C++

5. Dynamic memory

-Pandav Patel

What is lifetime of variables?

- Block variable? // Stored on stack
- Non-static function variable? // Stored on stack
- Static function variable? // Stored in data section
- Global variable? // Stored in data section
- In all cases seen so far, **lifetime of a variable is tied to its storage area**

- Can we allocate memory in function *fun* and can still access it after execution of function *fun* ends?
- Many times we may not want memory to vanish once function execution ends, and at the same time, we may also not want it to linger throughout the program execution
- **Can we (programmer) control the lifetime of allocated memory?**

How does vector work?

- How does vector manage its size dynamically?
- Where does it store elements?

```
vector<int> v1 = {1, 2, 3, 4, 5};
vector<int> v2(100, 0);

cout << sizeof(v1) << endl;
cout << sizeof(v2) << endl;

cout << v1.size() << endl;
cout << v2.size() << endl;
```

How does vector work?

- Vector stores **actual elements on heap section** and stores **relevant pointers on stack/data section** (pointer to first element, last element and end of capacity - in g++)

```
vector<int> v1 = {1, 2, 3, 4, 5};
vector<int> v2(100, 0);

cout << sizeof(v1) << endl;        // 24
cout << sizeof(v2) << endl;        // 24

cout << v1.size() << endl;         // 5
cout << v2.size() << endl;         // 100
```

How does vector work?

```
vector<int> v1 = {1, 2, 3, 4, 5};
cout << v1.size() << " " << v1.capacity() << endl;
for(int i = 0; i < 6; i++) {
    v1.push_back(6 + i);
    cout << v1.size() << " " << v1.capacity() << endl;
}
```

OUTPUT:

5 5
6 10
7 10
8 10
9 10
10 10
11 20

- **Vector doubles its capacity everytime it runs out of memory**
- How is this possible?
- **Can programmer control the size of allocated memory at runtime?**

Introduction to dynamic memory

- **Dynamic memory** makes it possible for programmer to **control *lifetime* and *size* of allocated memory at runtime**
- Following **4 library functions** declared in **stdlib.h** can be used to control dynamic memory in C and C++
  - malloc
  - calloc
  - realloc
  - free

malloc/calloc - for memory allocation

- malloc and calloc functions allocate memory **at runtime** (on **heap section**) and return pointer to it
- Memory allocated by one call to malloc/calloc is always contiguous.

- malloc
  - Arguments?
  - Return type?
- calloc
  - Arguments?
  - Return type?

malloc/calloc - arguments and return type

- malloc and calloc functions allocate memory **at runtime** (on **heap section**) and return pointer to it
  - malloc
    - One argument which should specify the **size of memory required in terms of bytes**
    - Returns **void *** - address of the first byte of allocated memory
  - calloc
    - Two arguments - **number of elements**, **sizeof each element**
    - Returns **void *** - address of the first byte of allocated memory
- Example:

  void *vp;

  vp = malloc(**100**);

  vp = calloc(**25, sizeof(int)**);

  vp = malloc(**25 * sizeof(int)**);

- Can we use allocated memory using **void pointer**? Why? What is the solution?

malloc/calloc - conversion of void *

```
int *ip;
ip = malloc(100);
ip = calloc(25, sizeof(int));
ip = malloc(25 * sizeof(int));
```

- Above code will work **fine in C**, but will result in following **error in C++**
  - **error: invalid conversion from ‘void*’ to ‘int*’**
- In C++, **void *** can not be allocated to other pointer type without explicit type casting

malloc/calloc - conversion of void *

```
int *ip;
ip = (int *)malloc(100);
ip = (int *)calloc(25, sizeof(int));
ip = (int *)malloc(25 * sizeof(int));
```

- This should work both in C and C++

malloc/calloc - difference

- What is **difference between malloc and calloc** apart from the fact that the arguments are different in both?
  - By **default** value stored at memory allocated by **malloc is garbage**
  - By **default** value stored at memory allocated by **calloc is zero**

- **Allocation can happen anywhere on the heap**.
  - We can not assume any pattern in the location of memory that is allocated during multiple calls to malloc/calloc.

malloc/calloc - what if memory allocation fails?

- They **return NULL pointer**
- Such situations need to be handled properly

```
int *ip;
ip = (int *)malloc(100);
if(NULL == ip) {
    // Write code to handle failed memory allocation
}
// Continue assuming memory allocation was success
```

malloc/calloc - example

```
int *dyn_arr = (int *)malloc(10 * sizeof(int));
if(NULL == dyn_arr) {
    cout << "Memory allocation failed" << endl;
    return -1;
}

for(int i = 0; i < 10; i++)
    cin >> dyn_arr[i];
for(int i = 0; i < 10; i++)
    cout << dyn_arr[i] << " ";
cout << endl;
```

INPUT:
1 2 3 4 5 6 7 8 9 10

OUTPUT:
1 2 3 4 5 6 7 8 9 10

What is the real benefit of dynamic memory?

- So far we have created an array whose size can be specified at runtime.
  - But that can be done using **variable length array** too.
    - int size;
    - cin >> size;
    - int arr[size];

- Then what is the **actual benefit of using dynamic memory**
  - We can **change the size** of dynamic array during runtime **using *realloc* function**
    - While size of variable length array can not be changed once specified
  - We can **control lifetime** of dynamic memory during runtime **using *free* function**
    - While lifetime of variable length array is tied to the block in which it is declared

realloc - how to alter size of dynamic array during runtime?

- *realloc* function defined in stdlib.h can be used to alter the size of dynamic array.
- ***realloc***
  - **Arguments** -
    - **Pointer to dynamic memory** whose size has to be reallocated
    - **New size**
  - **Return type** -
    - If reallocation is **successful**, it returns **void pointer to reallocated memory**. **Otherwise** it returns **NULL**

realloc - example

```
// Assume that dyn_arr is dynamic int array
void *vp = realloc(dyn_arr, 20 * sizeof(int));
if(NULL == vp) {
    cout << "Memory reallocation failed" << endl;
    return -2;
}
dyn_arr = (int *)vp;
// From here on dyn_arr can be used with new size
```

realloc - internal details

- Reallocation can happen at the same address or at the new address
  - If memory is **re-allocated at the same address**
    - It simply increases/decreases the size of allocated memory and returns the same address
  - If memory is **re-allocated at the new address**, *realloc* does following things before returning address of new location
    - **Data from old location is copied to new location**
    - **Memory at the old address is freed**
- If size has increased then **added bytes will have garbage value**
- If size has **reduced** then there will be **loss of data**

- If call to **realloc fails** then it **returns NULL**
  - **In that case realloc does not free original memory**

realloc - beware of memory leak

```
// Assume that dyn_arr is dynamic int array
void *dyn_arr = realloc(dyn_arr, 20 * sizeof(int));
if(NULL == dyn_arr) {
    // We lost pointer to original memory which is not yet freed
    // This leads to memory leak
    // As memory can not be freed till program execution ends
    cout << "Memory reallocation failed" << endl;
    return -2;
}
// From here on dyn_arr can be used with new size
```

free - how to control lifetime of dynamic memory?

```
// code before this
int *dyn_arr = (int *)malloc(10 * sizeof(int)); //lifetime starts
if(NULL == dyn_arr) {
    cout << "Memory allocation failed" << endl;
    return -1;
}

for(int i = 0; i < 10; i++)
    cin >> dyn_arr[i];
for(int i = 0; i < 10; i++)
    cout << dyn_arr[i] << " ";
cout << endl;

free(dyn_arr); //lifetime ends
// remaining code
```

free - how to control lifetime of dynamic memory?

```
void *allocate(int size) {
    void *ptr = malloc(size); // lifetime starts
    if(NULL == ptr) {
        cout << "Memory allocation failed" << endl;
        exit(-1);
    }
    return ptr;
}

int main() {
    int *dyn_arr = (int *)allocate(10 * sizeof(int));

    for(int i = 0; i < 10; i++)
        cin >> dyn_arr[i];
    for(int i = 0; i < 10; i++)
        cout << dyn_arr[i] << " ";
    cout << endl;

    free(dyn_arr); //lifetime ends

    return 0;
}
```

free - internal details

- *free* takes one argument (void pointer to dynamic memory) and return **void** (does not return anything)
    - **Address passed to free must be the one returned by malloc, calloc or realloc and it must not already be freed**
    - If we pass **invalid address** then program will **crash**
        - E.g. **free(dyn_arr + 1);**
- Lifetime of dynamic memory ends once it is freed
    - We should **not use freed memory** otherwise it will result in **undefined behaviour**
    - Hence it is wise idea to **reset pointer to NULL** after call to free
        - free(dyn_arr);
            - frees memory but **dyn_arr still holds address of freed memory**
        - dyn_arr = NULL; // Wise idea
- We must **free dynamic memory once it is no longer needed**
- If we do not explicitly free the dynamic memory then it will be **freed when program execution ends**

Summary

- Lifetime of pointer pointing to dynamic memory is still tied to its storage area, but lifetime of dynamic memory is controlled by programmer
- Vector internally uses dynamic memory to control the size of the vector