C++ Array, Structure, Pointer and Address space layout

Seonghyun Park

1 November, 2018

Outline

1. More types
   - Array
   - Structure

2. Memory management
   - Pointer
   - Address space

Today's Goal

- Learn more types (array, structure).
- Learn how to point location in the address space layout with pointers
- See how the address space layout looks like (especially stack and heap)
- Understand how call-by-reference and call-by-value works differently

Array

- Arrays are used to handle multiple data of same type.
- When creating an array, the size of the array should be known.
- You can access the array elemnt with `[]` operator.
  (e.g. `int arr[5]; arr[3] = 4; int y = arr[4];`)
- NEVER attmept to access index out of bound!

Example

```
int arr[] = { 1, 2, 3, 4, 5 };
char message[10];
std::string friends[3];
```

Structure

- Structures are used to define a new that type consists of different types of associated values.
- A structure can contain arrays, other structures (of course an array of structures), and pointers, including pointer to the type of itself
- In C, `structs` are used with prefix `struct` (e.g. `struct foo`), but in C++ we just use it as `foo`
- Member variables are accessed via `.` operator (e.g. `point.x`).

Example

```
struct Car {
Wheel wheels[4];
Seats seats[2];
};
Car car = { { w1, w2, w3, w4 }, { s1, s2, s3, s4 } };
```

Pointer

- Each object (or even code) has its own address in the memory.
- When we want to access object in the memory with its address, we use *Pointers*.
- Pointers has type name like (e.g. `int*`, `Car*`) and pointers hold address as its value. (e.g. `int* xp = 0x7ff56e`)
- We can *dereference* a value from a pointer using `*` operator (e.g. `int y = *x;`)
- We can initialize a pointer with the address of existing object's address using `&` operator (e.g. `int* xp = &y;`)
- If we have a pointer to a structure, we can dereference the member variables using `->` operator (e.g. `Wheel w = Car->wheels[1];`)

Pointer (Cont'd)

- An array can be used as a pointer (e.g. `int arr[10]; int* x = arr;`).
- We can do arithmetics with pointer of type which has concrete size. i.e (+, -)
- As we know the size of the type, pointer knows how many bytes it has to shift to encounter next object (usually used with array).
- Note that these raw pointer arithmetics are not recommended and error-prone.

Pointer Arithmetics

```
int arr[5] = { 1, 2, 3, 4, 5 };
int *iptr = arr;
for (int i = 0; i < 5; i++) {
std::cout << *(iptr + i) << " ";
}
```

Address space

- Each process (running program) has its own *address space*, where its code and data resides.
- Think of address space as a very large linear memory i.e. `char mem[]`.
- You can probe elsewhere in the address space using *pointer*s.
- When we declare local variables inside functions, the data is stored in the *stack*.
- Stack grows from higher address to lower address. If a function calls another function a new *stack frame* is created.
- Stack frame is destroyed if the functions returns.

Address space (Cont'd)

- Pointers are used to point location the address space. It holds address (from either stack or heap) and usually has associated type that tells which type of data is stored in the pointer
- If we create data within a function and return the address of created value in the stack, the program will crash.
- Instead we dynamically allocate memory in the *heap* and pass the pointer to the object allocated in the heap.
- We allocate new object in the heap using `new` and allocated memory should be deallocated by `delete` (or `delete[]`) for further reuse.