# Section 3.2
# Dynamic Memory Allocation

1. Allocating memory
2. Deallocating memory
3. Memory leaks
4. Double pointers

© 2018 Christine Laurendeau, Doron Nussbaum


## 3.2.1 Allocating Memory

- Functions used to allocate memory at runtime:
  - `malloc`
    - allocates a number of bytes
  - `calloc`
    - clears memory and allocates a number of elements
    - each element is a specified number of bytes

© 2018 Christine Laurendeau, Doron Nussbaum


## Allocating Memory (cont.)

- Both allocation functions return pointer to `void`
  - what is `void`?
    - placeholder data type
    - Used to hide variables (in conjunction with a pointer)
  - `void` indicates
    - no data type is returned
      - function does not return a value
    - any data type (in conjunction with a pointer)
      - to be typecast
  - returned pointer must be *typecast*

© 2018 Christine Laurendeau, Doron Nussbaum


## Allocating Memory (cont.)

- What is typecasting?
  - explicit type conversion
    - changing the data type of a variable to another data type
    - Change the interpretation of the data (variable)
  - usually used on pointers
    - change the data type that pointer points to
    - possible because pointers are all the same size
  - can be used on non-pointer variables

    - cannot always "fit" a variable of one type into another type

© 2018 Christine Laurendeau, Doron Nussbaum


## Memory Allocation Function

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

- Description:
  - reserves in memory the number of bytes specified by ***size***
  - returns the start address of the new block of reserved memory
  - typecast the returned pointer as pointer to data type required
  - do not lose this address!

© 2018 Christine Laurendeau, Doron Nussbaum


## Clear and Allocate Function

```
void *calloc(size_t nitems, size_t size)
```

- Description:
  - reserves in memory the number of elements specified by ***nitems***, each of ***size*** bytes
  - returns the start address of the new block of reserved memory
  - every byte is initialized to zero

© 2018 Christine Laurendeau, Doron Nussbaum

# Accessing Allocated Memory

- Two ways:
  - pointer notation
    - dereference pointer
    - use pointer arithmetic
  - array notation
    - use subscripts

© 2018 Christine Laurendeau, Doron Nussbaum


# 3.2.2 Deallocating Memory

- When to deallocate?
  - When allocated memory is not needed
- Why deallocate memory?
  - May run out of memory
  - May access virtual memory
  - otherwise we get a memory leak
    - ... more on this soon ...
- Function to deallocate memory at runtime:
  - **`free`**
    - deallocates a specified block of memory

© 2018 Christine Laurendeau, Doron Nussbaum


# Deallocation Function

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

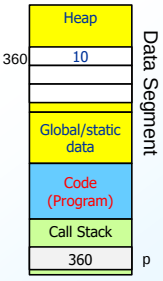
- Description:
  - marks as non-reserved the block of memory pointed to by ***`ptr`***
  - ***`ptr`*** must point to beginning of dynamically allocated block
    - otherwise, behaviour is undefined

© 2018 Christine Laurendeau, Doron Nussbaum


```
#define ARR_SIZE 3
int main(int argc, char **argv)
{
   int *p = NULL;

   p = malloc(ARR_SIZE * sizeof(int));
   p[0] = 10;

   free(p);
  return(0);
}
```




© 2018 Christine Laurendeau, Doron Nussbaum


# 3.2.3 Memory Leaks

- What is a memory leak?
  - dynamically allocated memory with no pointers to it
- How does this happen?
  - pointer gets *clobbered*
    - overwritten
  - pointer moves out of scope
    - e.g. pointer gets popped off the function call stack

© 2018 Christine Laurendeau, Doron Nussbaum


# Memory Leaks (cont.)

- Why is this a problem?
  - access to data is permanently lost
  - finite amount of heap space is allocated to each program
    - once allocated, memory is reserved until
      - it is explicitly deallocated
      - program terminates
    - if pointer is lost, memory remains reserved
      - we might run out of heap space!

© 2018 Christine Laurendeau, Doron Nussbaum

## Memory Leaks (cont.)

- How do we prevent leaks?
  - Good bookkeeping
  - always explicitly deallocate memory when you're done with it
    - use `valgrind` in Linux to check
  - if called function allocates memory, pass pointer by reference
    - use double pointers
      - a pointer to a pointer
  - make sure you don't clobber pointers into the heap
  - some languages do *garbage collection*
    - automated mechanism that kicks in to free unreferenced memory

© 2018 Christine Laurendeau, Doron Nussbaum


## 3.2.4 Double Pointers

- What is a double pointer?
  - a pointer to a pointer
- Why do we need these?
  - to pass pointers by reference
    - enables changing pointer values in called function
  - for dynamically allocated multi-dimensional arrays

© 2018 Christine Laurendeau, Doron Nussbaum