastage space, it consumes a lot of time for searching the chunk with smallest size.

4.2.3 Coalescing of free chunks

Coalescing may be done either at the call of `free` (early coalescing allocators) or when `malloc` does not find a free chunk with enough size (lazy coalescing allocators).

In both cases, the allocator transform the free list by merging in one free chunk two chunks that are neighboring in the heap.

Algorithm 2 Algorithm for allocate (n)

Require: $n \geq 0$
Require: $n \geq 0$
  $rsize \leftarrow n + size(header)$
  Scan free list for for smallest block with size $\geq rsize$
  if chunk not found **then**
    Failure (time for coalescing!)
  else if free chunk size is $k \geq rsize + size(header) + 1$ **then**
    Split chunk into a free chunk and a busy chunk of size $rsize$
    Free chunk size $\leftarrow k - rsize$
    Busy chunk size $\leftarrow rsize$
    Return pointer to the data part of the busy chunk
  else
    Unlink chunk from free list
    Return pointer to the data part of the chunk
  end if

5 Allocators Implemented

5.1 Leslie Aldrige's allocator

I began first by trying to understand the allocator written by Leslie Aldrige [?] to get an idea about the implementation of allocators. This implementation is based on the first fit method, using a free list.

By testing the code, I've noticed several bugs in his implementation.For example, a first problem is ...

I've fixed this problem by ...

The final code is presented in Appendix A.

5.2 First version

This first implementation is based on the first fit method with early coalescing. The code is presented in Appendix B.

The header has a size field and a next free chunk field, as explained in Section 5.

```
typedef struct header
{
  size_t size;
  struct header *fnx;
} header;
```

The field `size` is equal to the quotient by the `header`'s size (constant `BLOCK_SIZE`) of the chunk's size. Therefore, at allocation, the size requested for the free chunk is obtained by rounding the parameter `size` of `malloc` to the smallest multiple of `header` size (constant `BLOCK_SIZE`) plus 1:

```
size_t rsize = (size / BLOCK_SIZE + 1) + 1;
rsize = ((rsize & 1) == 1) ? rsize + 1 : rsize;
```

The early coalescing is implemented in `free`. As soon as a chunk is released, checks are made to merge the contiguous free chunks.

5.3 Second version

This second implementation is based on the best fit method with early coalescing. The code is presented in Appendix C.

The code is very similar to the one of the first version except the scan loop where the best fit is computed.

5.4 Third version

This third implementation is based on the first fit method with a free list. The code is presented in Appendix D.

This version is the same as the first version except that the code of allocation and release has been refactored in several functions (search a chunk, coalescing). IT IS NOT CLEAR WHAT IS NEW HERE!!

6 Unit Tests for Allocators

The purpose of unit tests is to verify the expected behavior of a function with respect to a given specification. Here the lack of precise specification forces us to define requirements of these functions, what behaviors are expected or not.

One of the goals of this internship is to conduct a series of tests on the different implementations of the `malloc()` and `free()` functions. As a result, we will first carry out functional tests and subsequently cover tests (test of instruction and condition, decision or MCDC). The tests written are included in Appendix B.

6.1 Functional tests

The functional tests consist of a call for each function and the definition of the expected answers. The individual tests will be performed out of any context of use while the test suites will verify a succession of several orders.

6.1.1 Test of `malloc`

In this test, we try to do some kind of dynamic allocation to make sure that it works properly and then we check that the heap start and the break are not `NULL`.

6.1.2 Test of `free`

In this test, we try to do some dynamic allocation and we release the memory and then we try to see the state of the block to ensure the proper functioning of the function.

6.2 Coverage tests

For the coverage tests, we will first perform tests on the functions in individual ways, out of any context of use and those for each of the functions present. Then we will perform the tests in a context of use with a succession of function call.

6.2.1 Statement Coverage

In this test, we try to check all the main lines of the code. We first check that heap start and heap end are `NULL` before making an allocation then we check that heap start and heap end are not `NULL` and we release the allocated memory.

6.2.2 MC/DC cover

In software testing, the modified condition/decision coverage (MC/DC) is a code coverage criterion that requires all of the below during testing:

1. Each entry and exit point is invoked
2. Each decision takes every possible outcome
3. Each condition in a decision takes every possible outcome
4. Each condition in a decision is shown to independently affect the outcome of the decision.

In this test, I made several malloc and free to make sure that the merge blocks, the algorithm used and all the conditions were respected.

7 Conclusion

This internship was an opportunity to revise C programming and the notions learnt during the "Operating Systems" course. In addition, I've learnt how to write unit tests using CUNIT library. I also learnt how to write documents using LaTeX.

A Code for Leslie Aldrige's Allocator

Here is the revised version of Aldrige's code:

```c
#ifndef _LESLIE_H
#define _LESLIE_H

#include <stdio.h>

void cfree (char *ap);
char *cmalloc (int nbytes);
void i_alloc (void);

#endif
```

```c
#include <unistd.h>
#include "leslie.h"

#define MAX_MEM 1000
#define warm_boot(s) fprintf (stderr, "%s", s)

typedef struct hdr
{
  struct hdr *ptr;
  unsigned int size;
} HEADER;

//extern
void *_heapstart, *_heapend;

static short memleft;
static HEADER *frhd;

void
cfree (char *ap)
{
  HEADER *nxt, *prev, *f;
  f = (HEADER *) ap - 1;
  memleft += f->size;

  if (frhd > f)
    {
      nxt = frhd;
      frhd = f;
      prev = f + f->size;
```

```
      if (prev == nxt)
    {
      f->size += nxt->size;
      f->ptr = nxt->ptr;
    }
      else
    f->ptr = nxt;        // ERROR: nxt->ptr;
      return;
    }

  // NOT USED: nxt=frhd;
  for (nxt = frhd; nxt && nxt < f; prev = nxt, nxt = nxt->ptr)
    {
      if (nxt + nxt->size == f)
    {
      nxt->size += f->size;
      f = nxt + nxt->size;
      if (f == nxt->ptr)
        {
          nxt->size += f->size;
          nxt->ptr = f->ptr;
        }
      return;
    }
    }

  prev->ptr = f;
  prev = f + f->size;
  if (prev == nxt)
    {
      f->size += nxt->size;
      f->ptr = nxt;
      return;
    }
  // ADDED
  else
    {
      f->ptr = nxt;
    }
}

char *
cmalloc (int nbytes)
{
  HEADER *nxt, *prev;
  int nunits;
  nunits = (nbytes + sizeof (HEADER) - 1) / sizeof (HEADER) +
    1;

```

```
  for (prev = NULL, nxt = frhd; nxt; nxt = nxt->ptr)
    {
      if (nxt->size >= nunits)
    {
      if (nxt->size > nunits)
        {
          nxt->size -= nunits;
          nxt += nxt->size;
          nxt->size = nunits;
        }
      else
        {
          if (prev == NULL)
        frhd = nxt->ptr;
          else
        prev->ptr = nxt->ptr;
        }
      memleft -= nunits;
      return ((char *) (nxt + 1));
    }
    }
  warm_boot ("Allocation Failed!\n");
  return NULL;
}

void
i_alloc (void)
{
  // Code changed to call sbrk
  _heapstart = sbrk (0);

  if (_heapstart == (void *) -1)
    warm_boot ("1. sbrk Failed!\n");

  _heapend = sbrk (MAX_MEM * sizeof (HEADER));

  if (_heapend == (void *) -1)
    warm_boot ("2. sbrk Failed!\n");

  frhd = (HEADER *) _heapstart;

  frhd->ptr = NULL;
  frhd->size = ((char *) &_heapend - (char *) &_heapstart);
  memleft = frhd->size;
}
```

```c
#include <stdio.h>
#include "leslie.h"

int
main (void)
{
  char *p;
  i_alloc ();
  p = cmalloc (10 * sizeof (char));
  printf ("%p\n", p);
  p[0] = 'a';
  int i;
  for (i = 1; i < 10; i++)
    p[i] = 'a' + i;
  for (i = 0; i < 10; i++)
    printf ("%c ", p[i]);
  printf ("\n");

  cfree (p);
  printf ("%p\n", p);
  printf ("%c\n", p[3]);

  p = cmalloc (10 * sizeof (char));
  printf ("%p\n", p);

  return 0;
}
```

B Code for the first version

```c
#ifndef MALLOC_H_INCLUDED
#define MALLOC_H_INCLUDED

void *cmalloc (size_t size);
void cfree (void *p);
void cscan ();

#endif // MALLOC_H_INCLUDED
```

```c
#include <unistd.h>
#include <stdio.h>
```

```c
#include "malloc.h"

typedef struct header
{
  size_t size;           //memory block size
  struct header *nxt;
} header;

//head of the list
void *global_base = NULL;
void *global_end = NULL;


#define warm_boot(s) fprintf (stderr, "%s", s)
#define BLOCK_SIZE sizeof(header)

// methods to get header information
#define HDR_GET_SIZE(p)      (p->size & (~1))
#define HDR_GET_STATUS(p)    (p->size & 1)
#define HDR_GET_NEXT(p)      ((header*) p + HDR_GET_SIZE(p))
#define HDR_SET_SIZE(p,nh)  p->size = (((nh + 1) >> 1) << 1) & (~HDR_GET_STATUS(p))
#define HDR_SET_STATUS(p)   p->size = p->size | 1
#define HDR_UNSET_STATUS(p) p->size = p->size & (~1)

//inline int hdr_get_size (header * p){ return p->size & (~1);}


void
cscan ()
{
  header *ptr = global_base;
  for (; ((void *) ptr) != global_end; ptr = HDR_GET_NEXT (ptr))
    printf ("Chunk %p: size %ld, status %ld\n", ptr, HDR_GET_SIZE (ptr),
        HDR_GET_STATUS (ptr));
  printf ("-------------------------------------\n");
}

void *
cmalloc (size_t size)
{
  header *block;

  // size in header blocks
  size_t rsize = (size / BLOCK_SIZE + 1) + 1;
  rsize = ((rsize & 1) == 1) ? rsize + 1 : rsize;

```

```
  // First call
  if (!global_base)
    {
      block = sbrk (0);
      // sbrk failed
      if (sbrk (rsize * BLOCK_SIZE) == (void *) -1)
    return NULL;

      block->size = rsize;

      global_base = block;
      global_end = sbrk (0);

      return (void *) (block + 1);
    }

  //finding a chunk with the First Fit Algorithm
  for (block = global_base; block != (header *) global_end;
       block = HDR_GET_NEXT (block))
    {
      //free chunk has enough space
      if (HDR_GET_STATUS (block) && HDR_GET_SIZE (block) >=
   rsize)
    {
      break;
    }
    }



  // Failed to find free block
  if (block == (header *) global_end)
    {
      block = sbrk (0);
      // sbrk failed
      if (sbrk (rsize * BLOCK_SIZE) == (void *) -1)
    return NULL;

      block->size = rsize;
      if (global_base == NULL)
    global_base = block;
      global_end = sbrk (0);
    }
  else              // Found free block
    {
      if (HDR_GET_SIZE (block) == rsize)
    {
      block->size = rsize;
    }
```

```
      else
    {
      //splitting block
      header *rblock = block + rsize;

      rblock->size = (block->size - rsize) | 1;
      block->size = rsize & (~1);   // set status to 0

      /* HDR_SET_SIZE(rblock, HDR_GET_SIZE(block)-rsize);
         HDR_SET_STATUS(rblock);
         HDR_UNSET_STATUS(block); */
    }
    }

  return (void *) (block + 1);
}

void
cfree (void *p)
{
  header *block = (header *) p;
  block = block - 1;
  //pointer on current block and last block
  header *ptr, *last, *nxt;

  //looking for wether the block exists
  for (ptr = (header *) global_base; ptr <= (header *)
    global_end;
       ptr = HDR_GET_NEXT (ptr))
    {
      if (ptr == block)
    {
      //already free
      if (HDR_GET_STATUS (ptr))
        return;
      //coalescing previous and current blocks
      if (last && HDR_GET_STATUS (last))
        {
          last->size = (last->size & (~1)) + (ptr->size & (~1))
    ;
          ptr = last;
        }
      //coalescing next and current blocks
      if (ptr != (header *) global_end)
        {
          nxt = HDR_GET_NEXT (ptr);
          if (nxt && HDR_GET_STATUS (nxt))
        {
          ptr->size = ptr->size + (nxt->size & (~1));
```

```
        }
        }
      // set status to 1
      ptr->size = ptr->size | 1;

      //printf ("free of %p\n\n", p);

      return;
    }
      last = ptr;
    }

  warm_boot ("free failed!\n");
  return;
}
```

```
#include <stdio.h>
#include "CUnit/Basic.h"
#include "malloc.c"

int
init_suite_cm ()
{
  return 0;
}

int
clean_suite_cm ()
{
  return 0;
}

void
test_couverture_instruction ()
{
    CU_ASSERT_PTR_NULL (global_base);
    CU_ASSERT_PTR_NULL (global_end);
    void *v = cmalloc (10);
    CU_ASSERT_PTR_NOT_NULL (global_base);
    CU_ASSERT_PTR_NOT_NULL (global_end);
    void *v1 = cmalloc (10);
    cfree (v1);
    v1 = cmalloc (10);
    cfree (v);
    cfree (v1);
};
```

```c

void
test_couverture_mcdc ()
{
  int *a, *b, *c, *d, *e;
  a = cmalloc (3 * sizeof (int));
  b = cmalloc (7 * sizeof (int));
  c = cmalloc (2 * sizeof (int));
  d = cmalloc (2 * sizeof (int));

  cfree (b);
  cfree (a);
  cfree (d);
  cfree (c);

  e = cmalloc (2 * sizeof (int));
  a = cmalloc (200 * sizeof (int));

  cfree (a);

  a = cmalloc (20 * sizeof (int));

  cfree (a);
 cfree(e);
};

void
test_free ()
{
  // precondition fonctionnement normal
  void *v = cmalloc (10);
  //tests
  cfree (v);
  header *mcb;
  mcb = v - sizeof (header);
  CU_ASSERT_EQUAL (HDR_GET_STATUS (mcb), 1);
}



void
test_malloc ()
{
  int taille = 10;
  void *v = cmalloc (taille);
  CU_ASSERT_PTR_NOT_NULL (global_base);
  CU_ASSERT_PTR_NOT_NULL (global_end);
  cfree (v);
}
```

```c

int
main ()
{
  /* initialize the CUnit test registry */
  if (CUE_SUCCESS != CU_initialize_registry ())
    return CU_get_error ();

  /* add a suite to the registry
   * The 1st test suite corresponds to the global functions,
    with a cover
   */
  CU_pSuite pSuite =
    CU_add_suite ("test_suite_couverture", init_suite_cm,
   clean_suite_cm);
  if (NULL == pSuite)
    {
      CU_cleanup_registry ();
      return CU_get_error ();
    }

  /* add the tests global to the suite */
  if (NULL ==
      CU_add_test (pSuite, "test_couverture_instruction",
           test_couverture_instruction)
      || NULL == CU_add_test (pSuite, "test_couverture_mcdc",
                  test_couverture_mcdc))
    {
      CU_cleanup_registry ();
      return CU_get_error ();
    }
  printf ("\n");
  /* add a suite to the registry
   * The 2nd test suite corresponds to the individual functions
   */
  CU_pSuite pSuite2 =
    CU_add_suite ("individual test suite in FF malloc",
   init_suite_cm,
          clean_suite_cm);
  if (NULL == pSuite2)
    {
      CU_cleanup_registry ();
      return CU_get_error ();
    }
  /* add the tests malloc to the suite */
  if (NULL == CU_add_test (pSuite2, "test of malloc",
   test_malloc))
    {
      CU_cleanup_registry ();
```

```
      return CU_get_error ();
    }

  /* add the tests free to the suite */
  if (NULL == CU_add_test (pSuite2, "test of free", test_free))
    {
      CU_cleanup_registry ();
      return CU_get_error ();
    }



  /* Run all tests using the CUnit Basic interface */
  CU_basic_set_mode (CU_BRM_VERBOSE);
  CU_basic_run_tests ();
  printf ("\n");
  CU_basic_show_failures (CU_get_failure_list ());
  printf ("\n");

  /* Clean up registry and return */
  CU_cleanup_registry ();
  return CU_get_error ();
}
```

C Code for the second version

```
#ifndef MALLOC_H_INCLUDED
#define MALLOC_H_INCLUDED

void *cmalloc (size_t size);
void cfree (void *p);
void cscan ();

#endif // MALLOC_H_INCLUDED
```

```
#include <unistd.h>
#include <stdio.h>
#include "malloc.h"

typedef struct header
{
```

```c
    size_t size;            //memory block size
    struct header *nxt;
} header;

//head of the list
void *global_base = NULL;
void *global_end = NULL;


#define warm_boot(s) fprintf (stderr, "%s", s)
#define BLOCK_SIZE sizeof(header)

// methods to get header information
#define HDR_GET_SIZE(p)     (p->size & (~1))
#define HDR_GET_STATUS(p)   (p->size & 1)
#define HDR_GET_NEXT(p)     ((header*) p + HDR_GET_SIZE(p))
#define HDR_SET_SIZE(p,nh)  p->size = (((nh + 1) >> 1) << 1) & (~HDR_GET_STATUS(p))
#define HDR_SET_STATUS(p)   p->size = p->size | 1
#define HDR_UNSET_STATUS(p) p->size = p->size & (~1)

inline int
hdr_get_size (header * p)
{
    return p->size & (~1);
}


void
cscan ()
{
    header *ptr = global_base;
    for (; ((void *) ptr) != global_end; ptr = HDR_GET_NEXT (ptr))
        printf ("Chunk %p: size %ld, status %ld\n", ptr, HDR_GET_SIZE (ptr),
                HDR_GET_STATUS (ptr));
    printf ("-----------------------------------\n");
}

void *
cmalloc (size_t size)
{
    header *block, *best_block;

    // size in header blocks
    size_t rsize = (size / BLOCK_SIZE + 1) + 1;
    rsize = ((rsize & 1) == 1) ? rsize + 1 : rsize;

```

```
    // First call
    if (!global_base)
    {
        block = sbrk (0);
        // sbrk failed
        if (sbrk (rsize * BLOCK_SIZE) == (void *) -1)
            return NULL;

        block->size = rsize;

        global_base = block;
        global_end = sbrk (0);

        return (void *) (block + 1);
    }

    //finding a chunk with the Best Fit Algorithm
    for (block = global_base; block != (header *) global_end;
            block = HDR_GET_NEXT (block))
    {

        if (HDR_GET_STATUS (block) && HDR_GET_SIZE (block) >= rsize)
        {
            if (!best_block
                    || (block && HDR_GET_SIZE (block) < HDR_GET_SIZE (best_block)))
                best_block = block;
        }
    }

    if(best_block)
        block=best_block;

    // Failed to find free block
    if (block == (header *) global_end )
    {
        block = sbrk (0);
        // sbrk failed
        if (sbrk (rsize * BLOCK_SIZE) == (void *) -1)
            return NULL;

        block->size = rsize;
        if (global_base == NULL)
            global_base = block;
        global_end = sbrk (0);
    }
    else                    // Found free block
    {
```

```
        if (HDR_GET_SIZE (block) == rsize)
        {
            block->size = rsize;
        }
        else
        {
            //splitting block
            header *rblock = block + rsize;

            rblock->size = (block->size - rsize) | 1;
            block->size = rsize & (~1); // set status to 0

            /* HDR_SET_SIZE(rblock, HDR_GET_SIZE(block)-rsize);
               HDR_SET_STATUS(rblock);
               HDR_UNSET_STATUS(block); */
        }
    }

    return (void *) (block + 1);
}

void
cfree (void *p)
{
    header *block = (header *) p;
    block = block - 1;
    //pointer on current block and last block
    header *ptr, *last, *nxt;

    //looking for wether the block exists
    for (ptr = (header *) global_base; ptr < (header *) global_end;
            ptr = HDR_GET_NEXT (ptr))
    {
        if (ptr == block)
        {
            //already free
            if (HDR_GET_STATUS (ptr))
                return;
            //coalescing previous and current blocks
            if (last && HDR_GET_STATUS (last))
            {
                last->size = (last->size & (~1)) + (ptr->size & (~1));
                ptr = last;
            }
            //coalescing next and current blocks
            if (ptr != (header *) global_end)
            {
```

```
                nxt = HDR_GET_NEXT (ptr);
                if (nxt && HDR_GET_STATUS (nxt))
                {
                    ptr->size = ptr->size + (nxt->size & (~1));
                }
            }
            // set status to 1
            ptr->size = ptr->size | 1;

            //printf ("free of %p\n\n", p);

            return;
        }
        last = ptr;
    }

    warm_boot ("free failed!\n");
    return;
}
```

```
#include <stdio.h>
#include "CUnit/Basic.h"
#include "malloc.c"

int
init_suite_cm ()
{
    return 0;
}

int
clean_suite_cm ()
{
    return 0;
}

void
test_couverture_instruction ()
{
    CU_ASSERT_PTR_NULL (global_base);
    CU_ASSERT_PTR_NULL (global_end);
    void *v = cmalloc (10);
    CU_ASSERT_PTR_NOT_NULL (global_base);
    CU_ASSERT_PTR_NOT_NULL (global_end);
    void *v1 = cmalloc (10);
    cfree (v1);
```

```
    v1 = cmalloc (10);
    cfree (v);
    cfree (v1);
};

void
test_couverture_mcdc ()
{
  int *a, *b, *c, *d;

  a = cmalloc (30 * sizeof (int));
  b = cmalloc (7 * sizeof (int));
  c = cmalloc (2 * sizeof (int));
  d = cmalloc (20 * sizeof (int));

  cfree (b);
  cfree (a);
  cfree (d);

  b = cmalloc (7 * sizeof (int));
  printf("1 ");
  cfree (c);

  a = cmalloc (200 * sizeof (int));

  cfree (a);

  a = cmalloc (20 * sizeof (int));

  cfree (a);
};

void
test_free ()
{
    // precondition fonctionnement normal
    void *v = cmalloc (10);
    //tests
    cfree (v);
    header *mcb;
    mcb = v - sizeof (header);
    CU_ASSERT_EQUAL (HDR_GET_STATUS (mcb), 1);
}



void
test_malloc ()
{
```

```c
    int taille = 10;
    void *v = cmalloc (taille);
    CU_ASSERT_PTR_NOT_NULL (global_base);
    CU_ASSERT_PTR_NOT_NULL (global_end);
    cfree (v);
}

int
main ()
{
    /* initialize the CUnit test registry */
    if (CUE_SUCCESS != CU_initialize_registry ())
        return CU_get_error ();

    /* add a suite to the registry
     * The 1st test suite corresponds to the global functions, with a cover
     */
    CU_pSuite pSuite =
        CU_add_suite ("test_suite_couverture", init_suite_cm, clean_suite_cm);
    if (NULL == pSuite)
    {
        CU_cleanup_registry ();
        return CU_get_error ();
    }

    /* add the tests global to the suite */
    if (NULL ==
            CU_add_test (pSuite, "test_couverture_instruction",
                         test_couverture_instruction)
            || NULL == CU_add_test (pSuite, "test_couverture_mcdc",
                                    test_couverture_mcdc))
    {
        CU_cleanup_registry ();
        return CU_get_error ();
    }
    printf ("\n");
    /* add a suite to the registry
     * The 2nd test suite corresponds to the individual functions
     */
    CU_pSuite pSuite2 =
        CU_add_suite ("individual test suite in BF malloc", init_suite_cm, clean_suite_cm);
    if (NULL == pSuite2)
    {
        CU_cleanup_registry ();
```

```
        return CU_get_error ();
    }
    /* add the tests malloc to the suite */
    if (NULL == CU_add_test (pSuite2, "test of malloc",
    test_malloc))
    {
        CU_cleanup_registry ();
        return CU_get_error ();
    }

    /* add the tests free to the suite */
    if (NULL == CU_add_test (pSuite2, "test of free", test_free
    ))
    {
        CU_cleanup_registry ();
        return CU_get_error ();
    }



    /* Run all tests using the CUnit Basic interface */
    CU_basic_set_mode (CU_BRM_VERBOSE);
    CU_basic_run_tests ();
    printf ("\n");
    CU_basic_show_failures (CU_get_failure_list ());
    printf ("\n");

    /* Clean up registry and return */
    CU_cleanup_registry ();
    return CU_get_error ();
}
```

D Code for the third version

```
#ifndef MALLOC_H_INCLUDED
#define MALLOC_H_INCLUDED

void *cmalloc (size_t size);
void cfree (void *p);
void cscan ();
#endif // MALLOC_H_INCLUDED
```

```
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include "malloc.h"

typedef struct header {
  size_t       size;         /* memory block size */
  struct header  *nxt;
} header;

/* Global limits of the heap */
void* global_base = NULL;
void* global_end = NULL;

/* Head of the free list */
header* frhd = NULL;

#define warm_boot(s) fprintf (stderr, "%s", s)
#define BLOCK_SIZE sizeof(header)

/* Interface for the header's information */
#define HDR_GET_SIZE(p)     (p->size & (~1))
#define HDR_GET_STATUS(p)   (p->size & 1)
#define HDR_GET_NEXT(p)     ((header*)p + HDR_GET_SIZE(p))
#define HDR_SET_SIZE(p,nh)  p->size = (((nh + 1) >> 1) << 1) &
    (~HDR_GET_STATUS(p))
#define HDR_SET_STATUS(p)   p->size = p->size | 1
#define HDR_UNSET_STATUS(p) p->size = p->size & (~1)

/* Forward declarations of utilities */
void    cscan(char*);
void    defrag(header *);
header* search(size_t);

void
cfree(void *p)
{
  header* block = (header *) p;
  block = block - 1;
  /* pointer on current block and last block */
  header* ptr;

#ifdef VERBOSE_M
    printf("\n  - search address %p\n", block);
#endif

  /* looking for if the block exists */
  for (ptr = (header*) global_base; ptr < (header*) global_end;
       ptr = HDR_GET_NEXT(ptr)) {
```

```
#ifdef VERBOSE_M
    printf("\t- visit address %p\n", ptr);
#endif
    if (ptr == block) {
      /* already free: error */
      if (HDR_GET_STATUS(ptr)) {
        warm_boot("free failed!\n");
    return;
      }

      /* set status to free */
      HDR_SET_STATUS(ptr);
      if (frhd == NULL) {
    /* empty free list */
        frhd = ptr;
    frhd->nxt = NULL;
      } else if (frhd < ptr) {
        /* non empty free list */
        /* insert such that the free list is sorted by address
    */
    header *tmp = frhd;

#ifdef VERBOSE_M
        printf("    - insert in free list %p\n", frhd);
#endif
        assert(tmp != NULL && tmp < ptr);
    while (tmp->nxt != NULL && tmp->nxt < ptr) {
#ifdef VERBOSE_M
          printf("\t-visit free chunk %p\n", tmp->nxt);
#endif
      tmp = tmp->nxt;
    }

    ptr->nxt = tmp->nxt;
    tmp->nxt = ptr;
      } else {
        assert (frhd > ptr);
    block->nxt = frhd;
    frhd = block;
      }
      cscan("After free");
      return;
    }
  }
  warm_boot("free failed!\n");
}

void*
cmalloc(size_t size)
```

```
{
  header* block = NULL;
  /* compute the size in header blocks */
  size_t rsize = ((size / BLOCK_SIZE) + 1) + 1;
  /* make it even to have the last bit unused */
  rsize = ((rsize & 1) == 1) ? (rsize + 1) : rsize;

  if (global_base != NULL && frhd != NULL) {
    /* non empty free list */
    /* search a free chunk that fits the request, uses First
   Fit Algorithm */
    block = search(rsize);
    if (block == NULL) {
      /* no free chunk fits, defragmentate the free list */
      defrag(frhd);
      /* redo the search */
      block = search(rsize);
    }
    if (block)
      return (void *)(block + 1);
  }

  /* block is still null */
  assert (block == NULL);

  /* extend the memory */
  if (global_base == NULL) {
    global_base = sbrk(0);
    global_end = global_base ;
  }
  /* the block will be at the end of the current region */
  block = global_end;
  /* extend the memory by a system call */
  if (sbrk(rsize * BLOCK_SIZE) == (void *)-1)
    /* sbrk failed */
    return NULL;

  block->size = rsize;
  /* HDR_GET_STATUS(p) is occupied, i.e., 0 */
  /* update the limit of the memory region */
  global_end = sbrk(0);

  return (void *)(block + 1);
}

void
cscan(char* msg)
{
#ifdef VERBOSE_M
```

```
  header* ptr = global_base;
  printf("* %s:\n", msg);
  printf("  - [%p, %p)\n", global_base, global_end);
  for (; ((void *)ptr) != global_end; ptr = HDR_GET_NEXT(ptr))
    printf("Chunk %p: size %ld (=%ld B), status %ld\n", ptr,
   HDR_GET_SIZE(ptr),
        HDR_GET_SIZE(ptr) * BLOCK_SIZE,
        HDR_GET_STATUS(ptr));
  printf("------------------------------------------\n");
#endif
  return;
}

void
defrag(header* frhd)
{
  header* ptr = frhd;

  assert (ptr != NULL);
  while (ptr->nxt != NULL) {
    if (HDR_GET_NEXT(ptr) == ptr->nxt) {
      /* for two consecutive free blocks, do coalescing */
      /* increase size */
      ptr->size = HDR_GET_SIZE(ptr) + HDR_GET_SIZE(ptr->nxt);
      /* set again as free */
      HDR_SET_STATUS(ptr);
      /* update the next pointer */
      ptr->nxt = (ptr->nxt)->nxt;
    } else {
      ptr->nxt = (ptr->nxt)->nxt;
    }
  }
  cscan("After defrag");
}

header*
search(size_t rsize)
{
  header* block, *prev;

  for (prev = NULL, block = frhd; block != NULL;
       prev = block, block = block->nxt) {
    assert (HDR_GET_STATUS(block) == 1);
    if (HDR_GET_SIZE(block) >= rsize) {
      if (HDR_GET_SIZE(block) == rsize) {
    /* no need to split */
        block->size = rsize;
    /* remove block from free list */
        if (prev)
```

```
      prev->nxt = block->nxt;
    else
      frhd = block->nxt;
    return block;
      } else {
    /* split the block, reserve its end for this allocation */
        header* rblock = block + (HDR_GET_SIZE(block) - rsize);

    rblock->size = rsize;
        /* update the size of the free block */
    block->size = HDR_GET_SIZE(block) - rsize;
        /* let block in the free list */
        HDR_SET_STATUS(block);
        cscan("After split");
        return rblock;
      }
    }
  }
  return NULL;
}
```