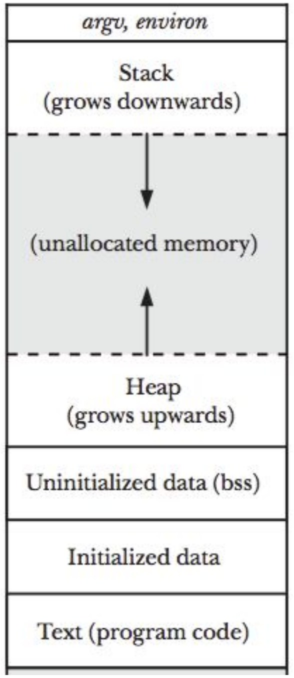
C Programming

Dynamic Memory Management

Dynamic Memory Management

- What we know about different segments of a program
  - **Code Segment** is used to store the program code.
  - The global and static variables and constant string literals are stored in the **Data Segment**.
  - A **Stack** is used to store local variables of functions and and the required information for functions calls and returns
- Sometimes amount of data a program should process is unknown during development and size of the data will be specified during run time.
  - E.g. When we ask the user to enter a name
  - Size of the data is unknown during compilation



Memory Layout of a C Program

Environment is the used for the environment variables and arguments passed to the program. **BSS** (Block Started by Symbol) includes the uninitialized global and static variables. **DS** includes the global initialized constants , static variables and string literals.

Dynamic Memory Management

- In C, there are two mechanisms to handle dynamic size data
  - *Size of such data is unknown during compilation and will be specified/changed during runtime*
  - **Flexible Arrays**
    - A flexible array is a **local** array whose size can be specified during runtime.
    - The **Stack Segment** is used to store flexible arrays
    - ***We shall avoid using flexible arrays***
  - **Dynamic Memory Management**
    - In this mechanism we can allocate and deallocate memory dynamically during runtime
    - The **Heap** memory is a large pool of memory which can be used for storing dynamic size data
    - The **programmer is responsible** to manage the dynamic memory
    - Data stored in the heap, is alive until we release the allocated memory for the data
      - Unlike local variables which exist only in their scops

Dynamic Memory Management

- C provides some functions in stdlib.h for dynamic memory management.
  - **malloc** and **calloc** which are used to allocate a new block of memory
  - **realloc** which is used to resize an allocated block of memory.
  - **free** which is used to release an allocated block of memory.

- Dynamic memory allocation using **malloc**

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

  - It reserves a continuous memory block whose size in *bytes* is ***size***.
  - It returns a pointer (**void pointer**) to the allocated memory block. If it fails, it returns NULL
  - The content of the allocated memory block is unknown.
  - E.g. `int *ptr = (int *)malloc(10 * sizeof(int));`
    - **ptr** is pointing to the first element an array type of int which has 10 elements

Dynamic Memory Management

- Dynamic memory allocation using **calloc**

  *void *calloc(size_t **count**, size_t **size**);*

  - It reserves a continuous memory block whose size in *bytes* is ***count* × *size***.
  - It returns a pointer (**void pointer**) to the allocated memory block. If it fails, it returns NULL
  - It **initializes** all the bytes of the memory block with **0**.
  - E.g. `int *ptr = (int *)calloc(10, sizeof(int));`
    - **ptr** is pointing to the first element of an array type of int which has 10 elements

- **When we use malloc and calloc to allocate memory we shall always check for failures**
  - E.g. `int *ptr = (int *)malloc(10 * sizeof(int)); if (ptr == NULL) { /* error handling */ }`
  - In embedded programming, generally we shall avoid using dynamic memory allocation

Dynamic Memory Management

- Resizing a dynamically allocated memory block using **realloc**

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

- **ptr** is the pointer to the beginning of allocated memory and **size** is the new size of the block.
    - If **ptr** is NULL, realloc behaves like **malloc**.
- It **releases** the memory block pointed by **ptr** and allocates an new block of ***size*** bytes.
- It copies the content of the original memory block to the new block (up to the new size)
    - If the new size is **larger** than the old size, the value of the additional bytes are unspecified
- It returns a pointer(**void pointer**) to the new block.
    - If it fails, it returns NULL, but it does not release the original block or change its content
    - ***We shall always check for failure.*** For example:
        - `int *ptr  = (int *)malloc(2 * sizeof(int)); if (ptr == NULL) { /* error handling */ }`
        - `int *rptr = (int *)realloc(ptr, 4 * sizeof(int)); if (rptr == NULL) { /* error handling */ }`

Dynamic Memory Management

- Releasing a dynamically allocated memory block using **free**

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

  - **ptr** is the pointer to the beginning of the block we want to release
  - When we don't need a dynamically allocated memory block, we shall release it and return it to the operating system
  - If we don't free a dynamically allocated memory properly, a **memory leak** occurs
    - Means that such a memory is lost and it can never be freed
  - As a good practice, after calling **free**, set the pointer to **NULL**. E.g. free(**ptr**); ptr = NULL;
  - *We shall use free to free **only** dynamically allocated memories.*
  - *We shall free a dynamically allocated memory **only** once.*
- Look at **memset**, **memcmp**, **memcpy** and **memmove** functions in **string.h**.