# NWEN 241
# Dynamic Memory Management

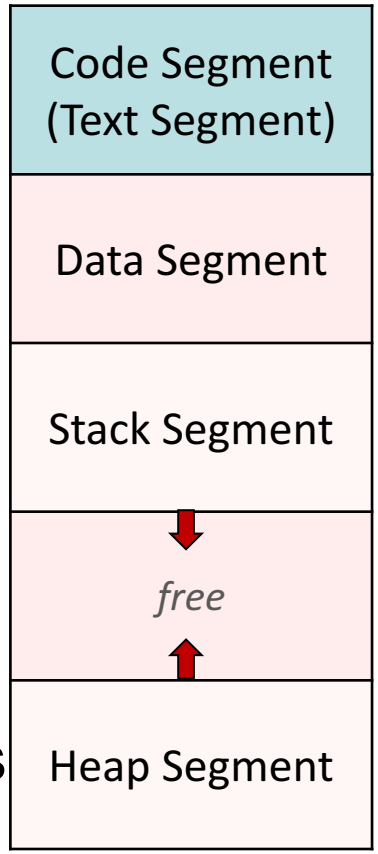
Winston Seah

School of Engineering and Computer Science
Victoria University of Wellington

# Memory Layout of a Program

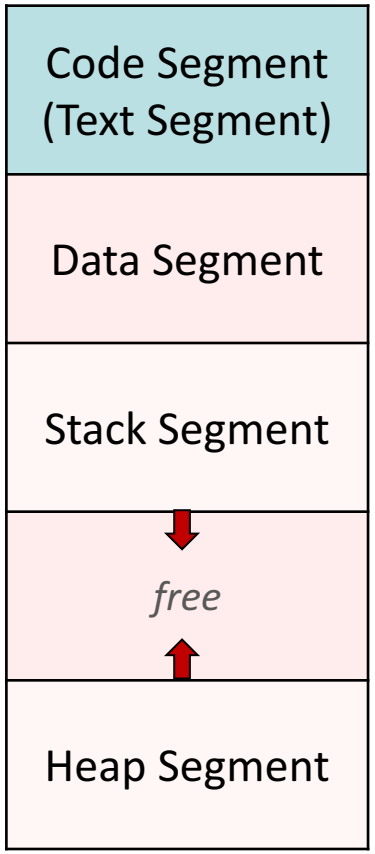
Memory space for program code includes space for machine language code and data

- **Text / Code Segment**
  - Contains program's machine code
- Data spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Stack Segment** – For temporary data, e.g. local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs



# Memory Layout of a Program

- Local variables in functions allocated when function starts:
  - Memory allocated on the Stack Segment
  - When function ends, memory space is freed up
  - Size of data item (int, array, etc.) when allocated (*static allocation*)



# Static Memory Allocation

- Predefine the sizes of arrays, and other variables.
- What if we don't know how much space we will need ahead of time? We can:
  - ask user how many numbers to read in
  - read set of numbers into an array (of appropriate size)
  - calculate the average (look at all numbers)
  - calculate the variance (based on the average)
- Problem: how big do we make the array??
- Using static allocation, we have to make the array as big as the user might specify, and still might not be big enough → re-compile code with large memory allocation.

# Dynamic Memory Allocation

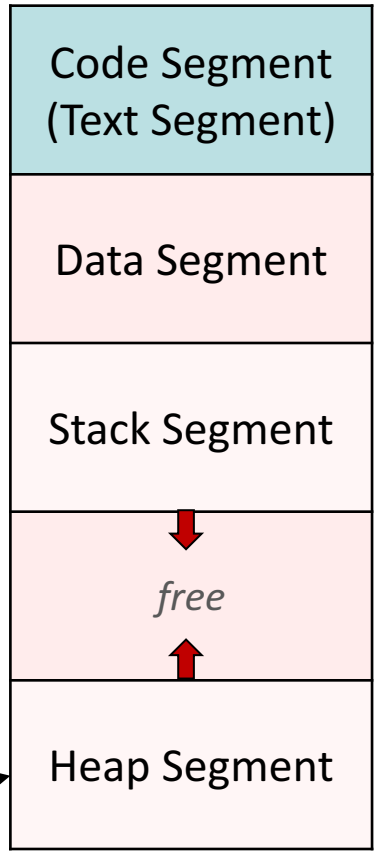
Allow the program to allocate some variables (notably arrays), during the program execution, based on variables in program (dynamically)

Previous example:

- ask the user how many numbers to read,
- then allocate array of appropriate size

Approach:

- Program has routines allowing user to request some amount of memory,
- the user then uses this memory, and
- returns it when they are done
- memory is allocated in the *Data Heap*



# Dynamic Memory Management Functions

**calloc** - allocate arrays of memory

**malloc** - allocate a single block of memory

**realloc** - extend the amount of space allocated previously

**free** - free up a piece of memory that is no longer needed by the program

**Note**: *memory allocated dynamically does not go away at the end of functions, you* ***MUST*** *explicitly* ***free*** *it up*

# calloc – allocate memory for array

Function prototype:

```
void * calloc(size_t num, size_t esize)
```

**`size_t`**

- special type used to indicate sizes,
- unsigned int

**`num`** – number of elements to be allocated in the array

**`esize`** – size of the elements to be allocated; to get the correct value, use **`sizeof (<type>)`**

memory of size **`num*esize`** is allocated on the Data Heap

**`calloc`** returns the address of the 1st byte of this memory;
※ cast the returned address to the appropriate type

if not enough memory is available, **`calloc`** returns **`NULL`**

# calloc Example

```
float *nums;
int a_size;
int idx;

printf("Read how many numbers:");
scanf("%d",&a_size);
nums = (float *) calloc(a_size, sizeof(float));

/* nums is now an array of floats of size a_size */
for (idx = 0; idx < a_size; idx++) {
  printf("Please enter number %d: ",idx+1);
  scanf("%f",&(nums[idx])); /* read in the floats */
}

/* Calculate average, etc. */
```

# free – return memory to heap

Function prototype:

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

- memory at location pointed to by **`ptr`** is released (so that it could be used again)
- program keeps track of each piece of memory allocated by where that memory starts;
- if we free a piece of memory allocated with **`calloc`**, the entire array is freed (released)
- results are problematic if we pass as address to **`free`** an address of something that was not allocated dynamically (or has already been freed)

# free Example

```
float *nums;
int a_size;

printf("Read how many numbers:");
scanf("%d",&a_size);
nums = (float *) calloc(a_size, sizeof(float));

/* use array nums */

/* when done with nums: */

free(nums);

/* would be an error to say it again - free(nums) */
```

# Importance of free()

```
void myfunc() {
  float *nums;
  int a_size = 5;

  nums = (float *) calloc(a_size, sizeof(float));
  /* But no call to free(nums) */
} /* myfunc() ends */
```

When function **`myfunc()`** is called, space for array of size **`a_size`** allocated; when function ends, variable **`nums`** goes away, but the space **`nums`** points at (the array of size **`a_size`**) remains allocated on the Data Heap;

Worse, we have lost the address of that memory space!!!

Problem called ***memory leakage***

# malloc – allocate memory

Function prototype:

**`void * malloc(size_t esize)`**

- Similar to **`calloc`**, except we use it to allocate a single block of the given size **`esize`**
- Like **`calloc`**, memory is allocated from Data Heap
- **`NULL`** returned if not enough memory available
- Memory must be released using **`free`** no longer needed
- Can perform the same function as **`calloc`** if we simply multiply the two arguments of **`calloc`** together
- Following are equivalent:

```
malloc(a_size * sizeof(float))
calloc(a_size, sizeof(float))
```

# realloc – increase memory allocation

Function prototype:

```
void * realloc(void * ptr, size_t esize)
```

**`ptr`** is a pointer to a piece of memory previously dynamically allocated

**`esize`** is new size to allocate (no effect if **`esize`** is smaller than the size of the memory block **`ptr`** points to already)

Function performs following action:

i. allocates memory of size **`esize`**,

ii. copies the contents of the memory at **`ptr`** to the first part of the new piece of memory, and lastly,

iii. old block of memory is freed up.

# realloc Example

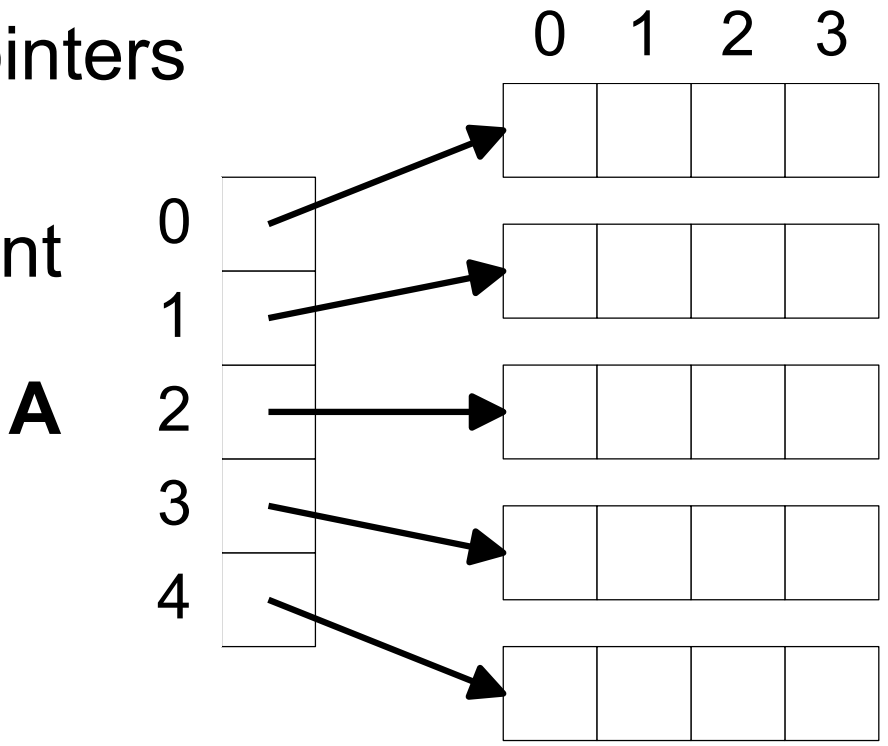
```
float *nums;
int a_size;

nums = (float *) calloc(5, sizeof(float));
/* nums is an array of 5 floating point values */

for (a_size = 0; a_size < 5; I++)
  nums[a_size] = 2.0 * a_size;
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */

nums = (float *) realloc(nums,10 * sizeof(float));
/* An array of 10 floating point values is allocated,
   the first 5 floats from the old nums are copied as
   the first 5 floats of the new nums, then the old nums
   is released */
```

# Allocating memory for 2D array

Can not simply allocate 2D (or higher) array dynamically

Solution:

1. allocate an array of pointers (1st dimension),
2. make each pointer point to a 1D array of the appropriate size



# Allocating memory for 2D array

```
float **A;  /* A is an array (pointer) of float
               pointers */
int X;

A = (float **) calloc(5,sizeof(float *));
/* A is a 1D array (size 5) of float pointers */

for (X = 0; X < 5; X++)
  A[X] = (float *) calloc(4,sizeof(float));
/* Each element of array points to an array of 4 float
   variables */

/* A[X][Y] is the Yth entry in the array that the Xth
   member of A points to */
```

# Irregular-sized 2D array

No need to allocate square 2D arrays:

```
float **A;
int X;

A = (float **) calloc(5,
          sizeof(float *));

for (X = 0; X < 5; X++)
  A[X] = (float **)
           calloc(X+1,
            sizeof(float));
```