Vediamo l’allocazione dinamica della memoria attraverso il segmento Heap. Lo abbiamo già citato ma spiegheremo adesso a cosa serve. L’idea è di avere un meccanismo di allocazione e disallocazione dinamica della memoria (contrariamente a quella statica): permette in qualsiasi momento di aggiungere e togliere memoria in qualsiasi momento. Lo Stack anch’esso può allocare memoria “dinamicamente”, ma ciò in realtà è già previsto a tempo di compilazione ed è puramente associato alla chiamata di funzione (cosa che talvolta si vuole evitare). Con l’Heap la gestione della memoria è completamente affidata al programmatore (che dovrà allocarla e rilasciarla).

Sono state definite, a questo scopo, delle API; quella che prenderemo in considerazione è quella dello standard POSIX. Al suo interno vi sono diverse funzioni chiamabili da un programma C che si occupano appunto della gestione del segmento Heap.

Per l’allocazione di memoria, essa avviene tramite la funzione malloc(n), che prende come parametro un = il numero di byte da allocare e che restituisce un puntatore (quindi p = malloc(50000) allocherà 50000 byte e assegnerà a p l’indirizzo base di questa area di memoria).

Per rilasciare la memoria si chiama la funzione free(p), dove p è il puntatore all’area dati precedentemente allocata con malloc.  
Bisogna però tenere conto di alcuni dettagli. Innanzitutto bisogna capire quanti byte dobbiamo richiedere per allocare lo spazio necessario a una struttura dati. Normalmente nei linguaggi di alto livello la quantità di byte non è stabilita esplicitamente, per questo motivo esiste un’altra funzione, sizeof(x) che prende in input una variabile e restituisce come risultato il numero di byte che il sistema utilizza per memorizzare quella variabile in memoria RAM.   
Per esempio, n = sizeof(int) ci restituisce il numero di byte che occupano gli interi.

Nel linguaggio C abbiamo più tipi base:

* Char
* Int
* Double
* Float
* Short
* Long
* Long long
* Ecc.

Sono tutti codici rappresentati a lunghezza fissa, ma cambiano il numero di byte (e non bit) usati in RAM per memorizzarli.

Per semplificare le cose è stato definito nello standard che i char siano rappresentati su un singolo byte, quindi sizeof(char) è sempre = 1 su tutte le macchine. Gli altri valori possono cambiare: sizeof(int) può avere, tipicamente, due risultati diversi, su alcune macchine il risultato sarà 4, su altre sarà 8 (in generale gli int corrispondono a una PAROLA di memoria, che cambia da 4 a 8 byte a seconda che il processore sia da 32 o da 64 bit).

Ovviamente come parametro nella malloc si possono mettere valori molto superiori rispetto a quelli dei tipi base. Allocare grandi quantità di memoria serve infatti, di solito, per allocare un array di variabili. Se noi pensiamo a un allocazione di array, può essere però più conveniente chiamare la funzione calloc(n, size): essa prende due parametri, il numero di elementi e il sizeof di quel tipo di elementi. Per allocare un array di dieci interi sarebbe corretto chiamare calloc(10, sizeof(int)). Gli array si possono allocare tranquillamente anche con malloc, basta dare come parametro attuale il prodotto tra il numero di elementi e la sizeof di ciascun elemento (se però il prodotto degli elementi andasse in overflow calloc ne terrebbe conto e solleverebbe errore, mentre malloc ci lascerebbe a noistessi).

Sia malloc che calloc restituiscono quindi un puntatore, che contiene un indirizzo base.

Quando chiamiamo una di queste funzioni, viene allocato nel nostro Heap un pezzo di memoria corrispondente a quanto abbiamo richiesto. Il puntatore che viene ritornato è l’indirizzo della prima parola: quindi quando si dereferenzia il puntatore (\*p) in pratica si accede alla prima cella (parola) di memoria tra quelle allocate. Se si pensa a quest’area come un array, il nostro puntatore sta puntando al primo elemento di tale array (\*p == p[0]).  
Nel primo laboratorio avevamo allocato gli array staticamente, ma l’idea dei puntatori funzionava allo stesso modo.

Se si volesse accedere al secondo elemento dell’array allocato dinamicamente si potrebbe usare sia la notazione \*(p+1) che p[1].

In C, può succedere che dopo aver allocato 10 elementi io provi ad accedere a p[20] (o \*p+20): questo fa uscire dall’area di memoria che è stata allocata dal sistema operativo. Quello che può succedere dipende da come è stato implementato internamente il segmento heap: se fosse implementato in maniera perfetta, esso dovrebbe contenere solo l’area di memoria allocata, quindi accedere a un elemento fuori dall’area allocata dovrebbe sollevare una trap, causando l’interruzione del programma tramite segmentation fault. In realtà non funziona esattamente così, perché tutti gli array allocati finiscono tutti nello stesso segmento, quindi se dopo aver allocato 80 byte, io ne alloco altri 150 gli 80 byte continuano a far parte del segmento heap dedicato al nostro programma: questo non provoca segmentation fault, ma è un errore forse anche più grave perché è più difficile da notare ma provoca un assegnazione di valori di nonsenso alle nostre variabili.

Vediamo adesso l’aritmetica dei puntatori.

Se si alloca un array di interi in memoria heap, il puntatore p dovrà essere dichiarato come puntatore ad intero: ciò permetterà al compilatore di calcolare p+2 in modo corretto, nel caso p puntasse a un array. Infatti, il +1/+2 ecc. cambia nella compilazione a seconda del tipo (e quindi del sizeof del tipo) puntato dal puntatore. Per esempio, se ho char \* c, c+=2 fa puntare c a 2 byte dopo. Se ho int \* i, i+=2 fa puntare i a 2\*sizeof(int) (quindi 8 o 16) byte dopo. C’è quindi un misto di basso e alto livello quando si tratta di allocare e deallocare memoria e di gestire i puntatori.

Oltre a tutti i tipi base, usati per le variabili, il linguaggio C prevede anche il puntatore a void (void \* p). Esso è un puntatore che non si sa a cosa punta (ciò è voluto): In questo caso il compilatore non sarà in grado di calcolare l’aritmetica dei puntatori (e non saprà indirizzare gli elementi di un array). Questo serve per poter dare un tipo al puntatore che ritorna Malloc. Questo perché malloc e calloc restituiscono un puntatore, ma essi possono restituirlo per qualsiasi tipo di dato: se restituisse int\* saremmo costretti a usarlo solo per gli interi, ad esempio. Per questo motivo non ha senso attribuire un tipo diverso da void al puntatore ritornato da malloc e da calloc.  
Se noi vogliamo che un area di memoria sia interpretata come un array di int è necessario che questa informazione in più sia data al compilatore: ciò avviene grazie al meccanismo dei cast.

Quindi si scrive int \* p = (int\*) calloc(n\_elem, size); //così il compilatore potrà calcolare p+1, p+2, ecc.

Il void \* quindi è un tipo fasullo, usato espressamente per questo tipo di allocazione dinamica della memoria.

Vediamo anche realloc(p, n). Il primo parametro, p, è un puntatore, il secondo è una dimensione. Restituisce come risultato un altro puntatore. L’obiettivo di questa funzione è di modificare la quantità di byte allocati precedentemente per una certo dato/struttura dati. Quindi si può fare p = malloc(10) e poi fare p = realloc(p, 100). Questo farà sì che un’area allocata di 10 bytes passerà a 100 bytes (in pratica dice “cambia l’area dedicata a questo puntatore”). Il secondo parametro, n, può essere sia più grande che più piccolo rispetto alla dimensione iniziale.

Se ho un array di 2 int, creato con p = malloc(2\*sizeof(int)), si può aggiungere lo spazio per un terzo elemento con p = realloc(p,3\*sizeof(int)).  
Si può richiamare realloc tutte le volte che si vuole.

Un caso particolare della chiamata di realloc è quando il puntatore non è un indirizzo di memoria nello heap, ma è una costante NULL. In questo caso realloc funziona esattamente come malloc (p = realloc(NULL, 10) = malloc(10)). Quando, invece si mette un puntatore che puntava a un’area allocata e si inserisce 0 nel nuovo numero di byte realloc si comporta esattamente come free (realloc(p,0) = free(p)).

Quindi realloc è in grado, da sola, di svolgere tutte le mansioni di gestione della memoria dinamica (funziona come malloc, come realloc e come free).

Quando il puntatore passato come parametro dovrebbe essere la stessa variabile in cui si salva il risultato di relloc?  
In generale ha senso che sia uguale quando si vuole diminuire leggermente la dimensione, perché magari il sistema non sa che farsene della memoria disallocata e quindi restituisce lo stesso puntatore di prima.  
Se stiamo però espandendo l’area allocata e l’area immediatamente successiva è occupata da qualcos’altro restituirà un puntatore a una posizione diversa (per evitare la sovrascrittura). Passare però un puntatore diverso rischia di essere scomodo per il programmatore, perché c’è il rischio che i byte scritti in precedenza non vengano persi (se si scrive su 80 byte e poi se ne riallocano 100 non si vuole che si perda quello che era scritto in quegli 80).

Per questo motivo la realloc fa sì che quando si rialloca la memoria, copia anche i valori contenuti nei byte precedenti per quanto possibile (se m1 e m2 sono le dimensioni delle memorie, se m1 < m2 allora i valori in m1 vengono salvati in m2 interamente, viceversa si salvano in m2 tutti i valori che stanno lì).

Anche per realloc bisogna effettuare il cast.

Alcuni dettagli dal punto di vista implementativo.

Discorso sull’allineamento delle variabili.

Abbiamo visto che nel processore Amber 23 le parole di memoria sono da 4 byte. Il processore può accedere a un solo byte oppure a un’intera parola, ciò varia a seconda dell’istruzione; un processore può accedere alla memoria in diversi modi (alcuni potrebbero permettere di accede a variabili di 2 byte, altri di 16 byte ecc.). A livello implementativo, per tenere conto di tutte queste particolarità è quello di determinare la massima dimensione di una variabile su quel tipo di processore.

Per esempio in una architettura a 32 bit, int è rappresentato su 4 byte e long long su 8, mentre in una architettura a 64 bit int è rapprentato su 8 byte e long long su 16.   
Ogni macchina avrà incrementi delle celle di memorie (da parola a parola) che saranno = alla dimensione della parola (ad esempio quando si incrementa il program counter).  
Tutto ciò serve a determinare quali indirizzi possono essere ritornati da malloc e realloc. L’ipotesi più conservativa fatta dal sistema operativo è che la memoria allocata possa essere utilizzata per salvare dei long long, quindi deve essere possibile passare di 8 indirizzi in 8. Quindi in una macchina a 32 bit malloc restituirà sempre un indirizzo multiplo di 8, mentre su una macchina a 64 bit malloc restituirà indirizzi multipli di 16. Questo è chiamato “allineamento”. Se infatti Malloc ritornasse come indirizzo un numero dispari si potrebbe effettuare il cast solo ad un char (perché una cella di memoria di indirizzo dispari non può corrispondere un int, long ecc.).

Se si chiama malloc e si richiede di allocare un’area di memoria troppo grande si genera una situazione di errore: il sistema dice “non posso, non c’è abbastanza memoria” e il modo in cui lo fa è restituire il puntatore NULL, che viene codificato con l’indirizzo 0, non valido nel segmento heap. Ciò può avvenire sia per richieste assurde, ma anche in situazioni normali in cui però la RAM è piena, quindi prima di utilizzare il puntatore restituito, bisogna controllare che il valore ritornato non sia NULL (e se è stato ritornato NULL si termina il programma con un messaggio di errore, il motivo per cui dobbiamo essere noi a fermare il programma è perché in genere questo non è un bug, ma un problema del sistema).

L’esercizio di questa settimana.

L’archivio è già completo e sarà utilizzato per le prossime 3 settimane. Questo primo esercizio ha uno sforzo minimo. L’unica cosa da fare è mandare in esecuzione un programma e vedere cosa manda in uscita (e ovviamente leggere il sorgente per capire cosa fa).

Il sorgente fa delle cose apparentemente banale, ma usa in modo bizzarro i puntatori che vengono ritornati. Una volta ottenuti i puntatori si vanno a vedere i contenuti della memoria a quell’indirizzo allocato ma anche a un po’ di indirizzi prima e un po’ di indirizzi dopo: questo non dovrebbe essere consentito, è infatti un errore nella normale programmazione. In questo caso tale errore è fatto volutamente con l’obiettivo di capire come funziona il meccanismo di allocazione della memoria.

C’è un problema concettuale nell’interfaccia POSIX, quando si alloca la memoria viene specificata la dimensione (e viene ritornato il puntatore all’area allocata), però gli indirizzi sono sempre ALLINEATI (su una macchina da 32 bit sono allineati sui multipli di 8). Quindi se si chiede di allocare 3 byte, succede che la struttura avrà una sua certa dimensione, verrà ritornato come indirizzo un multiplo di 8 (es. 800) e se si chiede di allocare un altro byte verrà ritornato il byte multiplo di 8 successivo come indirizzo (808) quindi in questo caso non si stanno occupando solo 4 byte, ma 9 (perché i 5 tra 803 e 808 non sono utilizzabili).   
Si noti infatti che alla free si passa solo un puntatore come parametro, ma il sistema deve sapere anche dove termina l’area di memoria allocata per poterla liberare. Il sistema per poter liberare quella memoria nella quantità giusta se lo dovrà scrivere da qualche parte, che deve essere in grado di ritrovare conoscendo soltanto il puntatore. Lo stesso problema sorge con la realloc (quando si cambia la posizione della memoria allocata).

Il nostro laboratorio è quindi un piccolo esempio di reverse engineering, facendo delle operazioni errate si cerca di capire come è fatto all’interno il sistema di allocazione della memoria. I risultati cambiano da macchina a macchina, perché stiamo andando a fare cose sbagliate per lo standard.

Normalmente la memoria è gestita su due livelli, uno che richiede l’intervento del sistema operativo, attraverso system call e un altro che invece non lo richiede (perché le system call sono costose). Quindi le nostre funzioni di libreria standard non chiamano sempre system call. Infatti all’avvio del programma un’area di memoria HEAP viene allocata automaticamente (ma non è utilizzabile), con i malloc e i realloc si può utilizzare questa porzione di heap e se si raggiunge la dimensione massima a quel punto viene effettuata la system call che alloca un altro blocco di heap (funziona in maniera simile al sistema size/capacity dei vector, così come si evita la riallocazione tramite una capacity > size, si evitano continue chiamate di system call fornendo un heap allocato allocabile già all’avvio del programma).

Ci sono diversi modi in cui la free può trovare la dimensione dell’area da deallocare. Un’opzione sarebbe avere una memoria associativa che collega puntatore a area di memoria. Un’alternativa sarebbe di inserire la dimensione da qualche parte relativa alla posizione del puntatore (quindi a puntatore +- x).