Started on	Friday, 16 August 2019, 2:02 PM
State	Finished
Completed on	Wednesday, 28 August 2019, 11:57 PM
Time taken	12 days 9 hours
Marks	23.67/24.00
Grade	**9.86** out of 10.00 (**99**%)

Information

Lab 6: Dynamic Memory

Goals

This lab introduces the use of the dynamic memory allocation and de-allocation functions (`malloc` and `free`). It is the last lab this term.

A playlist of videos for this topic is here. It is recommended you watch all the videos in that list before starting the lab.

By the end of this lab you should:

- Understand the concept of the heap and be able to picture the effects of calls to `malloc` to allocate a memory block from the heap and `free` to return a block to the heap.
- Be able to write code that calls `malloc` and `free` in a simple application.
- Understand what memory leaks are and how they arise.
- Be aware of the dangers of heap corruption and how it can occur.
- Have some appreciation of when it is appropriate to use dynamic memory techniques rather than fixed-size arrays.

Read Chapter 17 of the text before attempting this lab.

The tyranny of fixed-size arrays

You should by now have realised that one of the most annoying aspects of C is that you have to pre-allocate fixed-size arrays for everything. It's true that in C99 you can allocate a variable-sized local array in the current stack frame, but this is local data that disappears when functions return which is not sufficient for many situations. For example:

- If you want to read a line of input from stdin into a local array, how big should that array be? No matter how much you choose, you'll be in trouble if stdin is being taken from a large file without any newline characters in it, either accidentally (say by inadvertently processing a binary file) or deliberately (say by a hacker attempting to break your code).
- If you want to write a function that returns the concatenation of two strings, the memory space must be reserved in the caller's space because the space allocated within the called function's own stack space is freed when the function returns.

Why can't C do what languages like Python or Java do, and allocate you as much memory as you actually need? Well, it can, with caveats, by using what is called *dynamic memory*. Dynamic memory is in fact what is used by languages like Python and Java to manage their "object stores".

A trivial example

Inspect the following program `randomNumbers.c`. The functions *malloc* and *free* are introduced in the video *9.2 Malloc and free*.

Hide `randomnumbers.c` code

```
/* A simple demonstration of the use of malloc and free to
 * construct an array of random integers of some given size.
 * Written for Lab 6, Dynamic Memory, by RJL, June 2013.
 */

#include <stdio.h>
#include <stdlib.h>

/* Return a pointer to a dynamically allocated array of doubles of
 * the given size n, filled with random values in [0,1) i.e. the range
 * of numbers greater than or equal to 0 and less than 1.
 */
double* randomArray(int n)
{
    double* numbers = malloc(n * sizeof(double));
    for (int i = 0; i < n; i++) {      // Fill with random numbers
        numbers[i] = (1.0 * rand()) / RAND_MAX;
        // Note numerator is converted to a double
        // to ensure floating point division is done.
    }
    return numbers;
}

int main(void)
{
    int n = 0;
    double* randomNums = NULL;
    printf("How many random doubles? ");
    scanf("%d", &n);
    randomNums = randomArray(n);
    for (int i = 0; i < n; i++) {
        printf("%.6f\n", randomNums[i]);
    }
    printf("\n");
    free(randomNums);
    return 0;
}
```

Note the following:

1. The statement `double* numbers = malloc(n * sizeof(double))` declares a pointer to a double and initialises it by calling the memory allocator function `malloc`, passing as a parameter the number of bytes of memory required. As a result, `numbers` points to the base address of an area of memory that is exactly big enough to hold `n` doubles.
2. `malloc` allocates memory from the area called the *heap*; see lab 4, figure 2.
3. If you forget to multiply `n` by the size of a double, i.e., you call `malloc(n)` you'll get a block of memory one eighth the size you actually need. This is probably going to get ugly!
4. The block of memory given to you by `malloc` is yours to play in for as long as you like, until you call `free`.
5. `free` returns the memory to the heap, from which it may then be reallocated on later `malloc` calls.
6. Note that the call to `free` does not affect the pointer that you pass as a parameter; it continues to point at what used to be your memory block but may now be in use for some other purpose (e.g. in use by `printf`). Do not attempt to use a pointer once you have called `free` on it.
7. There is another memory allocation function called `calloc` that takes two parameters: the number of items for which space is to be allocated and the size of each item. It calls `malloc` and also clears the block of memory so allocated. You may use this if you wish, though I will generally ignore its existence from now on, as `malloc` is the more fundamental function.
8. You might wonder: how big is the heap? In fact, it isn't a fixed size at all. There's an operating system call called `brk` - type `man brk` for more information - which lets a program expand the size of its global data area. You shouldn't ever need to call this yourself, but `malloc` uses it to grow the heap space as required. The maximum allowable size is operating system dependent, but you should realise that it is quite possible to bring the operating system to its knees by demanding more dynamic memory than is physically available.

Although dynamic memory can be very useful, it is very easy to misuse it, e.g. by running off the end of your allocated block. To help improve your chances of detecting such problems, you should add the option `-lmcheck` to the end of the *build* command in `geany`. This links your program with slightly more user-friendly versions of `malloc` and `free` which carry out a few validity checks on your memory blocks. Do that now, then build and run `randomNumbers` a few times.

Question **1**

Correct

Mark 1.00 out of 1.00

Select the true statements from the following.

Select one or more:

- [x] A call to `malloc(40)` returns a pointer to an area of memory that can hold 40 char values. ✔
- [ ] A call to `malloc(40)` returns a pointer to an area of memory that can hold 40 int values.
- [ ] A call to `malloc(40)` returns a pointer to an area of memory that can hold 40 double values.
- [ ] A call to `malloc(40)` returns a pointer to an area of memory that can hold 40 values of whatever type is being declared.
- [ ] After a call to `free(p)`, the pointer `p` will be NULL.
- [x] In a call to `free(p)`, `p` should be a pointer that was given a value by an earlier call to `malloc` or `calloc`. ✔
- [x] A call to `free(p)` does not alter the value of `p`. ✔

Correct

Marks for this submission: 1.00/1.00.

Question **2**

Incorrect

Mark 0.67 out of 1.00

If you change the declaration and initialisation of the pointer `numbers` to

```
double* numbers = malloc(n);
```

what happens when you run the program, entering *n* = 1000? [It is assumed that you have added the option `-lmcheck` to the end of the build command.]

Select one:

- [ ] The program fails to compile.
- [ ] The program aborts with an error message regarding inconsistent array element sizing when you call *malloc.*
- [x] In this case the program performs exactly as before, but you're living dangerously as further calls to *malloc* and *free* are liable to result in heap corruption. ✖ Did you use the *-lmcheck* option? Did you put it right at the end of the build command?
- [ ] When *free* is called, the program aborts with a message that memory has been clobbered past the end of the allocated block.
- [ ] You get a segmentation fault.

Incorrect

Marks for this submission: 0.00/1.00. Accounting for previous tries, this gives **0.67/1.00**.

Information

Introducing valgrind

The program *valgrind* (named after the main entrance to Valhalla from Norse Mythology) is an invaluable debugging tool when you're developing programs that use dynamic memory. It checks each memory reference you make into the heap to ensure you're staying within the bounds of your allocated block(s). This means errors are picked up as they occur rather than just resulting in later failures through corruption of other variables. *valgrind* performs other useful debugging checks, such as identifying at what line in your code a segmentation fault occurred and telling you when you're using uninitialised data. It also checks when the program terminates that all memory you allocated using *malloc* or *realloc* has been correctly freed.

Valgrind is introduced in the video 9.5 Dynamic memory errors.

On the downside, *valgrind* dramatically slows down program execution and cannot be used on large programs. It's also available only under Linux.

Valgrind is documented in full at http://valgrind.org

Running a program with *valgrind* is trivial. Rather than executing a command as, say,

```
./myprog arg1 arg2 < blah
```

you just insert the word *valgrind*, followed by a space, at the start, e.g.

```
valgrind ./myprog arg1 arg2 < blah
```

You can, if you wish, set the geany build command for *Execute* to

```
valgrind "./%e"
```

which results in your using *valgrind* whenever you run code.

When the program terminates, *valgrind* generates screeds of output, which can be rather daunting. Much of that output need not concern you unless you're interested in the run time statistics of your program. But you should look out for the following messages in the output.

valgrind messages

Example message	Meaning

`Invalid write of size 4`
`  at 0x400C41: printStudents (structexample5.c:160)`
`  by 0x400CBA: main (structexample5.c:173)`
`  Address 0x0 is not stack'd, malloc'd or (recently) free'd`
	You tried assigning to (i.e., writing data to) an unavailable memory location (probably a segmentation fault). The message tells you where in your code the sin was committed (line 160 of structexample5.c), and the call stack at the time plus the memory address you were trying to access (0x0). Note that 0x0 is the NULL pointer.

`Invalid write of size 1`
`  at 0x4C3106F: strcpy (in ... irrelevant)`
`  by 0x4009B1: newStudent (hacking.c:48)`
`  by 0x400B3A: readOneStudent (hacking.c:106)`
`  by 0x400B6C: readStudents (hacking.c:122)`
`  by 0x400C80: main (hacking.c:169)`
`  Address 0x5205409 is 0 bytes after a block of size 9 alloc'd`
`  at 0x4C2DB8F: malloc (in ... irrelevant)`
`  by 0x400985: newStudent (hacking.c:47)`
`  by 0x400B3A: readOneStudent (hacking.c:106)`
`  by 0x400B6C: readStudents (hacking.c:122)`
`  by 0x400C80: main (hacking.c:169)`
	You assigned to a memory address in the heap that was just outside (by 0 bytes!) an allocated block. This is probably an out-by-one error - did you forget to leave space for a string terminator? You get told the state of the call stack at the time of the memory access and also the call stack at the time you malloc'd that particular block of memory - the one you just fell off the end of.

Example message	Meaning

`==12140== Conditional jump or move depends on uninitialised value(s)`
`==12140==  at 0x4C32D08: strlen (in ... irrelevant)`
`==12140==  by 0x4EBC9D1: puts (ioputs.c:35)`
`==12140==  by 0x108AAB: readNode (twaddle.c:35)`
`==12140==  by 0x108CF0: main (twaddle.c:63)`
	The statement "Conditional jump or move depends on uninitialised value(s)" means your code is testing the value of a memory location that has not been defined. This might be because you have not initialised a variable or because you've fallen off the end of an array. In this particular case, because it's in the library strlen function, which is being called by puts, you're probably printing an unterminated string.

`HEAP SUMMARY:` `  in use at exit: 190 bytes in 6 blocks`	You leaked memory, i.e. when the program finished you had not freed 190 bytes of memory that you had allocated, in a total of 6 different mallocs. To find out more, add the argument `--leak-check=full` to valgrind.

Obviously you should look for invalid reads from memory as well as invalid writes to it.

Question **3**

Correct

Mark 3.00 out of 3.00

Write a function with signature *int* ramp(int n)* that returns a pointer to a *n*-element dynamically-allocated array of ints containing the values 1, 2, 3, ... n. You may assume *n* is at least 1.

In this question and many others in the lab, your code is run under *valgrind.* However, the output from *valgrind* will not appear in your result table unless it contains error messages like those in the previous information panel; then the whole lot will be displayed.

You are advised to check your code with *valgrind* too, before submitting. Try making some deliberate errors in the program - for example, allocating one *int* too few or forgetting to multiply by sizeof(int) when computing the required space - to see what error message you get. Don't forget to fix the errors again before submitting!

For example:

Test	Result

`int* data = ramp(5);`
`for (int i = 0; i < 5; i++) {`
`    printf("%d ", data[i]);`
`}`
`free(data);`
	`1 2 3 4 5`

Answer: (penalty regime: 0, 10, ... %)

```
1  int* ramp(int n)
2  {
3      int* ramp = malloc(n * sizeof(int));
4      for (int i = 0; i != n; i++) {
5          ramp[i] = i + 1;
6      }
7      return ramp;
8
9
10
11 }
```

	Test	Expected	Got	

✔	`int* data = ramp(5);`
	`for (int i = 0; i < 5; i++) {`
	`    printf("%d ", data[i]);`
	`}`
	`free(data);`
		`1 2 3 4 5`	`1 2 3 4 5`	✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Information

As a general rule of good style, programs that allocate memory dynamically should free it again before they finish, even though the operating system will always clean up afterwards anyway. In the previous example the main program called *randomArray*, which allocated memory using *malloc* and returned a pointer to it. This approach places the onus on the caller of the function *randomArray* to free up the memory it has been given. Failure to do would result in a "memory leak" (i.e. dynamically allocated memory that is never freed), which could be significant if the *randomArray* function were called repeatedly in a loop, say. Furthermore, having the calls to *malloc* and *free* in two different functions makes it much harder for someone reading the code to prove that all dynamically allocated memory is indeed freed again.

For the above reasons it is often considered bad style to write functions that return pointers to dynamically allocated memory. Sometimes it *is* the best approach but you should always consider the alternative of having the caller allocate the memory instead, passing a pointer to it into the function as a parameter. For example, the above example could be rewritten as:

```
/* A an alternative design approach to the problem of
   creating and initialising an array of random numbers.
   RJL, August 2015.
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Fill a given array of doubles with random values in [0,1)
 */
void fillRandomArray(double* numbers, int n)
{
    for (int i = 0; i < n; i++) {      // Fill with random numbers
        numbers[i] = (double) rand() / RAND_MAX;
    }
}

int main(void)
{
    int n = 0;
    double* randomNums = NULL;
    printf("How many random doubles? ");
    scanf("%d", &n);
    randomNums = malloc(n * sizeof(double));
    fillRandomArray(randomNums, n);
    for (int i = 0; i < n; i++) {
        printf("%.6f\n", randomNums[i]);
    }
    printf("\n");
    free(randomNums);
    return 0;
}
```

With this modified design it is easy to pair up the call to *malloc* with the matching call to *free*, thereby guaranteeing that there is no memory leak. On the other hand, disallowing functions that return newly created objects can lead to a much clumsier and error prone design in which most functions are of type void and operate through pointer parameters.

There is no simple "right answer" as to which is the best way.

Question **4**

Correct

Mark 2.00 out of 2.00

The previous CodeRunner question asked you to write a ramp function that returned a pointer to a newly allocated int array. As with the *randomArray* example, you could alternatively have been asked to write a function with signature *void fillRamp(int* data, int n)* that was given a pointer to an already-allocated array, and which filled it with the required data.

Write that function.

For example:

Test	Result

`int* data = malloc(4 * sizeof(int));`
`fillRamp(data, 4);`
`for (int i = 0; i < 4; i++) {`
`  printf("data[%d] = %d\n", i, data[i]);`
`}`
`free(data);`
	`data[0] = 1`
	`data[1] = 2`
	`data[2] = 3`
	`data[3] = 4`

Answer: (penalty regime: 0, 10, ... %)

```
1 void fillRamp(int* data, int n)
2 {
3     for (int i = 0; i < n; i++) {
4         data[i] = i + 1;
5     }
6 }
```

	Test	Expected	Got	

✔	`int* data = malloc(4 * sizeof(int));`
	`fillRamp(data, 4);`
	`for (int i = 0; i < 4; i++) {`
	`  printf("data[%d] = %d\n", i, data[i]);`
	`}`
	`free(data);`
		`data[0] = 1`
		`data[1] = 2`
		`data[2] = 3`
		`data[3] = 4`
			`data[0] = 1`
			`data[1] = 2`
			`data[2] = 3`
			`data[3] = 4`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 2.00/2.00.

Information

`structExample3.c` revisited

The program `structExample3.c` in the last lab had a function

```
Student* newStudent(char* name, int age);
```

that took the next unused array element (a `Student` struct) from a global array called `studentPool`, filled it in with the given `name` and `age`, and returned a pointer to it. The last lab mentioned two problems with that approach:

1. We don't know how big to make the pool of student structs, i.e., what value to use for the symbol MAX_NUM_STUDENTS
2. We don't know how big to make the student's name array, i.e., what value to use for the symbol MAX_NAME_SIZE

Both of these problems can to some extent be handled by allocating arrays that are large enough for any forseeable use, but this is somewhere between inelegant and perilous. So: it's `malloc` and `free` to the rescue!

`structexample4.c`

Inspect the following code for program `structexample4.c`.

Hide `structexample4.c` code

```
/* structExample4.c
 * structExample3.c rewritten to use dynamically allocated memory
 * for the students and also for their names.
 *
 * WARNING: this program has a deliberate bug that causes a memory
 * leak.
 *
 * Author: Richard Lobb
 * Date: 13 August 2014
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_LINE_LENGTH 80      // The longest line this program will accept

/* Note that the definitions for the maximum number of students and
 * the maximum length of a student's name are no longer required.
 */

/* The declaration of the student record (or struct). Note that
 * the struct now contains a pointer to a dynamically allocated
 * name rather than reserving a fixed amount of space within the
 * struct for the name.
 */
typedef struct student_s Student;

struct student_s {
    char* name;
    int age;
    Student* next;      // Pointer to next student in a list
};

// *** The pool of students has been deleted from here. ***

/*
 * Return a pointer to a new dynamically allocated student struct
 * that contains a pointer to a dynamically-allocated name string.
 * Returns NULL if either malloc fails.
 * Note the need to allocate one byte more than the length of the
 * string when allocating space for a copy of it, in order to make
 * space for the null terminator.
 */
Student* newStudent(const char* name, int age)
{
    size_t nameLength = strlen(name);
    Student* student = malloc(sizeof(Student));
    if (student != NULL) {
        if ((student->name = malloc(nameLength + 1)) != NULL) {
            strncpy(student->name, name, nameLength + 1);
            student->age = age;
            student->next = NULL;
        } else {  // No space for name
            free(student);
            student = NULL;
        }
    }
    return student;
}


// Free the memory allocated for a student and their name
void freeStudent(Student* student)
{
    free(student->name); // Must do this first (why?)
    free(student);
}


/* Read a single student from a csv input file with student name in first column,
 * and student age in second.
 * Returns: A pointer to a Student record, or NULL if EOF or an invalid
 * student record is read. Blank lines, or lines in which the name is
 * longer than the provided name buffer, or there is no comma in the line
 * are considered invalid.
 */
Student* readOneStudent(FILE* infile)
```

```
{
    char buffer[MAX_LINE_LENGTH];  // Buffer into which we read a line from stdin
    Student* student = NULL;           // Pointer to a dynamically allocated student record

    // Read a line, extract name and age

    char* line = fgets(buffer, MAX_LINE_LENGTH, infile);
    if (line != NULL) {           // Proceed only if we read something
        char* commaPos = strchr(buffer, ',');
        if (commaPos != NULL) {
            int age = atoi(commaPos + 1);
            *commaPos = '\0';  // null-terminate the name
            student = newStudent(buffer, age);
        }
    }
    return student;
}

/* Reads a list of students from a given file. Input stops when
 * a blank line is read, or an EOF occurs, or an illegal input
 * line is encountered.
 * Returns: a pointer to the first student in the list or NULL if no
 * valid student records could be read.
 */
Student* readStudents(FILE* infile)
{
    Student* first = NULL;      // Pointer to the first student in the list
    Student* last = NULL;       // Pointer to the last student in the list
    Student* student = readOneStudent(infile);
    while (student != NULL) {
        if (first == NULL) {
            first = last = student;    // Empty list case
        }
        else {
            last->next = student;
            last = student;
        }
        student = readOneStudent(infile);
    }
    return first;
}


// Free all the memory allocated to a list of students.
void freeAllStudents(Student* student)
{
    Student* next = NULL;
    while (student != NULL) {
        next = student->next;
        free(student);
        student = next;
    }
}

// printOneStudent: prints a single student, passed by value
void printOneStudent(Student stud)
{
    printf("%s (%d)\n", stud.name, stud.age);
}


/* printStudents: print all students in a list of students, passed
 * by reference */
void printStudents(const Student* studPtr)
{
    while (studPtr != NULL) {
        printOneStudent(*studPtr);
        studPtr = studPtr->next;
    }
}


int main(void)
{
    FILE* inputFile = fopen("studlist.txt", "r");
    if (inputFile == NULL) {
        fprintf(stderr, "File not found\n");
    }
    else {
```

```
        Student* studentList = readStudents(inputFile);
        fclose(inputFile);
        printStudents(studentList);
        freeAllStudents(studentList);
    }
}
```

Note the following:

1. The program contains a bug that causes a memory leak. We'll worry about that a bit later.
2. The "pool" of student structs has been dispensed with. Students are now allocated dynamically from the heap, so there is no fixed maximum number of students that can be allocated.
3. The `student` struct has been changed so that the `name` field is just a single pointer to the name rather than an array of characters containing the name. The memory required to store the name separately from the student record is dynamically allocated. This removes the restriction on the maximum length of a student name.
4. There are now functions `freeStudent` and `freeAllStudents` that free the memory dynamically allocated to a single student and a list of students respectively.
5. The two constants MAX_NUM_STUDENTS and MAX_NAME_LENGTH have both been dispensed with. There is still an arbitrary MAX_LINE_LENGTH constant remaining but we won't worry about that just yet.
6. When you allocate space for a string, you must allocate one more byte than the string length in order to hold the terminating null character.
7. Once `free` has been called with a particular pointer, you MUST NOT use the memory it was pointing at again! The chunk of memory returned to the heap may be reused almost immediately by library functions like `printf` or other ones you didn't even know were being called (e.g. debugger functions, if you're trying to debug your program)! This can lead to some very nasty hard-to-find bugs.
8. If your program runs out of memory (perhaps after `malloc`ing several gigabytes), `malloc` will return NULL (zero). This is probably a bug (a combination of an infinite loop and a "memory leak" - see later) rather than a validly occuring event. The approved wisdom is that your program should "degrade gracefully" when it runs out of memory, rather than just crashing with an error message, but it is difficult in general to do this. The `newStudent` function avoids a crash by simply returning a null student but it's a moot point whether that is actually better than deliberately crashing the program. The style guideline for ENCE260 is that you should always at least check the return value from `malloc`, but it is acceptable for you to print an error message and call `exit` or `abort` immediately, rather than attempting to keep running, as is done here.

Build and run `structexample4.c` using the same input file `studlist.txt` as in the previous lab. It should still work as before.

Now run `structexample4` with the command

```
valgrind ./structexample4 < studlist.txt
```

or, for more detailed information,

```
valgrind --leak-check=full ./structexample4 < studlist.txt
```

Where's the bug, i.e., why is memory being lost?

Question **5**

Correct

Mark 1.00 out of 1.00

If *Point* is a struct with integer fields *x* and *y* and a *next* field that can be used to point to the next Point in a list, make the appropriate selections from the drop-down menus in the following code segment so that it correctly frees a list of items of type Point. [It should be correctly indented, too.]

```
void freePoints(Point* p)
{
    Point *ptr = NULL;        ✔
    while (p != NULL) {       ✔
        ptr = p->next;        ✔
        free(p);              ✔
        p = ptr;              ✔
    }
}
```

Correct

Marks for this submission: 1.00/1.00.

Question **6**

Correct

Mark 1.00 out of 1.00

The *freeStudent* function begins with the line

```
free(student->name); // Must do this first (why?)
```

Why should that call to *free* precede the one that follows it?

Select one:

- Because after calling *free(student) , student* will be NULL so you can't refer to its *name* field.
- Because after calling *free(student)* it is no longer safe to reference the memory area pointed to by *student*. ✔ Yes, that's the right answer. It's true that you will probably get away with it in this case, but in general it's dangerous, particularly once you get into multithreaded programs.
- Because the call to *free(student),* if done first, will call *free(student->name)* itself and you'd then free the name twice.

Correct

Marks for this submission: 1.00/1.00.

Question **7**

Correct

Mark 1.00 out of 1.00

How many bytes does *valgrind* report are definitely lost (i.e. not freed) by *structexample4.c* (assuming standard input is from the file *studlist.txt*) ?

Answer: 132 ✔

Correct

Marks for this submission: 1.00/1.00.

Question **8**

Correct

Mark 1.00 out of 1.00

What's the cause of the "lost bytes" that *valgrind* reports?

Select one:

- The memory allocated to student names is not being freed, because *freeAllStudents* isn't freeing each student correctly. ✔ That's right! It should be calling *freeStudent(student)*, not *free(student)*!
- There is one byte unfreed in each student name because of the *nameLength + 1* parameter to *malloc*.
- It's nothing to do with the freeing of students -- it's a result of the input file not being closed.
- There is one student still unfreed, because *freeAllStudents* doesn't free the student at the head of the list.
- There is one student still unfreed, because *freeAllStudents* doesn't free the student at the end of the list.

Correct

Marks for this submission: 1.00/1.00.

Information

Growing a memory block

`structexample4.c` contains one remaining arbitrary constant - MAX_LINE_LENGTH - to set the size of the line buffer into which a line from the input file should be read. We'll now see how even that constant can be eliminated by use of the *realloc* function, explained in the video *9.4 realloc*

Examine the following new version (`structexample5.c`).

Hide `structexample5.c` code

```
/* structexample5.c
 * This is structexample4.c further modified to eliminate the need for an
 * arbitrary upper limit on the length of an input line.
 *
 * WARNING: this program has a deliberate bug that causes a memory
 * leak.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define BUFFER_INCREMENT 10  // An artificially small value for demo purposes

/* Note that the definitions for the maximum number of students and
 * the maximum length of a student's name are no longer required.
 */

/* The declaration of the student record (or struct). Note that
 * the struct now contains a pointer to a dynamically allocated
 * name rather than reserving a fixed amount of space within the
 * struct for the name.
 */
typedef struct student_s Student;

struct student_s {
    char* name;
    int age;
    Student* next;      // Pointer to next student in a list
};


/*
 * Return a pointer to a new dynamically allocated student struct
 * that contains a pointer to a dynamically-allocated name string.
 * Returns NULL if either malloc fails.
 * Note the need to allocate one byte more than the length of the
 * string when allocating space for a copy of it, in order to make
 * space for the null terminator.
 */
Student* newStudent(const char* name, int age)
{
    size_t nameLength = strlen(name);
    Student* student = malloc(sizeof(Student));
    if (student != NULL) {
        if ((student->name = malloc(nameLength + 1)) != NULL) {
            strncpy(student->name, name, nameLength + 1);
            student->age = age;
            student->next = NULL;
        } else {  // No space for name
            free(student);
            student = NULL;
        }
    }
    return student;
}


/* Free the memory allocated for a student and their name */
void freeStudent(Student* student)
{
    free(student->name); /* Must do this first (why?) */
    free(student);
}


/* Read a single student from a csv input file with student name in first column,
 * and student age in second.
 * Returns: A pointer to a Student record, or NULL if EOF or an invalid
 * student record is read. Blank lines, or lines in which the name is
 * longer than the provided name buffer, or there is no comma in the line
 * are considered invalid.
 */
Student* readOneStudent(FILE* infile)
{
    char* buff = NULL;     // Pointer to a dynamically allocated line buffer
    Student* student = NULL;   // Pointer to a dynamically allocated student record
    int buffSize = 0;      // Number of bytes allocated to line buffer
```

```
int i = 0;             // Line buffer index

// Read a line, increasing buffer size as required to handle the line

int c = '\0';   // Input characters go here
while ((c = fgetc(infile)) != EOF && c != '\n') {
    if (i >= buffSize - 1) {             // Out of space?
        buffSize += BUFFER_INCREMENT;    // Yes. Make buffer bigger
        buff = realloc(buff, buffSize);  // Grow the space (copies if necessary)
        if (buff == NULL) {
            fprintf(stderr, "Out of memory");
            exit(EXIT_FAILURE);
        }
    }
    buff[i] = c;
    i += 1;
}

// Unless buff is NULL, we now have an input line. Extract name and age

if (buff != NULL) {
    buff[i++] = '\0'; // Terminate the string
    char* commaPos = strchr(buff, ',');
    if (commaPos != NULL && commaPos > buff) {
        int age = atoi(commaPos + 1);
        *commaPos = '\0';  // null-terminate the name
        student = newStudent(buff, age);
        }
    }
    return student;
}

/* Reads a list of students from a given file. Input stops when
 * a blank line is read, or an EOF occurs, or an illegal input
 * line is encountered.
 * Returns: a pointer to the first student in the list or NULL if no
 * valid student records could be read.
 */
Student* readStudents(FILE* infile)
{
    Student* first = NULL;      // Pointer to the first student in the list
    Student* last = NULL;       // Pointer to the last student in the list
    Student* student = readOneStudent(infile);
    while (student != NULL) {
        if (first == NULL) {
            first = last = student;  // Empty list case
        }
        else {
            last->next = student;
            last = student;
        }
        student = readOneStudent(infile);
    }
    return first;
}


// Free all the memory allocated to a list of students.
void freeAllStudents(Student* student)
{
    Student* next = NULL;
    while (student != NULL) {
        next = student->next;
        freeStudent(student);
        student = next;
    }
}

// printOneStudent: prints a single student, passed by value
void printOneStudent(Student stud)
{
    printf("%s (%d)\n", stud.name, stud.age);
}


/* printStudents: print all students in a list of students, passed
 * by reference */
void printStudents(const Student* studPtr)
{
```

```
    while (studPtr != NULL) {
        printOneStudent(*studPtr);
        studPtr = studPtr->next;
    }
}


int main(void)
{
    FILE* inputFile = fopen("studlist.txt", "r");
    Student* studentList = readStudents(inputFile);
    fclose(inputFile);
    printStudents(studentList);
    freeAllStudents(studentList);
    return EXIT_SUCCESS;
}
```

Note the following:

1. The read loop in `readOneStudent` now uses `fgetc`, which gets characters one-by-one from a file, rather than `fgets`. This makes the logic a little easier. Some people object to the use of this function, arguing that it's "inefficient". However, data is still read from the file a block at a time (probably a kilobyte or two) and `fgetc` is a simple function that just passes the next byte back to the caller each time until it needs to read another block. Even with Solid State Drives (SSDs), the time taken to read a block from the file system significantly exceeds the time taken to process the bytes one by one, so trying to get more "efficiency" by use of `fgets` is just wasted effort.
2. Like `getchar`, `fgetc` returns an `int`, not a `char`. The distinction is important if the check for EOF is to be valid. Whether you get away with declaring it to be a `char` or not depends on whether the compiler treats bytes as signed or unsigned (the language spec leaves this open) and whether or not there's a character with the value octal 377 in the file. In short, declaring c as `char` results in flakey and unreliable code.
3. The constant MAX_LINE_LENGTH has gone, but there's a new constant BUFFER_INCREMENT, which sets the amount by which the buffer is grown each time it fills up. The value of this is somewhat arbitrary but affects only the run-time efficiency of the program, not its correctness.
4. Function `realloc` is used to dynamically resize the line buffer whenever it fills up. The first parameter to `realloc` is a pointer to a dynamically allocated buffer and the second parameter is the required new size for that buffer.
5. The buffer pointer variable `buff` is initially NULL and `buffSize` is zero, meaning that initially no space is allocated to hold the string being read from the input file. On the first time through the read loop, `realloc` will be called with the first parameter (the pointer to a buffer) NULL. It then behaves exactly like `malloc`, creating a new block of memory of size BUFFER_INCREMENT (10 bytes in this case, though that's an artificially small value, chosen to demonstrate how `realloc` works).
6. Every 10 times through the loop, `realloc` will be called to dynamically resize the buffer. It is VERY IMPORTANT to note that `realloc` returns a pointer to a *new* buffer, which may or may not be at the same place in memory as the old one. [Due to memory fragmentation it might not be possible to leave the buffer at the same place in memory as it was before.] If it's at a new place, `realloc` will have copied across the contents of the old block to the new one and freed the old one. Nasty bugs can arise if:
    - You simply ignore the return value and assume the buffer doesn't move. This will quite likely work for a while but eventually the buffer is sure to need moving and you'll start writing into an unallocated chunk of the heap.
    - You have any other pointers that point at or into the data block you `malloc`ed. The pointers will be invalid if the block moves.
7. This program grows the buffer by a fixed increment - currently set at 10 - every time. In practice you'd use a much larger value than 10 for such an increment: that's a ridiculously small amount of memory on a modern computer and the cost of doing a `realloc` every 10 bytes could be significant in some applications. If you wanted to further improve the performance of the memory allocator (e.g., to reduce the fragmentation of the heap) you could use more sophisticated strategies, such as doubling the buffer size each time.

Build and run `structexample5.c`. You should find it works just as before. Except it, too, has a memory leak, as shown by running it under `valgrind`. Where's the memory leak this time? Fix it!

Question **9**

Correct

Mark 1.00 out of 1.00

Which of the following will fix the memory leak in *structexample5.c*?

Select one:

- a. Append the statement `free(buff)` to the end of the body of the while loop, immediately after `i += 1`
- b. Append the statement `free(buff)` to the end of the block that's executed only if `(buff != NULL)`. ✔
- c. Insert the statement `free(buff)` after the while loop, just before `if (buff != NULL)`.
- d. Insert the statement `free(buff)` just after `if (buff != NULL) {`
- e. Insert the statement `free(buff)` immediately after the call to `realloc`.

Well done. Note that it's also okay to insert *free(buff)* immediately before *return student*. That will mean it's unconditionally executed, but that doesn't matter, since *free(buff)* does nothing if *buff* is NULL

Correct

Marks for this submission: 1.00/1.00.

Question **10**

Partially correct

Mark 1.00 out of 1.00

The specimen code in the lab increments the buffer size by some fixed amount whenever it fills up. In the example, the increment was just 10 bytes, but we would usually try to choose an increment that was sufficiently large that we would rarely need to resize the buffer. In that way we would minimise the load on the memory manager. However, if many resizes are required, and if the buffer needs to be copied each time, the cost of building the buffer become quadratic in the buffer size rather than linear, which is potentially catastrophic for large buffers. When dealing with cases where a small buffer may grow without bound, we should instead double the size of the buffer on each resize.

However, if such efficiency considerations were irrelevant, and we didn't mind being accused of writing inelegant code, we could actually increase the buffer size by just 1 byte each time through the loop in order to accommodate each new character.

Choose the appropriate lines of code from the pull-down boxes below so that it reads an arbitrarily long line in that way. All variables have the same types as in the specimen code within the lab.

Note:

1. Not all items available in the pull-down boxes are actually used/usable.
2. Drop-down menu items can be used more than once.
3. Indentation has been omitted to avoid giving too much away.

```
buff = malloc(1);
i = 0;
```
c = fgetc(fp); ✔

while (c != EOF && c != '\n ') { ✔

buff = realloc(buff, i + 1); ✖

```
buff[i] = c;
i += 1;
```
c = fgetc(fp); ✔

```
}
buff[i]= '\0';
```

Partially correct

Marks for this submission: 0.75/1.00. Accounting for previous tries, this gives **1.00/1.00**.

Information

Heap corruption

Another major issue when using dynamic memory is heap corruption, introduced in the video 9.5 Dynamic memory errors. A classic error is to allocate `strlen(s)` bytes as a buffer to hold a string `s`, forgetting that you need one extra byte to hold the null terminator. For example:

```
// Return a copy of the string s [BAD CODE]
char* copyString(char* s)
{
    int len = strlen(s);
    char* copy = malloc(len);
    strcpy(copy, s);
    return copy;
}
```

To see this sort of problem in action, change the `malloc` call within the `newStudent` function in `structexample5.c` from `malloc(nameLength + 1)` to just `malloc(nameLength)`. Then re-make `structexample5` and run it with the command `./structexample5` and also with the command `valgrind ./structexample5`. What happened?

The subsequent call to `strncpy` with a length parameter of `nameLength + 1` would probably alert you to the impending doom in this particular case, but if you had committed the sin of using `strcpy` rather than `strncpy` you'd never have noticed. The extra null byte would quietly have been placed over the top of whatever follows the allocated block. This could be all sorts of interesting things, like the low byte of a pointer used to link the following block on the free list, or the low byte of the size of the next block, or a data byte of some different block, or .... In short, you just corrupted something and you've no idea what. So what goes wrong? Probably nothing -- for a while. Then maybe a few function calls or a few seconds or even a few hours later the program crashes for no obvious reason at all!

`corrupt.c`

Consider now the following program, `corrupt.c`.

Hide `corrupt.c` code

```
/* Program to demonstrate various heap mismanagement problems.
 * Reads lines from a given file into an array of strings,
 * prints them all out and frees them.
 * BUGGY!
 * Written for COSC208, Richard Lobb, 2007, 2008, 2009; ENCE260 2012/3
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_SIZE 100

// Return a copy of the string s
char* copyString(char* s)
{
    int len = strlen(s);
    char* copy = malloc(len);
    strcpy(copy, s);
    return copy;
}

/* Read lines from the file into a dynamically allocated array of strings.
 * Returns a pointer to the array of strings, which is terminated by an
 * additional NULL pointer (i.e., there are n + 1 char* values in the array,
 * with the last value being NULL).
 * Each string will usually end with a newline character.
 */
char** readFile(FILE *fp)
{
    char** stringBuffer = NULL;
    int numStrings = 0;
    char line[MAX_LINE_SIZE];
    stringBuffer[0] = NULL;
    while (fgets(line, MAX_LINE_SIZE, fp) != NULL) {
        stringBuffer = realloc(stringBuffer, numStrings + 1);
        stringBuffer[numStrings] = copyString(line);
        numStrings += 1;
    }
    stringBuffer[numStrings] = NULL;  // Terminator
    return stringBuffer;
}

// Print all the strings in the given string array (terminated by NULL)
void printStrings(char* s[])
{
    int i = 0;
    while (s[i] != NULL) {
        printf("%s", s[i]);
        i += 1;
    }
}

// Free all the strings in the given string array (terminated by NULL)
void freeAllStrings(char* s[])
{
    int i = 0;
    while (s[i] != NULL) {
        free(s[i]);
        i += 1;
    }
}


int main(int argc, char* argv[])
{
    FILE* inputFile = NULL;
    char** stringArray = NULL;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s file\n", argv[0]);
        return EXIT_FAILURE;
    }
    inputFile = fopen(argv[1], "r"); /* Open file for reading */
    if (inputFile == NULL) {
        fprintf(stderr, "Can't open file '%s'\n", argv[1]);
        return EXIT_FAILURE;
    }
    stringArray = readFile(inputFile);
    printStrings(stringArray);
```

```
    freeAllStrings(stringArray);
    free(stringArray);
    fclose(inputFile);
    return EXIT_SUCCESS;
}
```

This program reads a whole file, one line at a time, into an array of strings. So that the caller of *readFile* knows the size of the array being returned, *readFile* adds an extra NULL pointer to the array as a terminator. The program then prints all those strings, frees them, frees the array of strings and exits. But ... oh dear ... it has a few problems. There are three or four small but potentially catastrophic errors. See if you can find them and fix them.

Question **11**

Partially correct

Mark 1.00 out of 1.00

Which of the functions in *corrupt.c* had major errors in their dynamic-memory handling code? [A "major error" is one that is liable to lead to heap corruption or memory leaks.] You must select all the bad functions to get full marks. Selecting a good function, i.e., one without major errors, will be penalised.

Select one or more:

- [x] readFile ✔
- [ ] printStrings
- [ ] main
- [x] freeAllStrings ✖
- [x] copyString ✔

Partially correct

Marks for this submission: 0.70/1.00. Accounting for previous tries, this gives **1.00/1.00**.

Question **12**

Correct

Mark 1.00 out of 1.00

With regard to the following code fragment, which of the statements below are correct? You must select *all* correct answers. [Yes, the use of *strcpy* is forbidden in this course, but you should still know about it and be able to answer the question.]

```
// Return a copy of the string s.
// It is assumed that the caller will call free when the copy is no longer needed.
char* copyString(char* s)
{
    int len = strlen(s);
    char* copy = malloc(len);
    strcpy(copy, s);
    return copy;
}
```

Select one or more:

- [ ] The code doesn't compile because strcpy takes an additional parameter to say how many bytes to copy.
- [ ] Although the code uses *strcpy* (which is considered bad style in ENCE260) it actually works correctly.
- [ ] It has a memory leak -- it should call *free(copy)* just before returning.
- [x] The parameter to *malloc* should be *len* + *1*, not *len*. ✔ Yes that's right -- you need space for the terminating null byte.

Correct

Marks for this submission: 1.00/1.00.

Information

Three more coding exercises

The following three CodeRunner questions will give you a bit more practice with using dynamic memory.

Question **13**

Correct

Mark 3.00 out of 3.00

Write a function *char* skipping(const char* s)* that takes a C string as input and returns a new dynamically-allocated C string containing every second character of *s* starting with the first one.

For example:

Test	Result

`char* s = skipping("0123456789");`
`printf("%s\n", s);`
`free(s);`
	`02468`

Answer: (penalty regime: 0, 10, ... %)

```
char* skipping(const char* s)
{
    //i allocate 1 more memory slot for string terminator
    //am i not allocating double the required memory here?
    char* data = malloc((strlen(s) + 1) * sizeof(int));
    int counter = 0;
    //add every 2nd char to a new string ( data )
    for (int i = 0; (i <= strlen(s)); i += 2) {
        data[counter] = s[i];
        counter++;
    }
    data[counter] = '\0';
    return data;
}

```

	Test	Expected	Got	

✔	`char* s = skipping("0123456789");`
	`printf("%s\n", s);`
	`free(s);`
		`02468`	`02468`	✔

✔	`char* s = skipping("A0B1C2D");`
	`printf("%s\n", s);`
	`free(s);`
		`ABCD`	`ABCD`	✔

✔	`char* s = skipping("");`
	`printf("%s\n", s);`
	`free(s);`
				✔

✔	`char* s = skipping("X");`
	`printf("%s\n", s);`
	`free(s);`
		`X`	`X`	✔

✔	`char buff[] = {'x', 0, 'y', 0, 0};`
	`char* s = skipping(buff);`
	`printf("%s\n", s);`
	`free(s);`
		`x`	`x`	✔

✔	`char* s = skipping("XY\nab\n");`
	`printf("%s\n", s);`
	`free(s);`
		`X`
		`b`
			`X`
			`b`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Question **14**

Correct

Mark 3.00 out of 3.00

A type StringPair, representing a pair of strings, could be defined by the following:

```
typedef struct stringpair_s {
    char* first;
    char* second;
 } StringPair;
```

Given the existence of such a typedef, write a function with signature

```
StringPair* newStringPair(const char* s1, const char* s2)
```

which returns a pointer to a new dynamically allocated *StringPair* structure that contains pointers to two newly created *copies* of the parameter strings *s1* and *s2*.

To develop and test your code, start by copying the program skeleton below. Then insert your function at the place marked. When you have it working, copy and paste *just the function definition* into the answer box below.

The output from the program when it's working correctly should be

```
String pair: ('My first string', 'Another one')
```

Make sure you understand the skeleton code before writing your function. If in doubt, ask a tutor for help.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

// Declare the StringPair type.
// Note that we have incorporated the struct declaration into
// the typedef, but that this only works because we don't have any
// StringPair pointers in the structure (e.g. StringPair* next).
typedef struct stringpair_s {
    char* first;
    char* second;
 } StringPair;

// **** Insert your newStringPair function definition here ***

int main(void)
{
    char s1[] = "My first string";
    char s2[] = "Another one";
    StringPair* pair = NULL;

    pair = newStringPair(s1, s2);

    // Before printing, alter the initial strings to ensure
    // the function hasn't just copied the pointers.
    strncpy(s1, "Smasher1", sizeof(s1));
    strncpy(s2, "Clobber2", sizeof(s2));

    // Now print the new StringPair.
    printf("String pair: ('%s', '%s')\n", pair->first, pair->second);

    // Lastly free all dynamic memory involved.
    free(pair->first);
    free(pair->second);
    free(pair);
}
```

Answer: (penalty regime: 0, 10, ... %)

```
1  StringPair* newStringPair(const char* s1, const char* s2)
2  {
3      StringPair* newPair = malloc(sizeof(StringPair));
4      char* data1 = malloc((strlen(s1) + 1) * sizeof(char));
5      char* data2 = malloc((strlen(s2) + 1) * sizeof(char));
6      newPair->first = data1;
7      newPair->second = data2;
8      strncpy(newPair->first, s1, (strlen(s1)) * sizeof(char));
9      strncpy(newPair->second, s2, (strlen(s2)) * sizeof(char));
10     newPair->first[strlen(s1)] = '\0';
11     newPair->second[strlen(s2)] = '\0';
12     return newPair;
13
14 }
```

	Test	Expected	Got	

✔	`// Test with the same code as in the spec`
	
	`char s1[] = "My first string";`
	`char s2[] = "Another one";`
	`StringPair* sp = NULL;`
	
	`sp = newStringPair(s1, s2);`
	
	`// Before printing, alter the initial strings to ensure`
	`// the function hasn't just copied the pointers.`
	`strncpy(s1, "Smasher1", sizeof(s1));`
	`strncpy(s2, "Clobber2", sizeof(s2));`
	
	`// Now print the new StringPair.`
	`printf("String pair: ('%s', '%s')\n",`
	`  sp->first, sp->second);`
	
	`// Lastly free all dynamic memory involved.`
	`free(sp->first);`
	`free(sp->second);`
	`free(sp);`
		`String pair: ('My first string', 'Another one')`	`String pair: ('My first string', 'Another one')`	✔

✔	`// A slight variation`
	
	`char s1[] = {'a', 'b', 'c', 0, 'x', 'y', 'z'};`
	`char s2[] = {'1', '2', '3', 0, '*', '*', '*'};`
	`StringPair* sp = NULL;`
	
	`sp = newStringPair(s1, s2);`
	
	`// Before printing, alter the initial strings to ensure`
	`// the function hasn't just copied the pointers.`
	`strncpy(s1, "BAM", sizeof(s1));`
	`strncpy(s2, "POW", sizeof(s2));`
	
	`// Now print the new StringPair.`
	`printf("String pair: ('%s', '%s')\n",`
	`  sp->first, sp->second);`
	
	`// Lastly free all dynamic memory involved.`
	`free(sp->first);`
	`free(sp->second);`
	`free(sp);`
		`String pair: ('abc', '123')`	`String pair: ('abc', '123')`	✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Question **15**

Correct

Mark 3.00 out of 3.00

In a certain program, the structure for representing a person is defined as:

```
typedef struct {
    char* name;
    int age;
    double height;
} Person;
```

Given this declaration for `Person`, write two functions `newPerson` and `freePerson` with the signatures:

```
Person* newPerson(char* name, int age, double height);
void freePerson(Person* person);
```

The `newPerson` function returns a pointer to a newly allocated Person on the heap, with the `name`, `age`, and `height` fields set appropriately. Here, the `name` field must be a dynamically allocated copy of the corresponding string parameter `name`. All memory (de)allocation must be performed using `malloc` and `free`, and each memory allocation should use the exact required block size—no larger or smaller. You do not need to deal with the possibility that *malloc* fails.

The `freePerson` function frees all memory that was allocated by `newPerson` when creating the `Person` currently pointed to by the parameter.

Paste **only** the two functions `newPerson` and `freePerson` into the answer box - the struct declaration will be automatically included.

For example:

Test	Result

`Person* employee = newPerson("Billy", 30, 1.68);`
`printf("%s is age %d and is %.2f m tall\n", employee->name, employee->age, employee->height);`
`freePerson(employee);`
	`Billy is age 30 and is 1.68 m tall`

Answer: (penalty regime: 0, 10, 20, ... %)

```
1  Person* newPerson(char* name, int age, double height)
2  {
3      Person* newHuman = malloc(sizeof(Person));
4      char* personName = malloc(sizeof(char) * strlen(name) + 1);
5      strncpy(personName, name, sizeof(char) * strlen(name));
6      newHuman->name = personName;
7      newHuman->name[strlen(name)] = '\0';
8      newHuman->name = personName;
9      newHuman->height = height;
10     newHuman->age = age;
11     return newHuman;
12
13 }
14
15
16 void freePerson(Person* person)
17 {
18     free(person->name);
19     free(person);
20 }
```

	Test	Expected	Got	

✔	`Person* employee = newPerson("Billy", 30, 1.68);`
	`printf("%s is age %d and is %.2f m tall\n", employee->name, employee->age, employee->height);`
	`freePerson(employee);`
		`Billy is age 30 and is 1.68 m tall`	`Billy is age 30 and is 1.68 m tall`	✔

✔	`char* name = "Einstein";`
	`Person* person = newPerson(name, 76, 1.72);`
	`if (person->name == name) {`
	`  puts("Test feedback: 'name' has not been copied!");`
	`}`
	`freePerson(person);`
	`__check_mem();`
		`Test feedback: total 33 bytes freed`	`Test feedback: total 33 bytes freed`	✔

	Test	Expected	Got	

✔	`Person* p1 = newPerson("nickname", 33, 1.8);`
	`Person* p2 = newPerson("fake name", 99, 1.61);`
	`freePerson(p2);`
	`__check_mem();`
	`freePerson(p1);`
	`__check_mem();`
		`Test feedback:`
		`total 34 bytes freed`
		`Test feedback:`
		`total 67 bytes freed`
			`Test feedback:`
			`total 34 bytes freed`
			`Test feedback:`
			`total 67 bytes freed`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Information

Parting comment

Dynamic memory underpins many of the features we take for granted in higher-level languages like Python and Java, such as strings, lists (`ArrayLists` in Java), dictionaries (`Maps` in Java) and in fact all on-the-fly object creation. However, such object-oriented languages all have a runtime component called a *garbage collector*, which keeps track of allocated memory and intermittently frees any allocated memory that is no longer reachable. The lack of such a component in C makes use of dynamic memory considerably harder because of the risk of memory leaks. For this reason, dynamic memory is generally not used in embedded systems or other mission critical systems that need to function over long time frames without human intervention. See for example The JPL Coding Standards document, which includes the rule "Do not use dynamic memory allocation after task initialization."

◄ Quiz5: Code and Data Structures Jump to... Superquiz 1 ►