Memory Management I

- two kinds of memory: stack and heap
- stack memory:
  - essentially all non-pointer (why not pointers? there's a caveat) variables and pre-declared arrays of fixed (i.e. fixed before compilation) length live in the stack
  - all local (and non-`static`) variables within a function live on the stack and they "die" automatically when the function exits
- heap memory:
  - "dynamically" allocated memory (hold your breath, its coming your way!) reside in the heap and they do not "die" automatically, you need to "allocate" and "dellacote" them

Memory Management II

- when to use which?
- suppose your job is to *store* and do some "stuff" with the first `N`-natural numbers, where `N` is used specified
  - suppose beforehand you fix an upper limit and ask of the user to only specify `N < 10000` then you may declare:

```
#define MAX_NN 10000

int iarr[MAX_NN];
```

  - note fixing an upper limit is kind of restrictive so you may decide to take a postive number, say, `nn`, as an input from the user and work with an array of length `nn` then you may declare (hold on, details coming later):

```
int nn, *iarr;

/* scanf the value of nn here */
iarr = (int *) malloc(nn * sizeof(int));
```


Memory Management III

- the following two snippets do not do the same thing:
  - below `MAX_NN` is literally substitued by 10000 before compilation i.e. pretend that manually you had typed 10000 in your code wherever you see `MAX_NN` (why do this? it avoids "magic numbers", coming later!)

```
#define MAX_NN 10000

int iarr[MAX_NN];
```

  - below `max_nn` (note `C` is case-sensitive and hence `max_nn` and `MAX_NN` are different "symbols") is a variable and hence the size of `iarr` is also variable

```
int max_nn = 100000, iarr[max_nn];
```

  - the above is not allowed by `gcc -ansi -pedantic` i.e. in ANSI C but C99 allows it so plain `gcc` without those options would compile it just fine
  - do not use this feature: is the right place to use dynamic memory management with `malloc( )`, `free( )` and the sort

Memory Management IV

- example of stack memory:

```
#define SMALL_LEN 50

double *
mem_stack1 (void)
{
        double dval;

        return &dval;
}

double *
mem_stack2 (void)
{
        double dval[SMALL_LEN];

        return dval;
}
```

Memory Management V

- heap memory:
  - allocation:

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

- note `malloc` returns a `void *` pointer but, say, we want to allocate space for `SMALL_LEN` many `double`s, here's how you do it:
  - get the right amount of space:

    ```
    malloc(SMALL_LEN * sizeof(double))
    ```

  - "cast" the `void *` pointer to a pointer to `double`:

    ```
    (double *) malloc(SMALL_LEN * sizeof(double))
    ```

  - note, `(double *)` is the "cast" operator, in general `(typename) foo` casts variable `foo` to type `typename`

Memory Management VI

- heap memory:
  - deallocation:

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

- note `free` takes a `void *` as an argument thus any pointer could be passed to it:

  ```
  double *dval_arr = NULL;

  dval_arr = (double *) malloc(SMALL_LEN * sizeof(double));
  free(dval_arr);
  ```

- note however though, `free` only “frees” the space pointed to by `dval_arr`, it doesn’t “free” `dval_arr` itself, what does that mean?
- the above “thing” is called the problem of “dangling pointers”

Memory Management VII

- example of heap memory:

```
#define SMALL_LEN 50

double *
mem_heap1 (void)
{
        double *dval_arr;

        dval_arr = (double *) malloc(SMALL_LEN * sizeof(double));
        return dval_arr;
}
```

Memory Management VIII

- the stack memory *may/may not*, we don't know, cause runtime error giving `Segmentation Fault`
- the heap memory if managed properly would work just fine
- remember to always deallocate heap memory which you wouldn't be using anymore
- failure to properly deallocate causes what is known as "memory-leak" which makes the program slow and it ultimately gets killed by the operating system

```
#define BIG_LEN 200000

/* Memory leak example: don't do something like the following */
for (ii = 0; ; ) {
        printf("count = %d\n", ++ii);
        dval_arr = (double *) malloc(BIG_LEN * sizeof(double));
}
```

Memory Management IX

- example stack memory mis-usage and heap memory usage:

```
int
main (int argc, char **argv)
{
        int ii;
        double *dval_arr = NULL;

        dval_arr = mem_stack1( );
        for (ii = 0; ii < SMALL_LEN; ++ii) {
                dval_arr[ii] = 10 * ii;
                printf("dval_arr[%d] = %g\n", ii, dval_arr[ii]);
        }
        dval_arr = NULL;

        dval_arr = mem_heap1( );
        for (ii = 0; ii < SMALL_LEN; ++ii) {
                dval_arr[ii] = 10 * ii;
                printf("dval_arr[%d] = %g\n", ii, dval_arr[ii]);
        }
        free(dval_arr);
        dval_arr = NULL;
        return 0;
}
```

Memory Management X

- how to guard against memory leakage? do the proper book-keeping
- whenever you use a function and there are pointers involved either in the arguement(s) or in the return value find out who is responsible for the memory of those pointers:
  - the *callee*:

    ```
    char *
    strdup(const char *str);
    ```

    here the callee allocates memory for the return value
  - or the *caller*

    ```
    char *
    stpcpy(char *dst, const char *src);
    ```

    here the caller allocates memory for `dst` and makes sure its big enough to hold a copy of `src`

Memory Management XI

- other memory allocation tools:
  - allocating cleared space: allocate memory and clear it to zero

    ```
    void *
    calloc (size_t count, size_t eltsize);
    ```

  - resizing memory: change (increase or decrease) the size of the block whose address is `ptr` to be `newsize`

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

Memory Management XII

- example of calloc:

```
dval_arr = (double *) calloc(SMALL_LEN, sizeof(double));
```

- example of realloc:

```
dval_arr = (double *) malloc(SMALL_LEN * sizeof(double));
dval_arr = (double *) realloc(dval_arr, 2 * SMALL_LEN * sizeof(double));
```

or

```
dval_arr = (double *) calloc(SMALL_LEN, sizeof(double));
dval_arr = (double *) realloc(dval_arr, 2 * SMALL_LEN * sizeof(double));
```

Memory Management XIII

- checking allocated memory: `malloc`, `calloc` and `realloc` all may return `NULL` to indicate failure to acquire/resize memory
- so we need to *always* check the return value of these functions:

```
double *
mem_heap2 (void)
{
        double *dval_arr;

        dval_arr = (double *) malloc(SMALL_LEN * sizeof(double));
        if (dval_arr == NULL) {
                printf("malloc falied to acquire memory: aborting...\n");
                abort( );
        }
        return dval_arr;
}
```

Code Files

prog9.c