Dynamic Heap Memory Management

Up until this point, we've only used global variables and local (stack allocated) variables – which are destroyed as soon as they go out of scope

There is another area of memory called the heap which can hold dynamic memory

A chunk of dynamic memory may be created (or **allocated**) at any point in the run-time of a program, typically inside a constructor function which returns a pointer to it, and then the pointer is passed around, and the data to which it points modified, **for the rest of the lifetime of the program**

In Java and Kotlin, destroying unreferenced dynamically allocated values was done though a process of **automatic garbage collection**

In C, you will need to design your own strategy for deallocating dynamically allocated memory when you have finished with it

Incorrect memory handling is a major source of C programming errors

To use `malloc()`, you must include `stdlib.h`:

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

int *makeintarray( int n ) {
  int *p = malloc( n * sizeof(int) );       // allocate memory block
  assert( p != NULL );                      // check malloc() succeeded
  for( int i=0; i<n; i++ ) { p[i] = i; }    // initialize it
  return p;
}

int main( void ) {
  int size = 100;
  int *array = makeintarray( size );

  for( int i=0; i<size; i++ ) { array[i] *= 10; }
  printf( "array[5] = %d\n", array[5] );

  free(array);                              // deallocate it once we're done
  return 0;
}
```

Here, `main()` calls `makeintarray(100)`

On entering `makeintarray()`, we get two local variables, `n = 100` and `int *p`

`malloc()` makes `p` point to the beginning of a suitably sized block of memory on the heap

`malloc()` can fail, returning NULL to signal that the heap is full. Here I've asserted that it must succeed

A non-NULL `p` may be used exactly like a basal pointer to an array of size `n` - so we initialize it's elements

`makeintarray()` then returns that pointer to it's caller: Here that's `main()` which stores the pointer in it's local `int *array` variable, and uses it as a basal pointer

To deallocate the block when we're done: `free(array)`

`malloc()` is completely unaware of how the returned memory will be used

`malloc()` takes its size parameter in **bytes**, **not elements**

We need to use `sizeof()` to work out how much memory we need - **portably**

`malloc()`'s return type is `void *` (a generic pointer to anything)

A void pointer may be assigned to/from a pointer variable by a simple assignment, the value is implicitly typecast to/from `void *`, as in the idiomatic:
`int *p = malloc( n * sizeof(int) );`

`malloc()` also takes care to align the block of memory that it allocates, where necessary. For example, on an architecture where an 32-bit int has to be stored in memory on a 4-byte boundary, `malloc()` would align any block with size 4n on a 4-byte boundary, just in case it's storing a dynamic array of integers

`void *malloc( size_t size );` allocates a memory block of `size` bytes and returns a pointer to the allocated memory

`void *calloc( size_t nmemb, size_t size );` allocates a memory block that can hold `nmemb` elements of `size` bytes. Every byte in the block is initialised to 0

`void *realloc( void *ptr, size_t size );` resizes an already allocated memory block to the supplied size, preserving the contents. May need to move the block. See `man realloc` for details. It's quite tricky to use, but is vital when needed `size_t` is a standard type, think of it as the largest unsigned integer type available - `sizeof()` is defined to return a `size_t`

`malloc()`, `realloc()` and `calloc()` all return NULL if the allocation fails

You must check for this, and handle it

`malloc()` and `realloc()` return an uninitialised memory block. `calloc()` returns a zeroed memory block, which sounds convenient, but I prefer to do my own initializations for safety and portability

`void free( void *ptr );` deallocates a memory block previously allocated. The pointer passed to `free()` must come from `malloc` / `calloc` / `realloc` or be NULL as a convenient no-op

If you fail to `free()` all the memory you allocate, you have a **bug**

It won't cause your code to crash immediately. However, it makes a later `malloc()` more likely to fail - because it causes your program to leak memory. If your program is short-lived, you may get away with leaking memory, as `exit()` (and returning from `main()`) destroys the whole heap safely

But if your program hangs around forever then this will cause you serious problems

I give a worked example of how to diagnose and fix such leaks using `valgrind` in the second C Tools lecture

Remember to `free()` everything that you `malloc()` / `calloc()`, and **use `valgrind` when you think you have `free()`d everything and want to check that you're right**

In the code you write, `exit()`ing if any `malloc()` fails is probably ok, as in:

```
int *p = malloc(...);
if( p == NULL ) { fprintf( stderr, "Internal error: run out of memory\n" ); exit(1); }
```

This would not be acceptable in production quality code - you'd have to handle the failure somehow and carry on

You might want to handle the failure check outside of the allocation function:

```
int *makeintarray( int n ) {
  int *p = malloc( n * sizeof(int) );       // allocate memory block
  if( p == NULL ) return NULL;              // return failure if it failed
  for( int i=0; i<n; i++ ) { p[i] = i; }    // otherwise initialize it
  return p;
}

int main( void ) {
  int size = 100;
  int *array = makeintarray( size );
  assert( array != NULL );
  ...
```

Dynamically allocating Structs on the heap

You can also allocate structs on the heap using `malloc()`:

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

struct point { double x, y; };

int main( void ) {
        struct point *p1 = malloc( sizeof(struct point) );
        assert( p1 != NULL );
        p1->x = 10;
        p1->y = 20;
        printf( "p1: (%g, %g)\n", p1->x, p1->y );
        free( p1 );
        return 0;
}
```

Note that you use `malloc( sizeof(struct point) )` in order to have the compiler work out how big a block to allocate

A common mistake is to allocate a pointer-sized block, not a struct-sized block

Earlier we said that every type must have been defined before it is used, so that it's size and memory layout is known

The only exception to this is that you can **forward declare a struct**:
`struct wibble; // "struct wibble exists"`

Then declare a pointer to that struct, giving an incomplete type:
`typedef struct wibble *wibble; // wibble is a pointer to struct wibble, whatever that is`

You can then use that incomplete type without knowing the full definition (and hence the size or layout) of the struct

Such an incomplete type is typically placed in a header file and enables an ADT to confine the knowledge of how the structure type is implemented inside the implementation

This works as long as users of the ADT never need to dereference the pointer themselves, only using the public ADT functions (methods?). It works because all pointers have the same size on a particular architecture

An Example:

wibble.h file

```
struct wibble;                 // "struct wibble exists"
typedef struct wibble *wibble; // our "incomplete type", wibble.

extern wibble make_wibble( void );
extern void free_wibble( wibble );
```

wibble.c file

```
#include "wibble.h"

struct wibble {        // give true definition of "struct wibble"
  double x, y;
  ...
};

// now implement the public functions named in the .h file
wibble make_wibble( void ) {
  ...
}
```

Inside `wibble.c`, for example inside `make_wibble()`, you can `malloc(sizeof(struct wibble))` because the size is known

Since global, stack and heap allocated memory may contain uninitialised values, it's useful to be able to initialize large blocks quickly

We can set every byte in a memory block to a specific value using the `memset()` function from `string.h`:
`void *memset( void *s, int c, size_t n );`

Every byte in the `n`-byte block pointed to by `s` is set to the value `c`

Although `c` is an int, it is converted to an `unsigned char` first

`s` is returned - and usually ignored

memset() example misc/memset_example.c

```
char quote[] = "To be or not to be";
memset( quote+3, '.', 9 );
printf( "%s\n", quote );
```

Output

```
To ...... to be
```

Copying blocks of memory may be done using the `memcpy()` function from `string.h`:
`void *memcpy( void *dest, const void *src, size_t n );`

This copies `n` bytes from `src` to `dest`, returning `dest`

The source and destination blocks must not overlap

If they do, use `memmove()` - see `man memmove` for details

memcpy() example misc/memcpy_example.c

```
char str[] = "Morning World!";
char *time = "Eventime";
memcpy( str, time, 4 );
printf( "%s\n", str );
```

Output

```
Evening World!
```