9. Dynamic memory allocation

- Motivation
- *malloc/calloc*
- *free*
- How it works
- *realloc*
- Memory leaks
- Heap corruption

Motivating example

- Consider a simulation of a community of rabbits:

```
forever {
    simulationTime += deltaT;
    numRabbitsBorn = currentNumRabbits * birthRate * deltaT;
    for each new rabbit {
        construct new Rabbit record and add to population
    }
    for each rabbit in population {
        if (deathProbability * deltaT * randomInRange(0,1) > 0.5) {
            Destroy rabbit
        else
            Have rabbit do something for its deltaT
    }
}
```

How do we do these steps?

NB: The lab exercises use *Students* rather than *Rabbits* ☺

The “Rabbit Pool” solution

A rudimentary (but highly efficient) single-entity dynamic memory system.

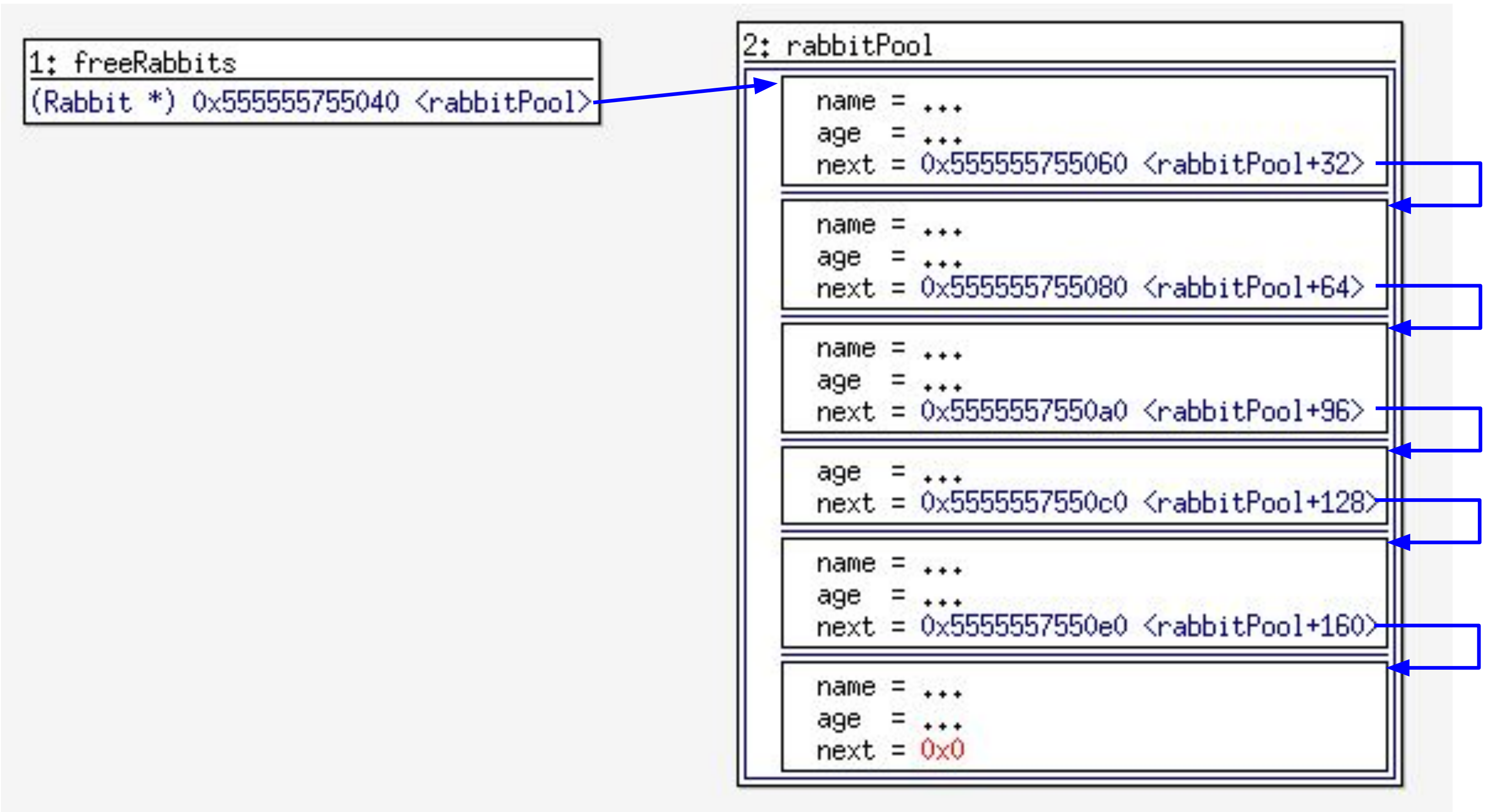
```
typedef struct rabbit_s Rabbit;
struct rabbit_s
{
    char name[MAX_NAME_LENGTH];
    int age;
    Rabbit* next;
};

Rabbit rabbitPool[MAX_NUM_RABBITS];
Rabbit* freeRabbits = NULL;

/* Link all the rabbits in the pool onto the freeRabbits list */
void initialise(void)
{
    for (int i = 0; i < MAX_NUM_RABBITS - 1; i++) {
        rabbitPool[i].next = &rabbitPool[i + 1];
    }
    rabbitPool[MAX_NUM_RABBITS - 1].next = NULL;
    freeRabbits = &rabbitPool[0];
}
```

Continues on next slide

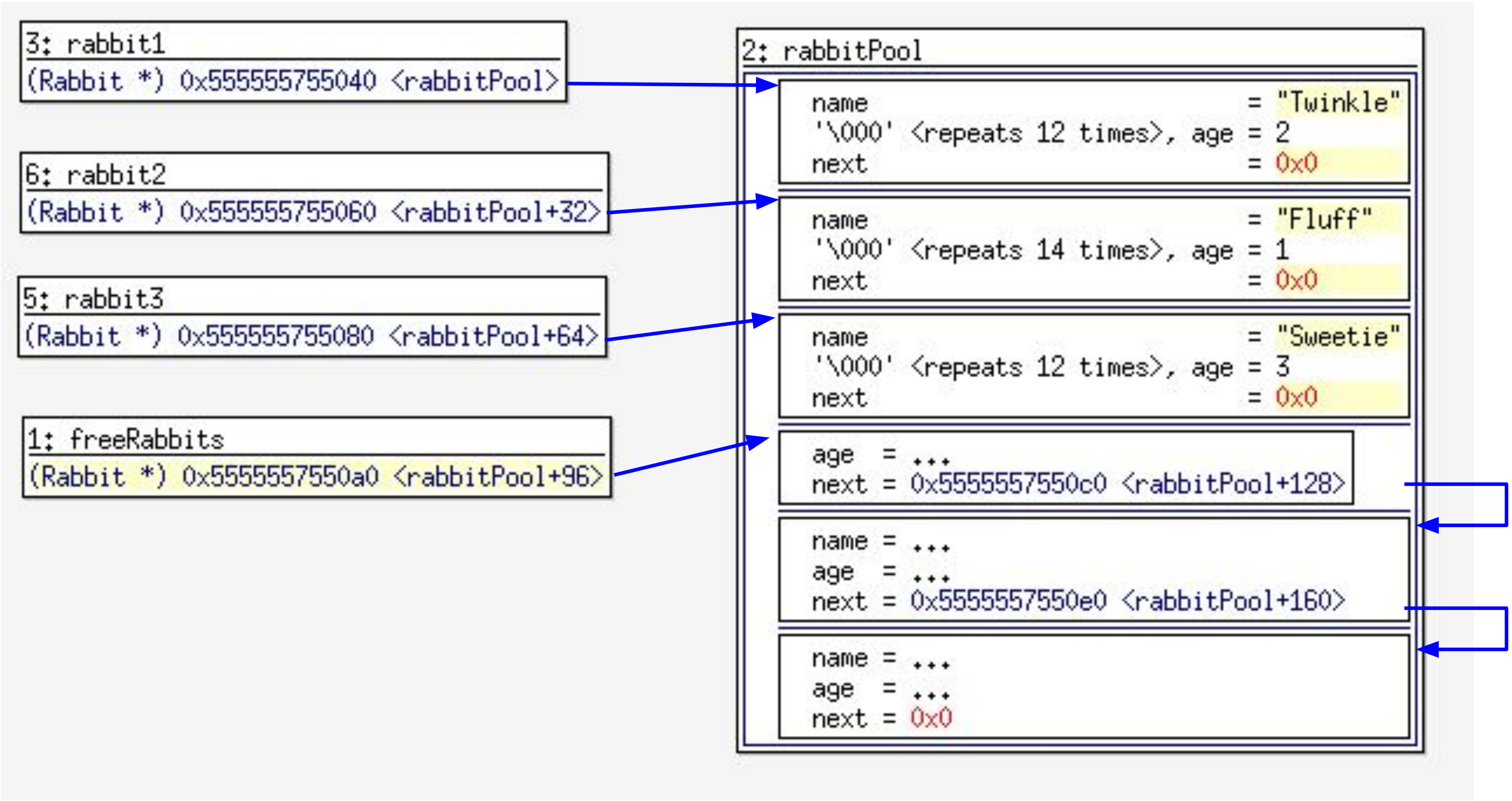
Rabbit pool (cont'd)

```
Rabbit* newRabbit(const char* name, int age) {
    if (freeRabbits == NULL) {
        fprintf(stderr, "Out of rabbits");
        exit(1);
    }
    Rabbit* rabbit = freeRabbits;
    strncpy(rabbit->name, name, MAX_NAME_LENGTH);
    rabbit->name[MAX_NAME_LENGTH - 1] = 0;
    rabbit->age = age;
    freeRabbits = rabbit->next;
    rabbit->next = NULL;
    return rabbit;
}

void freeRabbit(Rabbit* rabbitPtr) {
    rabbitPtr->next = freeRabbits;
    freeRabbits = rabbitPtr;
}

int main(void) {
    initialise();
    Rabbit* rabbit1 = newRabbit("Twinkle", 2);
    Rabbit* rabbit2 = newRabbit("Fluff", 1);
    Rabbit* rabbit3 = newRabbit("Sweetie", 3);
    freeRabbit(rabbit2);
}
```

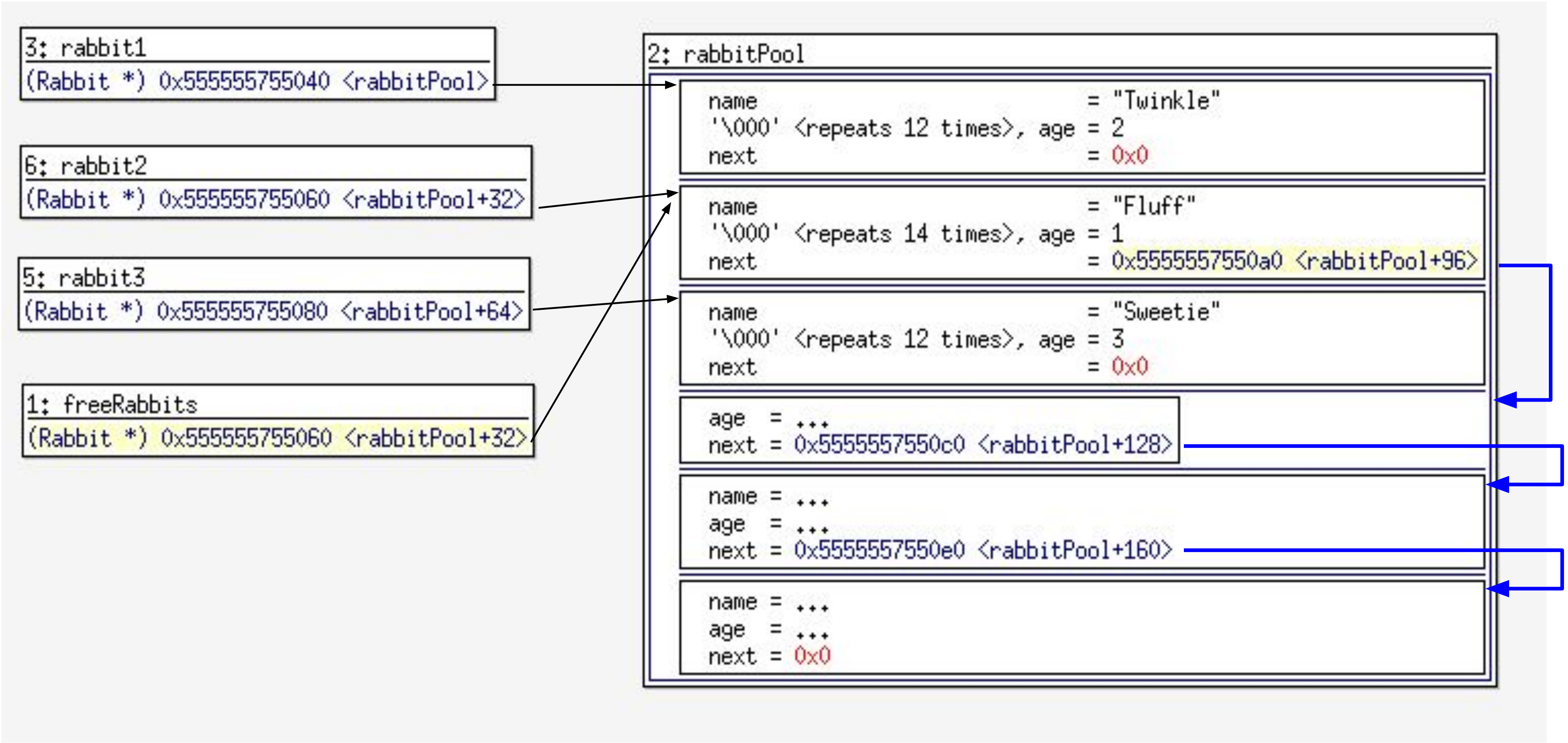
Rabbit Pool after initialise()



Rabbit Pool before freeRabbit call



Rabbit Pool after freeRabbit call



"Fluff" is back on the free list, so *rabbit2 should not be used.

Problems

- How big is *MAX_NUM_RABBITS?*
  - If we choose it too low, program crashes
  - If we choose it too high, program wastes memory
    - May lead to unnecessary thrashing
- If we have other dynamic items to deal with (fleas, trees, grass, ...) it becomes impracticable to find good pool-size constants for all.
- SO:
  1. Allocate all new entities from a single pool of available memory
     - *malloc* and *free* library function calls
  2. Grow that pool by calls to the operating system as necessary.
     - *brk*/*sbrk* system calls (hidden inside *malloc)*

A modified "student" module

```
typedef struct student_s Student;

struct student_s {
     char* name;
     int age;
     Student* next;
};

// Free the student struct,
// so it can be used
void freeOneStudent(Student* sp)
{
     free(sp->name);
     free(sp);
}
```

```
Student* newStudent(const char* name,
                         int age)
{
     Student* sp = NULL;
     int nameSize = 0;
     sp = malloc(sizeof(Student));
     if (sp != NULL) {
          // if we're not out of memory
          nameSize = strlen(name);
          sp->name = malloc(nameSize + 1);
          if (sp->name == NULL) {
               // We're out of memory
               free(sp); // Free stud struct
               sp = NULL; // Must return null
          } else {
               strncpy(sp->name, name,
                          nameSize + 1);
               sp->age = age;
               sp->next = NULL;
          }
     }
     return sp;
}
```

NB!

Points to note

- *malloc* allocates a chunk of heap memory of the specified size (in bytes).
  - Returns a (void*) pointer (NULL if out of memory)
    - Use it wisely!
- For strings, must *malloc* (length+1) bytes
- *free* returns a *malloc*'d chunk of memory to the heap free list
  - Freed memory is a boojum[1]
  - Looking at it will kill you.

[1] *boojum* is from Lewis Carroll's poem "The hunting of the snark".

How *not* to allocate dynamic memory

- The following seems like a much easier approach.
  - And it compiles, too, if you don't use `-Wall -Werror`
- But it's CATASTROPHIC (*twice over*).
- Why?

```
Student* newStudent(const char* name, int age) {
    Student stud;
    stud.age = age;
    stud.name = name; // Assume struct with char* name
    stud.next = NULL;
    return &stud;
}
```

Bad!
Bad!
Bad!
Bad!
Bad!
Bad!
Bad!
Bad!
Bad!
Bad!
Bad!

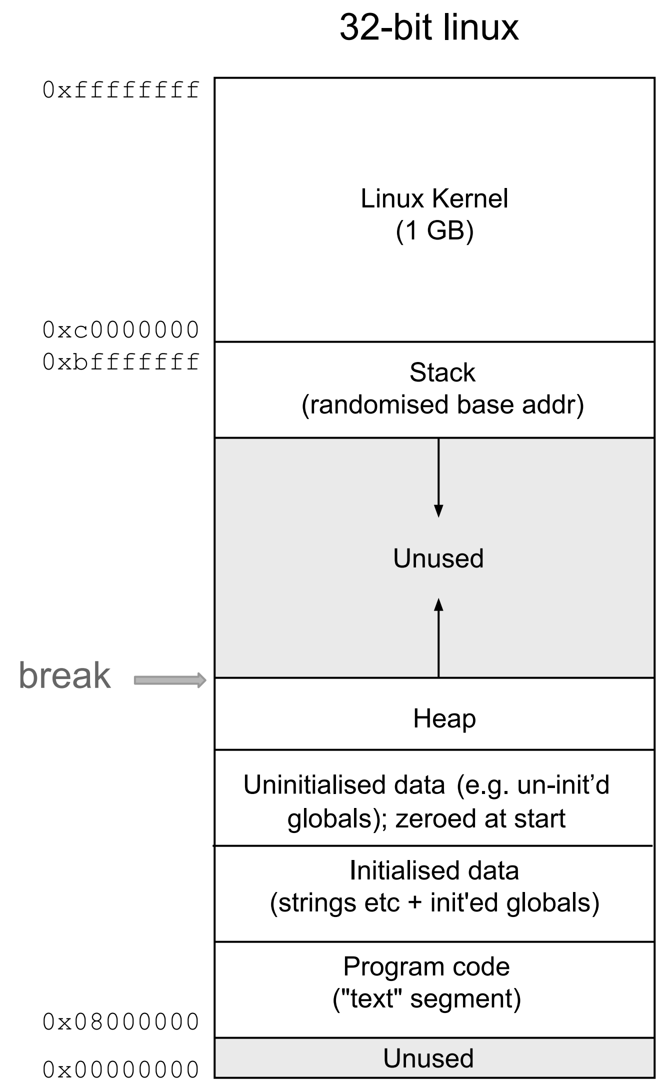
It's really really *IMPORTANT* that you know the answer to this (BOTH reasons).

Interlude

- Request for extension on SQ2. But:
  - It's only 5% (c.f. 20% for test)
  - It's already due 2 days after end of term
  - You shouldn't spend too much time on it. The test is much more important.
- Lab question drill quiz
- Practice test available by end of week
- SQ2/Q1 & Q2 - discuss
- **Use valgrind**!

How it works

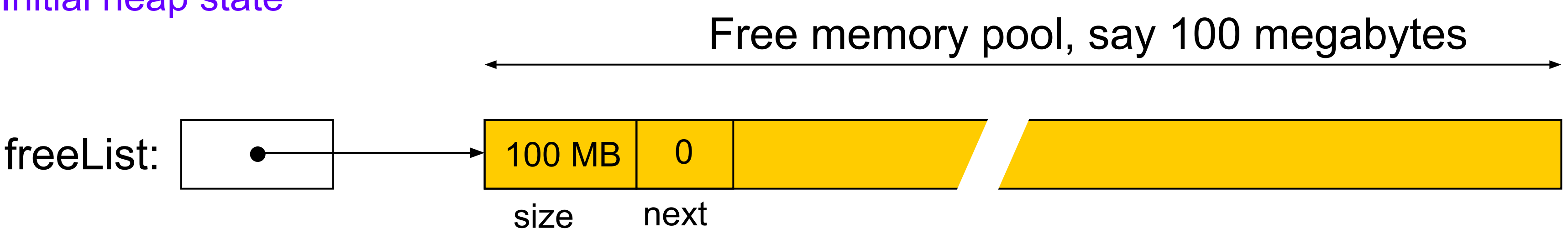
- *malloc* and *free* are the two main functions in the *memory allocator* module.
- They manage the "heap"
  - The free memory area above the initialised data segment
  - Essentially they maintain a "booking sheet" of (un)available memory
    - A global variable in the module
- The size of the heap is grown/shrunk by OS calls *brk* or *sbrk*
  - Type *man brk* to find out more



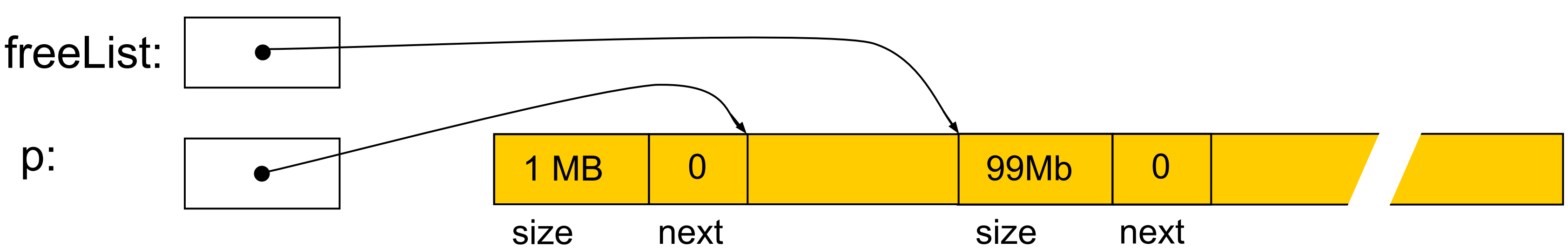
A simplistic view of malloc/free

For the full story see https://code.woboq.org/userspace/glibc/malloc/malloc.c.html

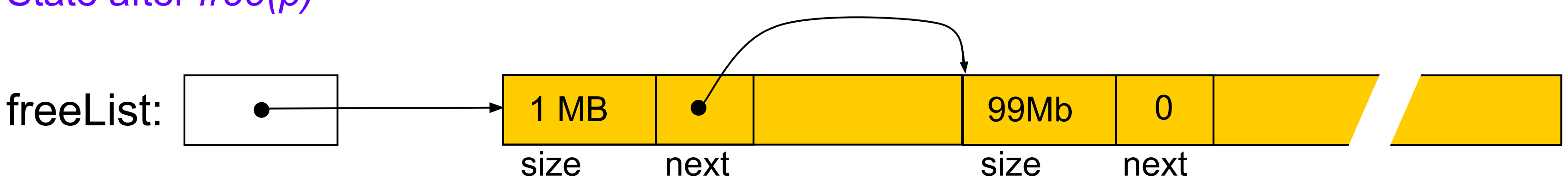
Initial heap state



State after *char *p = malloc(1000000); // 1 MB*



State after *free(p)*



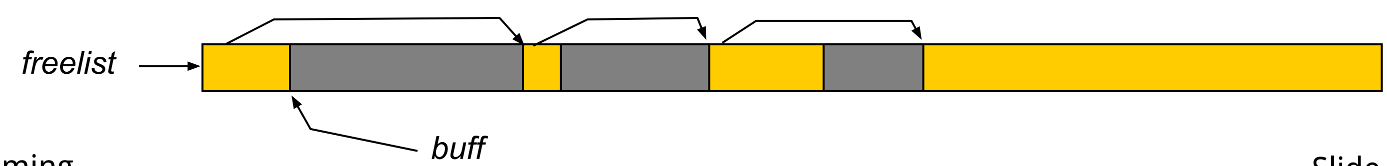
Growing malloc'd blocks

- Often need to increase size of an allocated block
- Use function `void* realloc(void* memPtr, int newSize)`
  - e.g. [Danger: bad code ahead!]

Ptr to a malloc'd block
If *null*, realloc = malloc

```
char* readLine(void)
{
     char* buff = NULL;
     int numBytes = 0;
     int c = 0;
     while ((c = getchar()) != EOF && c != '\n') {
          buff = realloc(buff, numBytes + 1);  // Get a new bigger block
          buff[numBytes++] = c;
       }
       if (buff != NULL) {
            buff[numBytes] = '\0';
       }
       return buff;  // NULL if no data read
}
```

NB: DON'T assume new block is at same place as old block!



BUT ...

There are at least three things wrong with that code:

1. It has a serious bug (possibly leading to a segment fault)
   - UDOO: FIX IT!
2. *realloc*-ing memory blocks is potentially expensive
   - may copy whole buffer if old one can't just be grown
   - doing it for every char is too inefficient for even me to tolerate
3. The specs of that function aren't defined at the start and probably aren't what you want!
   - What are they, i.e., *exactly* what does the function return? [Consider blank lines, unterminated last lines, empty files, ...]
   - Write a better function (see *fgets* for possible spec ideas)

- Also, should be checking for *null* return from *realloc*
  - But hard to recover from it.

A better realloc *strategy*

- Better to *realloc* only intermittently
- Allocate an initial buffer big enough for the average case
- Double the buffer size whenever space runs out.

```
char* buff = malloc(INITIAL_BUFF_SIZE);
int buffSize = INITIAL_BUFF_SIZE;
int numBytes = 0;
while (...) {
    if (numBytes >= buffSize - 1) {
        buffSize *= 2;  // Double the buffer size
        buff = realloc(buff, buffSize);
    }
    // etc
}
```

- Alternatively can increment by a fixed (largish) amount
  - But this can lead to $O(n^2)$ behaviour due to repeated buffer copying

Memory leaks

- Programs that don't free *malloc*'d memory have *memory leaks*
- Their memory footprint grows without bound
- To avoid this, every *malloc* should be matched to a corresponding *free*.
- Similarly, every call to a function like *newStudent* must have a matching call to the corresponding *freeStudent*.
- Need good clean multi-layered design
  - Match allocation and deallocation at each level

Example *(from Lab 6, simplified)*

```
for (i = 0; i < NUM_REPEATS; i++)
{
    studs = readStudents(inFile);
    printStudents(&studs );
    freeStudents(&studs );
    rewind(inputFile);
}
```

match

```
StudentList readStudents(FILE* fp)
{
    StudentList studs = {NULL, NULL};
    Student* sp = NULL;
    while ((sp=readStudent(fp))!=NULL)
    {
        addStudent(&studs, sp);
    };
    return studs;
}

void freeStudents(StudentList* studs)
{
    /* **** TBS **** */
}
```

must match

```
Student* readStudent(FILE* fp)
{
    // Read from file then call …
    // newStudent which does ...
    sp = malloc(sizeof(Student));
    buffSize = strlen(name) + 1;
    sp->name = malloc(buffSize);
    ...
    strncpy(sp->name, name, buffSize);
    sp->age = age;
    sp->next = NULL;
    return sp;
}

void freeOneStudent(Student* sp) {
    free(sp->name);
    free(sp);
}
```

match

match

Every line of code that allocates memory should have a corresponding line that deallocates it.

Detecting memory leaks

- Need a tool for the job. Lots available. e.g.:
  - *valgrind* – see *http://valgrind.org/*
    - Installed on our lab machines
    - Lots of capabilities
    - Simplest:

      ```
      valgrind --leak-check=yes processStudents studlist.txt
      ```
    - Large, slow, complex internally, but fairly easy to use
  - On-the-fly in-code checking with *mallinfo()*

Heap corruption

- Over-running a *malloc*'d buffer is fatal
  - Probably
  - Eventually
- Difficult to debug
  - Solution: don't bug! *Not-bugging* is easier than *de-bugging*!
- Again there are many tools to help you find heap corruptions
  - *valgrind* checks every heap memory reference (great tool for small projects but too slow and expensive for large projects)
  - Link program with *–lmcheck*
    - Uses versions of *malloc, free* that do some runtime checks
    - Aborts on error
      - But checks only when *malloc, free* called
    - Should always use this when developing code that uses *malloc/free*

Parting comments on dynamic mem

- ***Do*** understand that dynamic memory underpins all modern object-based languages
  - Objects, strings, lists etc are all allocated dynamically (i.e., on the heap)
- *BUT*
  - ***Don't*** use dynamic memory on small microcontrollers
    - Memory is too precious, and you can't afford the risk of a memory leak
  - ***Don't*** use dynamic memory on ultra-reliable systems
    - Too hard to prove correctness
    - See http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf