# Week 04: Dynamic Data Structures

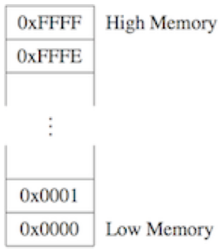
## Memory

Reminder:

Computer memory … large array of consecutive data cells or bytes

- `char` … 1 byte
- `int,float` … 4 bytes
- `double` … 8 bytes
- *`any_type *`* … 4 bytes (on CSE lab computers)

Memory addresses shown in Hexadecimal notation

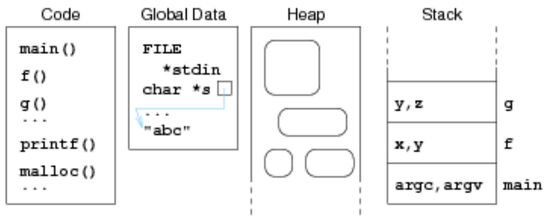


## C execution: Memory

An executing C program partitions memory into:

- *code* … fixed-size, read-only region
  - contains the machine code instructions for the program
- *global data* .. fixed-size, read-write region
  - contain global variables and constant strings
- *heap* … very large, read-write region
  - contains dynamic data structures created by `malloc()` (see later)
- *stack* … dynamically-allocated data (function local vars)
  - consists of frames, one for each currently active function
  - each frame contains local variables and house-keeping info

## ... C execution: Memory



## Exercise #1: Memory Regions

```
int numbers[] = { 40, 20, 30 };

void insertionSort(int array[], int n) {
   int i, j;
   for (i = 1; i < n; i++) {
      int element = array[i];
         while (j >= 0 && array[j] > element) {
         array[j+1] = array[j];
         j--;
      }
      array[j+1] = element;
   }
}

int main(void) {
   insertionSort(numbers, 3);
   return 0;
}
```

Which memory region are the following objects located in?

1. `insertionSort()`
2. `numbers[0]`
3. `n`
4. `array[0]`
5. `element`

1. code
2. global
3. stack
4. global
5. stack

## Dynamic Data Structures

## Dynamic Memory Allocation

So far, we have considered *static* memory allocation

- all objects completely defined at compile-time
- sizes of all objects are known to compiler

Examples:

```
int   x;     // 4 bytes containing a 32-bit integer value
char *cp;    // 4 bytes (on CSE machines)
             //   containing address of a char
typedef struct {float x; float y;} Point;
Point p;     // 8 bytes containing two 32-bit float values
char  s[20]; // array containing space for 20 1-byte chars
```

## ... Dynamic Memory Allocation

In many applications, fixed-size data is ok.

In many other applications, we need flexibility.

Examples:

```
char name[MAXNAME];    // how long is a name?
char item[MAXITEMS];  // how high can the stack grow?
char dictionary[MAXWORDS][MAXWORDLENGTH];
                     // how many words are there?
                     // how long is each word?
```

With fixed-size data, we need to guess sizes ("large enough").

---

## ... Dynamic Memory Allocation

Fixed-size memory allocation:

- allocate as much space as we might ever possibly need

Dynamic memory allocation:

- allocate as much space as we actually need
- determine size based on inputs

But how to do this in C?

- all data allocation methods so far are "static"
  - however, stack data (when calling a function) is created dynamically (size is known)

---

## Dynamic Data Example

Problem:

- read integer data from standard input (keyboard)
- first number tells how many numbers follow
- rest of numbers are read into a vector
- subsequent computation uses vector (e.g. sorts it)

Example input: `6 25 -1 999 42 -16 64`

How to define the vector?

---

## ... Dynamic Data Example

Suggestion #1: allocate a large vector; use only part of it

```
#define MAXELEMS 1000

// how many elements in the vector
int numberOfElems;
scanf("%d", &numberOfElems);
assert(numberOfElems <= MAXELEMS);

// declare vector and fill with user input
int i, vector[MAXELEMS];
for (i = 0; i < numberOfElems; i++)
   scanf("%d", &vector[i]);
```

Works ok, unless too many numbers; usually wastes space.

Recall that `assert()` terminates program with standard error message if test fails.

---

## ... Dynamic Data Example

Suggestion #2: create vector after count read in

```
#include <stdlib.h>

// how many elements in the vector
int numberOfElems;
scanf("%d", &numberOfElems);

// declare vector and fill with user input
int i, *vector;
size_t numberOfBytes;
numberOfBytes = numberOfElems * sizeof(int);

vector = malloc(numberOfBytes);
assert(vector != NULL);

for (i = 0; i < numberOfElems; i++)
   scanf("%d", &vector[i]);
```

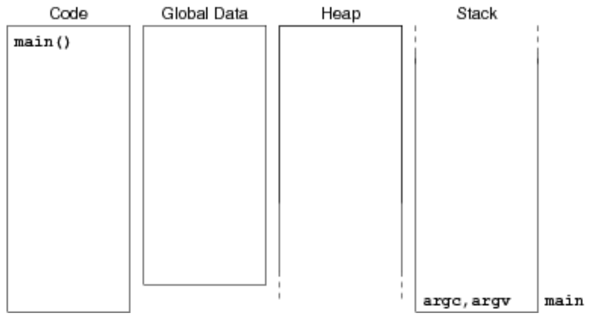
Works unless the *heap* is already full (very unlikely)

Reminder: because of pointer/array connection `&vector[i] == vector+i`

---

## The `malloc()` function

Recall memory usage within C programs:



## ... The `malloc()` function

`malloc()` function interface

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

What the function does:

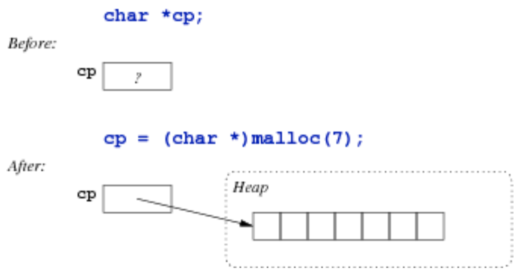
- attempts to reserve a block of `n` bytes in the *heap*
- returns the address of the start of this block
- if insufficient space left in the heap, returns `NULL`

Note: `size_t` is essentially an `unsigned int`

- but has specialised interpretation of applying to memory sizes measured in bytes

## ... The `malloc()` function

Example use of `malloc`:



## ... The `malloc()` function

Things to note about `void *malloc(size_t)`:

- it is defined as part of `stdlib.h`
- its parameter is a size in units of *bytes*
- its return value is a *generic* pointer `(void *)`
- the return value must *always* be checked (may be `NULL`)

Required size is determined by *#Elements* `*` `sizeof(`*ElementType*`)`

## Exercise #2: Dynamic Memory Allocation

Write code to

1. create space for 1,000 speeding tickets (cf. Lecture Week 1)
2. create a dynamic $m \times n$-matrix of floating point numbers, given $m$ and $n$

How many bytes need to be reserved in each case?

1. Speeding tickets:

```
typedef struct {
        int day, month, year; } DateT;
typedef struct {
        int hour, minute; } TimeT;
typedef struct {
        char plate[7]; DateT d; TimeT t; } TicketT;
```

```
TicketT *tickets = malloc(1000 * sizeof(TicketT));
assert(tickets != NULL);
```

28,000 bytes allocated

2. Matrix:

```
float **matrix = malloc(m * sizeof(float *));
assert(matrix != NULL);
int i;
for (i = 0; i < m; i++) {
   matrix[i] = malloc(n * sizeof(float));
   assert(matrix[i] != NULL);
}
```

$4m + 4 \cdot mn$ bytes allocated

## Exercise #3: Memory Regions

Which memory region is `tickets` located in? What about `*tickets`?

1. `tickets` is a variable located in the stack
2. `*tickets` is in the heap (after **`malloc`**'ing memory)

## ... The `malloc()` function

`malloc()` returns a pointer to a data object of some kind.

Things to note about objects allocated by `malloc()`:

- they exist until explicitly removed (program-controlled lifetime)
- they are *accessible* while some variable references them
- if no active variable references an object, it is *garbage*

The function `free()` releases objects allocated by `malloc()`

## ... The `malloc()` function

Usage of `malloc()` should always be guarded:

```
int *vector, length, i;
…
vector = malloc(length*sizeof(int));
// but malloc() might fail to allocate
assert(vector != NULL);
// now we know it's safe to use vector[]
for (i = 0; i < length; i++) {
        … vector[i] …
}
```

Alternatively:

```
int *vector, length, i;
…
vector = malloc(length*sizeof(int));
// but malloc() might fail to allocate
if (vector == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
}
// now we know its safe to use vector[]
for (i = 0; i < length; i++) {
        … vector[i] …
}
```

- `fprintf(stderr, …)` outputs text to a stream called **`stderr`** (the screen, by default)
- `exit(v)` terminates the program with return value *v*

## Memory Management

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

- releases a block of memory allocated by `malloc()`
- `*ptr` is a dynamically allocated object
- if `*ptr` was not `malloc()`'d, chaos will follow

Things to note:

- the contents of the memory block are not changed
- all pointers to the block still exist, but are not valid
- the memory may be re-used as soon as it is `free()`'d

## ... Memory Management

**Warning! Warning! Warning! Warning!**

Careless use of `malloc()` / `free()` / pointers

- can mess up the data in the heap
- so that later `malloc()` or `free()` cause run-time errors
- possibly well after the original error occurred

Such errors are very difficult to track down and debug.

Must be very careful with your use of `malloc()` / `free()` / pointers.

## ... Memory Management

If an uninitialised or otherwise invalid pointer is used, or an array is accessed with a negative or out-of-bounds index, one of a number of things might happen:

- program aborts immediately with a "`segmentation fault`"
- a mysterious failure much later in the execution of the program
- incorrect results, but no obvious failure
- correct results, but maybe not always, and maybe not when executed on another day, or another machine

The first is the most desirable, but cannot be relied on.

## ... Memory Management

Given a pointer variable:

- you can check whether its value is `NULL`
- you can (maybe) check that it is an address
- you **cannot** check whether it is a valid address

## ... Memory Management

Typical usage pattern for dynamically allocated objects:

```
// single dynamic object e.g. struct
Type *ptr = malloc(sizeof(Type));
assert(ptr != NULL);
… use object referenced by ptr e.g. ptr->name …
```

```
free(ptr);

// dynamic array with "nelems" elements
int nelems = NumberOfElements;
ElemType *arr = malloc(nelems*sizeof(ElemType));
assert(arr != NULL);
… use array referenced by arr e.g. arr[4] …
free(arr);
```

## Memory Leaks

Well-behaved programs do the following:

- allocate a new object via `malloc()`
- use the object for as long as needed
- `free()` the object when no longer needed

A program which does not `free()` each object before the last reference to it is lost contains a *memory leak*.

Such programs may eventually exhaust available heapspace.

### Exercise #4: Dynamic Arrays

Write a C-program that

- prompts the user to input a positive number $n$
- allocates memory for two $n$-dimensional floating point vectors **a** and **b**
- prompts the user to input $2n$ numbers to initialise these vectors
- computes and outputs the inner product of **a** and **b**
- frees the allocated memory

## Sidetrack: Standard I/O Streams, Redirects

Standard file streams:

- **`stdin`** … standard input, by default: keyboard
- **`stdout`** … standard output, by default: screen
- **`stderr`** … standard error, by default: screen

- `fprintf(stdout, …)` has the same effect as `printf(…)`
- `fprintf(stderr, …)` often used to print error messages

Executing a C program causes `main(…)` to be invoked

- with `stdin`, `stdout`, `stderr` already open for use

### ... Sidetrack: Standard I/O Streams, Redirects

The streams `stdin`, `stdout`, `stderr` can be *redirected*

- redirecting `stdin`

  ```
  prompt$ myprog < input.data
  ```

- redirecting `stdout`

  ```
  prompt$ myprog > output.data
  ```

- redirecting `stderr`

  ```
  prompt$ myprog 2> error.data
  ```

## Abstract Data Structures: ADTs

## Abstract Data Types

Reminder: An *abstract data type* is …

- an approach to implementing data types
- separates *interface* from *implementation*
- users of the ADT see only the interface
- builders of the ADT provide an implementation

E.g. does a client want/need to know how a Stack is implemented?

- ADO = *a*bstract *d*ata *o*bject (e.g. a single stack)
- ADT = *a*bstract *d*ata *t*ype (e.g. stack data type)

### ... Abstract Data Types

ADT *interface* provides

- an *opaque* user-view of the data structure (e.g. `stack *`)
- function signatures (prototypes) for all operations
- semantics of operations (via documentation)
- a contract between ADT and its clients

ADT *implementation* gives

- concrete definition of the data structure
- function implementations for all operations
- … including for *creation* and *destruction* of instances of the data structure

ADTs are important because …

- facilitate decomposition of complex programs
- make implementation changes invisible to clients
- improve readability and structuring of software

## Stack as ADT

Interface (in `stack.h`)

```
// provides an opaque view of ADT
typedef struct StackRep *stack;

// set up empty stack
stack newStack();
// remove unwanted stack
void dropStack(stack);
// check whether stack is empty
int StackIsEmpty(stack);
// insert an int on top of stack
void StackPush(stack, int);
// remove int from top of stack
int StackPop(stack);
```

ADT ***stack*** defined as a *pointer* to an *unspecified* struct named `StackRep`

## Sidetrack: Defining Structures

Structures can be defined in two different styles:

```
typedef struct { int day, month, year; } DateT;
// which would be used as
DateT somedate;

// or

struct date { int day, month, year; };
// which would be used as
struct date anotherdate;
```

The definitions produce objects with identical structures.

It is possible to combine both styles:

```
typedef struct date { int day, month, year; } DateT;
// which could be used as
DateT       date1, *dateptr1;
struct date date2, *dateptr2;
```

## Static/Dynamic Sequences

Previously we have used an *array* to implement a stack

- fixed size collection of heterogeneous elements
- can be accessed via index or via "moving" pointer

The "fixed size" aspect is a potential problem:

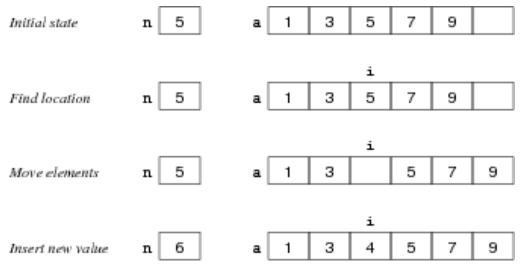
- how big to make the (dynamic) array? (big ... just in case)
- what to do if it fills up?

The rigid sequence is another problems:

- inserting/deleting an item in middle of array

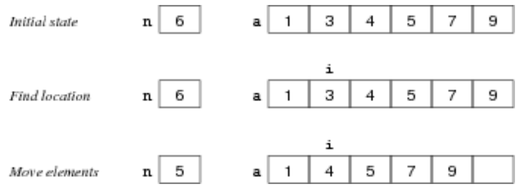
## ... Static/Dynamic Sequences

Inserting a value into a sorted array (`insert(a,&n,4)`):



## ... Static/Dynamic Sequences

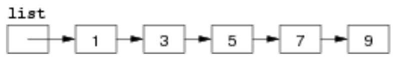
Deleting a value from a sorted array (`delete(a,&n,3)`):



## Dynamic Sequences

The problems with using arrays can be solved by

- allocating elements individually
- linking them together as a "chain"



Benefits:

- insertion/deletion have minimal effect on list overall
- only use as much space as needed for values

## Self-referential Structures

To realise a "chain of elements", need a *node* containing

- a value
- a link to the next node

In C, we can define such nodes as:

```
typedef struct node {
   int data;
   struct node *next;
} NodeT;
```

## ... Self-referential Structures

Note that the following definition does not work:

```
typedef struct {
   int data;
   NodeT *next;
} NodeT;
```

Because `NodeT` is not yet known (to the compiler) when we try to use it to define the type of the `next` field.

The following is also illegal in C:

```
struct node {
   int data;
   struct node recursive;
};
```

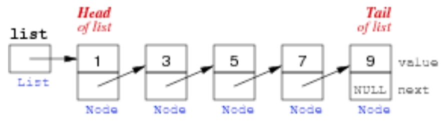
Because the size of the structure would have to satisfy `sizeof(struct node) = sizeof(int) + sizeof(struct node) = ∞`.

## Linked Lists in C

## Linked Lists

To represent a chained (linked) list of nodes:

- we need a pointer to the first node
- each node contains a pointer to the next node
- the `next` pointer in the last node is `NULL`



## ... Linked Lists

Linked lists are more flexible than arrays:

- values do not have to be adjacent in memory
- values can be rearranged simply by altering pointers
- the number of values can change dynamically
- values can be added or removed in any order

Disadvantages:

- it is not difficult to get pointer manipulations wrong
- each value also requires storage for `next` pointer

## Memory Storage for Linked Lists

Linked list nodes are typically located in the heap

- because nodes are dynamically created

Variables containing pointers to list nodes

- are likely to be local variables (in the stack)

Pointers to the start of lists are often

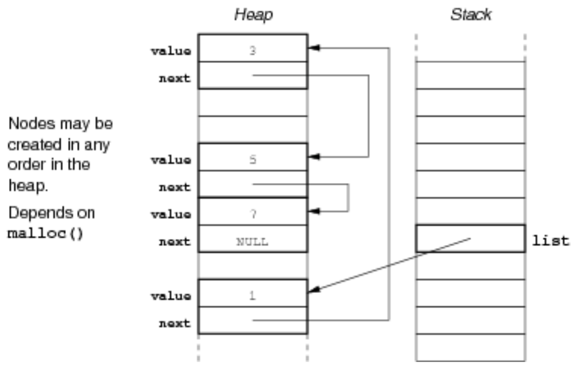
- passed as parameters to function
- returned as function results

## ... Memory Storage for Linked Lists

Create a new list node:

```
NodeT *makeNode(int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;       // initialise data
   new->next = NULL;    // initialise link to next node
   return new;          // return pointer to new node
}
```

## ... Memory Storage for Linked Lists



## Exercise #5: Creating a linked list

Write C-code to create a linked list of three nodes with values 1, 42 and 9024.

```
NodeT *list = makeNode(1);
list->next  = makeNode(42);
list->next->next = makeNode(9024);
```

## Iteration over Linked Lists

When manipulating list elements

- typically have pointer `p` to current node (`NodeT *p`)
- to access the data in current node: `p->data`
- to get pointer to next node: `p->next`

To iterate over a linked list:

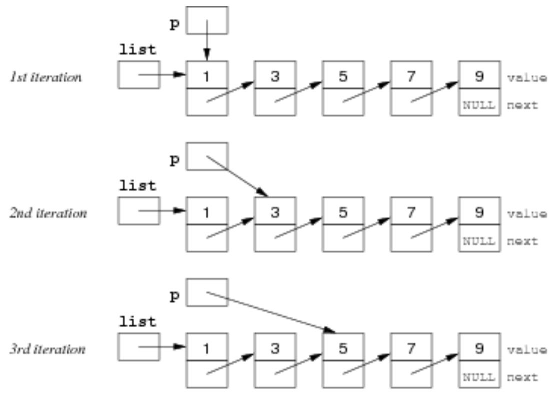
- set `p` to point at first node (head)
- examine node pointed to by `p`
- change `p` to point to next node
- stop when `p` reaches end of list (`NULL`)

## ... Iteration over Linked Lists

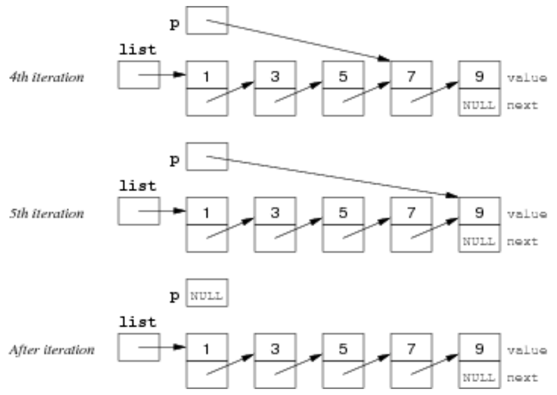
Standard method for scanning all elements in a linked list:

```
NodeT *list;  // pointer to first Node in list
NodeT *p;     // pointer to "current" Node in list

p = list;
while (p != NULL) {
        … do something with p->data …
        p = p->next;
}

// which is frequently written as

for (p = list; p != NULL; p = p->next) {
        … do something with p->data …
}
```

## ... Iteration over Linked Lists



## ... Iteration over Linked Lists



## ... Iteration over Linked Lists

Check if list contains an element:

```
int inLL(NodeT *list, int d) {
   NodeT *p;
   for (p = list; p != NULL; p = p->next)
      if (p->data == d)       // element found
         return 1;
   return 0;                  // element not in list
}
```

Print all elements:

```
void showLL(NodeT *list) {
   NodeT *p;
   for (p = list; p != NULL; p = p->next)
      printf("%6d", p->data);
}
```

## Exercise #6: Traversing a linked list

What does this code do?

```
1  NodeT *p = list;
2  while (p != NULL) {
3     printf("%6d", p->data);
4     if (p->next != NULL)
5        p = p->next->next;
6     else
7        p = NULL;
8  }
```

What is the purpose of the conditional statement in line 4?

Every second list element is printed.

If `*p` happens to be the last element in the list, then `p->next->next` does not exist.
The if-statement ensures that we do not attempt to assign an invalid address to `p` in line 5.

## Exercise #7: Traversing a linked list

Rewrite **`showLL()`** as a recursive function.

```
void printLL(NodeT *list) {
   if (list != NULL) {
      printf("%6d", list->data);
      printLL(list->next);
   }
}
```

## Modifying a Linked List

Insert a new element at the beginning:

```
NodeT *insertLL(NodeT *list, int d) {
   NodeT *new = makeNode(d);  // create new list element
   new->next = list;          // link to beginning of list
   return new;                // new element is new head
}
```

Delete the first element:

```
NodeT *deleteHead(NodeT *list) {
   assert(list != NULL);  // ensure list is not empty
   NodeT *head = list;    // remember address of first element
   list = list->next;     // move to second element
   free(head);
   return list;           // return pointer to second element
}
```

What would happen if we didn't `free` the memory pointed to by `head`?

## ... Modifying a Linked List

Delete a specific element (recursive version):

```
NodeT *deleteLL(NodeT *list, int d) {
   if (list == NULL) {            // element not in list
      return list;

   } else if (list->data == d) {
      return deleteHead(list);    // delete first element

   } else {                     // delete element in tail list
      list->next = deleteLL(list->next, d);
      return list;
   }
}
```

## Exercise #8: Freeing a list

Write a C-function to destroy an entire list.

Iterative version:

```
void freeLL(NodeT *list) {
   NodeT *p;

   p = list;
   while (p != NULL) {
      NodeT *temp = p->next;
      free(p);
      p = temp;
```

```
    }
}
```

Why do we need the extra variable `temp`?

## Stack ADT Implementation

Linked list implementation (`stack.c`):

Remember: `stack.h` includes `typedef struct StackRep *stack;`

```
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

typedef struct node {
   int data;
   struct node *next;
} NodeT;

typedef struct StackRep {
   int    height;   // #elements on stack
   NodeT *top;      // ptr to first element
} StackRep;

// set up empty stack
stack newStack() {
   stack S = malloc(sizeof(StackRep));
   S->height = 0;
   S->top = NULL;
   return S;
}

// remove unwanted stack
void dropStack(stack S) {
   NodeT *curr = S->top;
   while (curr != NULL) {  // free the list
      NodeT *temp = curr->next;
      free(curr);
      curr = temp;
   }
   free(S);           // free the stack rep
}
```

```
// check whether stack is empty
int StackIsEmpty(stack S) {
   return (S->height == 0);
}

// insert an int on top of stack
void StackPush(stack S, int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;
   // insert new element at top
   new->next = S->top;
   S->top = new;
   S->height++;
}

// remove int from top of stack
int StackPop(stack S) {
   assert(S->height > 0);
   NodeT *head = S->top;
   // second list element becomes new top
   S->top = S->top->next;
   S->height--;
   // read data off first element, then free
   int d = head->data;
   free(head);
   return d;
}
```

## Summary: Memory Management Functions

**`void *malloc(size_t nbytes)`**

- aim: allocate some memory for a data object
- attempt to allocate a block of memory of size `nbytes` in the heap
- if successful, returns a pointer to the start of the block
- if insufficient space in heap, returns `NULL`

Things to note:

- the location of the memory block within heap is random
- the initial contents of the memory block are random

## ... Summary: Memory Management Functions

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

- releases a block of memory allocated by `malloc()`
- `*ptr` is the start of a dynamically allocated object
- if `*ptr` was not `malloc()`'d, chaos will ensue

Things to note:

- the contents of the memory block are not changed
- all pointers to the block still exist, but are not valid
- the memory may be re-used as soon as it is `free()`'d

## Tips for Week 4 Problem Set

Main theme: *Dynamic data structures*

- Test your understanding of memory allocation and deallocation
- Create, use and free a dynamic array (Exercise 3)
- Think about how to design and use a dynamic queue ADT (Exercise 4)
- Design and implement functions for dynamic linked lists (Exercise 5)

```
prompt$ ./llbuild
Enter an integer: 12
Enter an integer: 34
Enter an integer: 56
Enter an integer: quit
Finished. List is 12->34->56
```

- Challenge Exercise: wrack your brain — split linked list in two halves without traversing it twice

## Summary

- Memory management
- Dynamic data structures
- Linked lists

- Suggested reading:
    - Moffat, Ch.10.1-10.2
    - Sedgewick, Ch.3.3-3.5,4.4,4.6