Last week

Data can be allocated on the stack
or on the heap (aka dynamic memory)

- Data on the stack is allocated automatically when we do a function call, and removed when we return

```
f() { ... int table[len]; .... }
```

- Data on the heap must be (de)allocated manually, using **`malloc`** and **`free`**

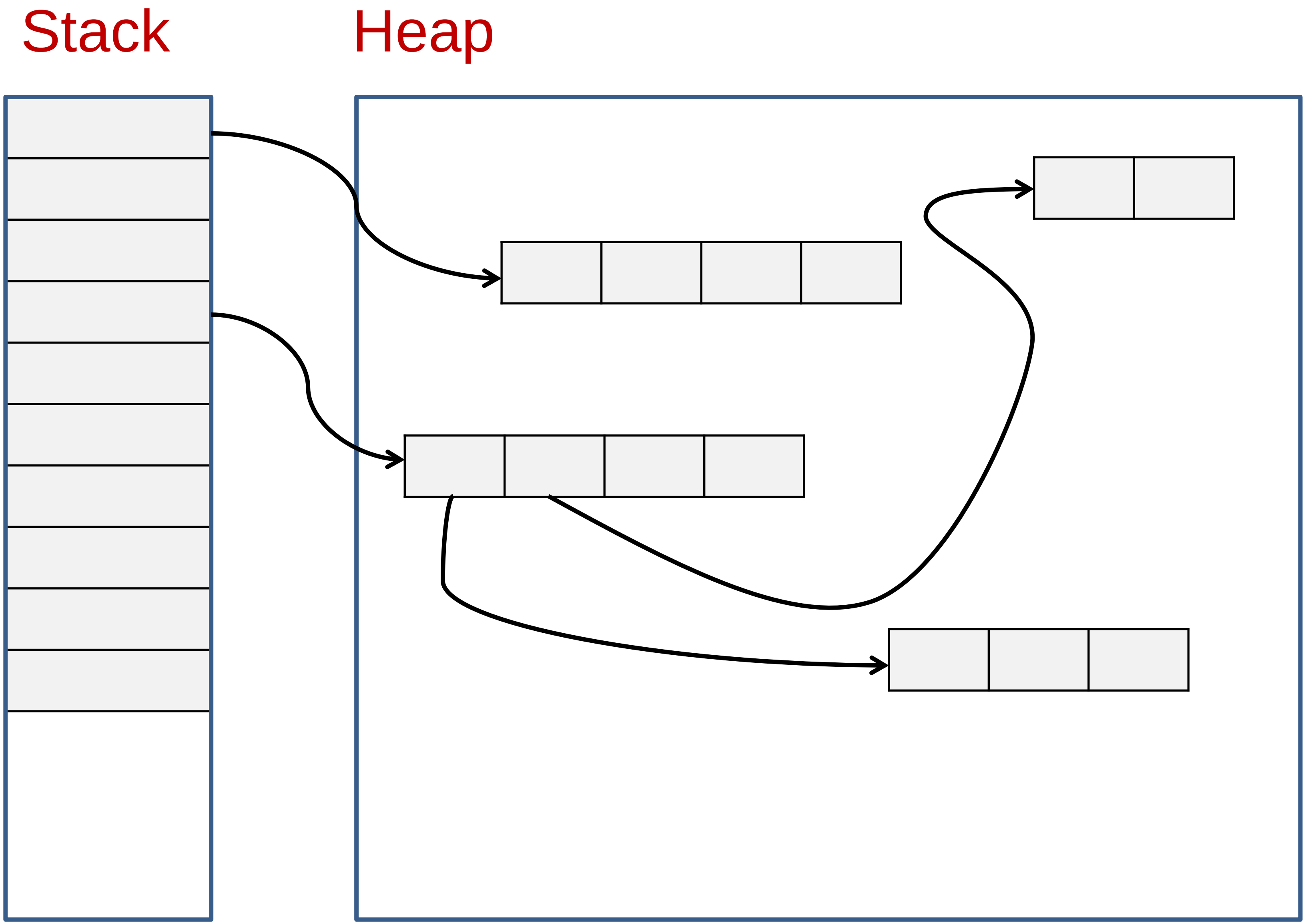
```
int *table = malloc(len*sizeof(int));
...
free(table);
```

Stack vs heap

- To use data on the heap, we *must* use pointers!
  - otherwise the data is lost and we cannot use it

- Pointers to data allocated on the heap can be
  - on the stack
  - in the heap itself

You can have pointers from the heap to the stack,
but typically you do not need them, or want them!


Stack
Heap

Memory (security) problems

Malicious code, buggy code, and insecure code can access data *anywhere* on the heap and stack, eg

- by doing pointer arithmetic
- by overrunning array bounds

More generally, security problems with memory can be due to

1. running out of memory
2. lack of initialisation of memory
3. bugs in program code

   esp for heap, as dynamic memory is more complex & error-prone

   Hence MISRA-C guidelines for safety-critical software include

   Rule 20.4 (REQUIRED) Dynamic heap allocation shall not be used

Running out of stack memory (aka stack overflow)

- Max size of the stack is finite and typically fixed on start-up of a process
- Normally, stack overflow will simply crash a program
  - as demo-ed last week

- *Are there sensible alternatives?*
- *Are there more dangerous alternatives?*

memory initialisation

What will this program print?

```
    char b;
    printf(”b is %i .\n”, b); // %i to print integer
```

In C memory is *not initialised*, so **`b`** can have any value.

Some programming language do provide a default initialisation.

Why is that nicer and more secure?

- programs behave more deterministic; a program with uninitialised variables can behave differently each time it’s run, which is not nice esp. when debugging
- for security: by reading uninitialised memory, a program can read confidential memory contents left there earlier

`calloc`

Memory allocated on the heap with **`malloc`** is typically not initialised

- Many OSs will zero out memory for a new process, but recycling of memory within that process means that malloc-ed memory may contain old junk.
- If OS does not zero out memory for new processes, you can access confidential information left in memory by other processes by malloc-ing large chunks of data!

The function **`calloc`** will initialise the memory it allocates, to all zeroes

- downside: this is slower
- upside: This is good for security and for avoiding accidential non-determinism due to missing initialisation in a (buggy) program
- But, in security-sensitive code, you may still want to zero out confidential information in memory yourself before you free it

Stack vs heap allocation

Consider `main() {while (true) { f(); } }`

Difference in behaviour for the two functions `f()` below?

```
void f(){
    int x[300]; x[0]=0;
    for (int i=1; i<300; i++) {x[i] = x[i-1]+i;}
    printf(“Result: %i \n”, x[299]);
  }
```

possible stack overflow

```
void f(){
    int *x = malloc(300*sizeof(int));
    x[0]=0;
    for (int i=1; i<300; i++) {x[i] = x[i-1]+i;}
    printf(“Result: %i \n”, x[299]);
  }
```

malloc may fail;
possible heap overflow

memory leak!
the memory for x
is not freed, so `main`
***will* crash when heap**
memory runs out

Stack vs heap allocation

Correct and *secure* version of function f that uses the heap:

```
void (f){
     int *x = malloc(300*sizeof(int));
     if (x==NULL) { exit(); }
     x[0]=0;
     for (int i=1; i<300; i++) {x[i] = x[i-1]+i;}
     printf(“result is %i \n”, x[299]);
     free(x); // to avoid memory leaks
  }
```

Moral of the story: heap allocation is more work for the programmer

Heap problems: memory leaks

Memory leaks occur when you forget to use free correctly.

Programs with memory leaks will crash if they run for long enough.

You may also notice programs running slower over time if they leak memory.
Restarting such a program will help, as it will start with a fresh heap

More heap problems: dangling pointers

Never use memory after it has been de-allocated

```
int *x = malloc (1000);
free (x);
...
print(“Let’s use a dangling pointer %s”, x);
```

A pointer to memory that has been de-allocated (freed) is called a dangling pointer. When using dangling pointers, all bets are off...

More heap problems: using free incorrectly

- Never free memory that is not dynamically allocated

```
char *x = ”hello”;
free (x); // error, since ”hello”
          // is statically allocated
```

- Never double free

```
char *x = malloc (1000);
free (x);
...
free (x); // error
```

Memory management trouble: spot the bug

```
int *x = malloc (1000);
int *y = malloc (2000);
y = x;
```

memory leak!
we cannot access the 2000 bytes that y pointed to, and we cannot free them!

Aliasing – spot the bug!

Aliasing can make some of these bugs hard to spot

```
int *x = malloc (1000);
int *y = malloc (2000);
int *z = x;
y = x;
int *w = y;
free (w);
free (z);
```

double free! **this memory is already de-allocated in the line above**

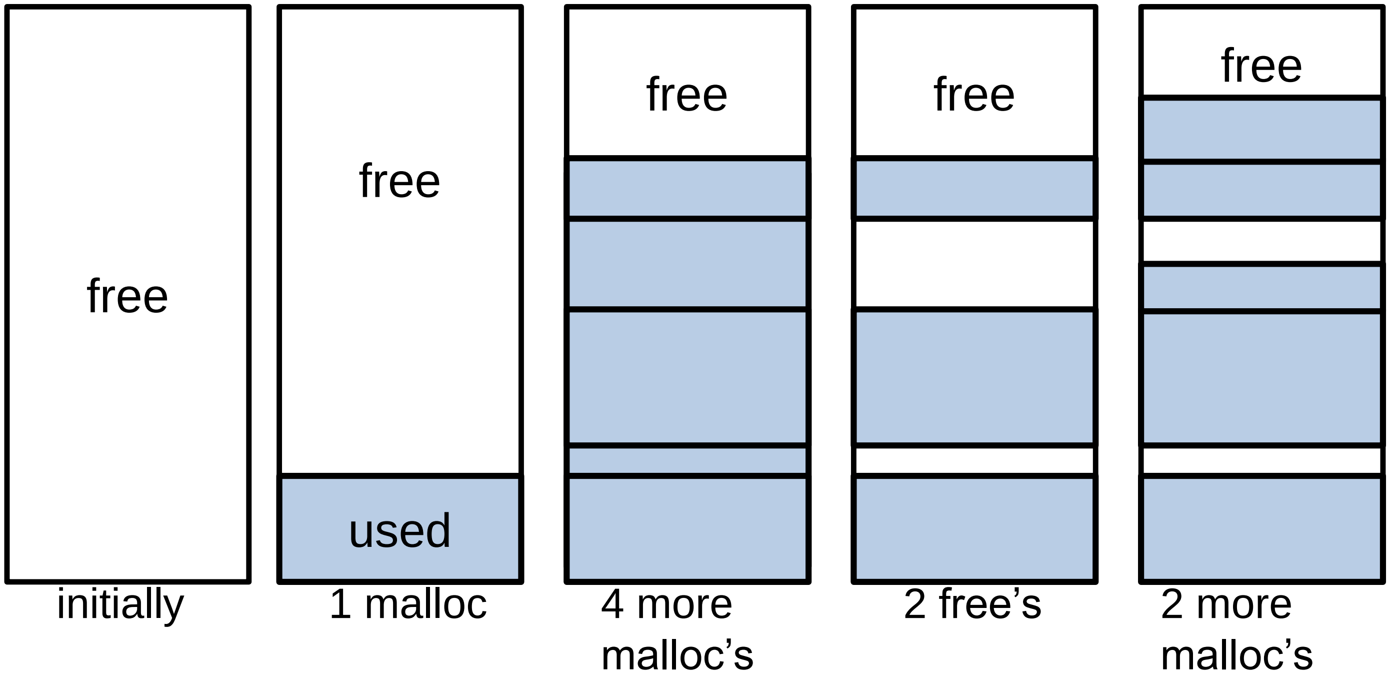
Two pointers are called aliases if they both point to the same address

Aliasing, together with the fact that **`malloc`** and **`free`** can happen in different places of the program, make dynamic memory management extremely tricky!!

Heap memory management

The implementations of **`malloc`** and **`free`** have to keep track of which parts of heap are still unused.

- Initially, the free memory is one contiguous region.
- As more blocks are malloc-ed and freed, it becomes messier

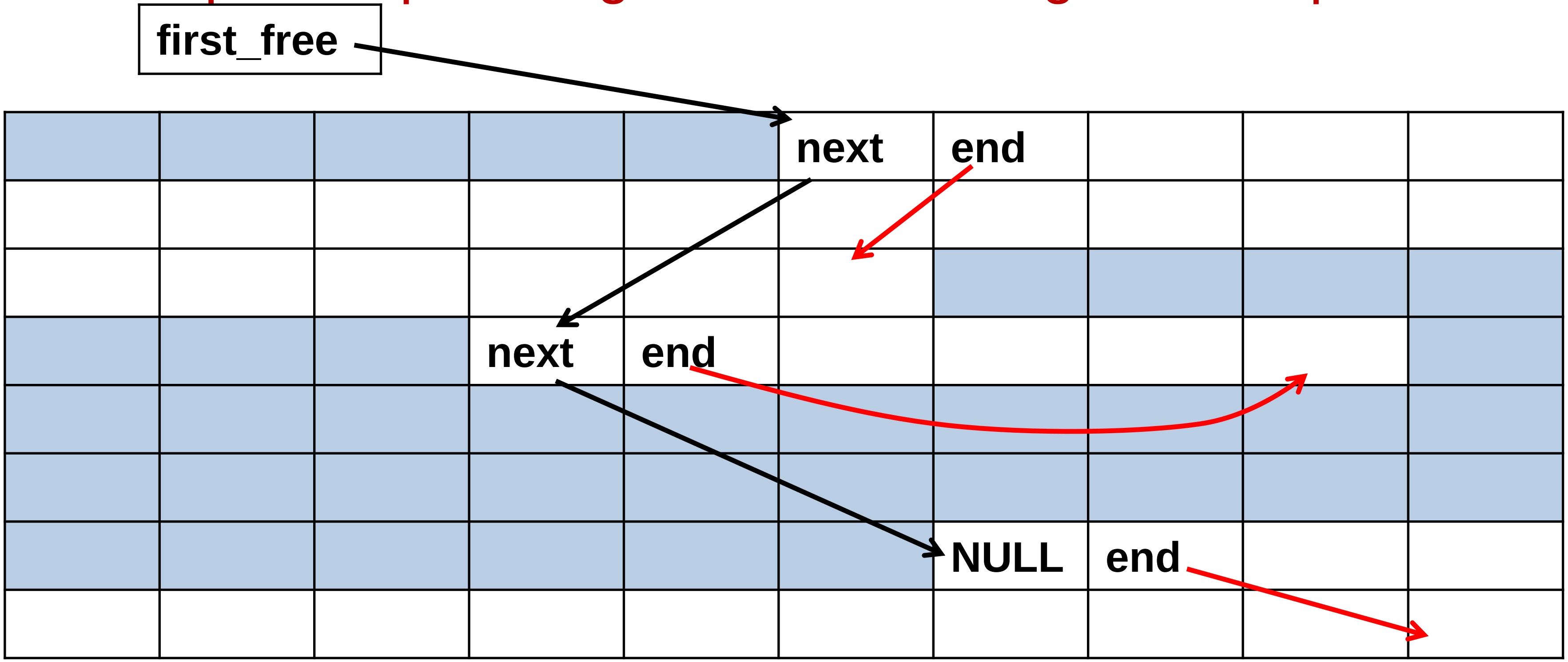


Heap memory management

The implementations of **`malloc`** and **`free`** have to keep track of which parts of heap are still unused.

How would you do this?

Example heap management: recording free heap chunks



Inside each free chunk

- a pointer to the next free chunk
- a pointer to the end of the current free chunk

Not very efficient: real **`malloc`** and **`free`** do more admin for efficiency

Heap memory management

One way is to maintain a free list of all the heap chunks that are unused.

- This info can be recorded on the heap itself, namely in the unused parts of the heap.
- You can also maintain meta-information *in* the used chunks on the heap to help in de-allocation (eg the size of the chunk)
- NB an attacker can try to corrupt any this data!

Padding malloc-ed data to a round number reduces fragmentation.

The programmer can make memory management easier and reduce fragmentation by often allocating chunks of data of the same size.

Malloc-ed data can *not* be moved or shifted on the heap, because this will break pointers to that data!

garbage collection

In modern programming languages (Java, C#, ...), instead of the programmer having to free dynamic memory, there is a garbage collector which automatically frees memory that is no longer used.

Advantage: much less error-prone

Disadvantage: performance

- Garbage collection is an expensive operation (it involves analysis of the entire heap), so garbage collection brings some overhead.
- Moreover, garbage collection may kick in at unexpected moments, temporarily resulting in a very bad response time.

  Still, there are clever garbage collection schemes suitable for real-time programs.

Recap: stack vs heap

Stack

- variables are allocated and de-allocated automatically
- allocation is much faster than for the heap
- data on the stack can be used without pointers
- data needs have to be known at compile time
- stack space may run out, eg. due to infinite recursions
- max size of the stack usually fixed by OS when program starts

Heap

- (de)allocation has to be done manually by the programmer;
  this is highly error-prone!
- allocation of heap memory slower than for stack memory
- to access data on the heap, you *must* use pointers;
  this is also error-prone!
- more flexible, and must to be used when data needs are not known at compile time
- heap space may run out too, but can grow during the life time

Lack of memory protection

Data is typically not initialised when allocated

- except static global variables, memory allocated by **`calloc`**, and possibly fresh heap memory allocated by **`malloc`** the first time it is used

Irrespective of whether we store data on the heap or the stack:

malicious code , *buggy code*, and *insecure* code

can access data *anywhere* on heap or stack, eg

- by doing pointer arithmetic
- by overrunning array bounds

Buggy or insecure code acting on malicious input supplied by an attacker can be used for malicious purposes.

Malicious, buggy or insecure code can be in libraries or in, say,
a browser plugin.

Sun tarball problem (1993)

Every tarball produced on Solaris 2.0 contained fragments of the password file **`/etc/password.`**

How did this happen?

– **`tar`** looked up user info directly prior to producing tarball:
  - password file was loaded in memory for this
  - this memory was then released
- then **`tar`** allocated memory for constructing the tarball
  - allocated memory was always the memory just released
  - memory not zeroed out on allocation...

Solution: replacing **`char *buf (char*)malloc(BUFSIZE)`**

by **`char *buf (char*)calloc(BUFSIZE)`**