Dynamic Memory Allocation

Introduction

- C manages memory statistically, automatically and dynamically.
- Static-duration variables are allocated in main memory, usually along with the executable code of the program, and persist for the lifetime of the program;
- Automatic-duration variables are allocated on the stack and come and go as functions are called and return.
- For static-duration and automatic-duration variables, the size of the allocation is required to be compile-time constant .
- If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.

Run-time allocation

- Limitations of static allocation is avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by allocating it from the heap, an area of memory structured for this purpose.
- In C, the library function **malloc** is used to allocate a block of memory on the heap. The program accesses this block of memory via a **pointer** that malloc returns.
- When the memory is no longer needed, the pointer is passed to **free** which deallocates the memory so that it can be used for other purposes.

- malloc()

This function is used to allocate memory dynamically.

Declaration:void *malloc(size_t size);

The argument size specifies the number of bytes to be allocated.

On success malloc() returns a pointer to the first byte of allocated memory.

Ptr=(datatype)*malloc(specific size)

Note: `void *` denotes a generic pointer type

Allocating new heap memory

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

- Allocate a block of `size` bytes,
  return a pointer to the block
  (`NULL` if unable to allocate block)

```
void *calloc(size_t num_elements, size_t element_size);
```

- Allocate a block of `num_elements * element_size` bytes,
  initialize every byte to zero,
  return pointer to the block
  (`NULL` if unable to allocate block)

Allocating new heap memory

```
void *realloc(void *ptr, size_t new_size);
```

- Given a previously allocated block starting at `ptr`,
  - change the block size to `new_size`,
  - return pointer to resized block
    - If block size is increased, contents of old block may be copied to a completely different region
    - In this case, the pointer returned will be different from the `ptr` argument, and `ptr` will no longer point to a valid memory region
- If `ptr` is `NULL`, `realloc` is identical to `malloc`
- Note: may need to cast return value of `malloc/calloc/realloc`:

```
char *p = (char *) malloc(BUFFER_SIZE);
```

Deallocating heap memory

```
void free(void *pointer);
```

- Given a pointer to previously allocated memory,
  - put the region back in the heap of unallocated memory
- Note: easy to forget to free memory when no longer needed...
  - especially if you're used to a language with "garbage collection" like Java
  - This is the source of the notorious "memory leak" problem
  - Difficult to trace – the program will run fine for some time, until suddenly there is no more memory!