Mid2 -- Part III

Array

An ordered collection of data values of the same type.

- Declear an array

```
int array_1D[10]; // data_type variable_name[size];
int array_2D[5][5]; // 5*5 2D-array
```

- Initialize an array

```
// 1. initializer list
// 1D
int zeros[20] = {0};                        // All elements initialized as 0
int notAllZero[20] = {1};                    // Only the first one will be 1, other will be 0
int numbers[5] = {1, 2, 3, 4, 5};
int numbers[] = {1, 2, 3, 4, 5};                 // Equivalent to previous
// 2D
int zeroArray_2D[2][4] = {0};  // All elements initialized as 0
int nonZeroArray_2D[2][4] = {
  {10, 11, 12, 13},
  {14, 15, 16, 17}
};
int nonZeroArray[2][4] = {10, 11, 12, 13, 14, 15, 16, 17}; // Equivalent to previous
// 2. Use for loop
int a[5];
for(int i=0;i<5;++i)
{
  a[i]=i;
}
int b[5][5];
```

```c
for(int i=0;i<5;++i)
{
  for(int j=0;j<5;++j)
  {
      b[i][j]=...;
  }
}
// 3. Use memset for char or use memset to initialize all number to 0
int a[10];
memset(a,0,10*sizeof(int));
```

- Pass an array to a function

```c
// 1D case: 2 ways
void func_1D(int* array, int size);
void func_1D(int array[], int size);
// 2D case: The first array dimension does not have to be specified. The second (and any subsequent) dimensions must be given.
// Here M,N are global const or macro
void func_2D(int array[M][N]); // here the dimension of array that is passed into it must fit with M*N
void func_2D(int array[][N], int row);
```

Pointer

Variable that stores the address of another variable
*: obtain the value of the address
&: obtain the address of a variable
address is a long long int

```c
int x = 0;
int* px = &x;
printf("%p\n", px);  // print the address of x
printf("%d\n", *px);  // print the value of x
```

If you want to modify the value passed into a function, pass its reference or pass a pointer pointing to it (P186 example)

Dynamic Memory Allocation

Why dynamic allocation

- variable length array is dangerous (and if you use it, you may receive small deduction) eg. avoid writing something like

  ```
  int n;
  scanf("%d",&n);
  int a[n];
  ```

  reason: size of stack is limited...

  - Stack
    - Local variables are allocated in the stack
    - Once the block execution is finished, the stack pointer will be recovered to the status before the block execution, and the local variables will be destroyed
    - The size of the stack is limited, so do not declare a huge array as a local variable, which may result in "stack overflow"
    - In short, the memory in stack is assigned to variables when the variable is declared, and freed when the variable is destoried (out of scope).
  - Heap
    - The heap is a large pool of memory used for dynamic allocation. So when declare a huge array, we may prefer to put it in heap
    - The memory dynamically allocated in the heap must be freed manually
    - In short, the memory in heap is assigned and freed by the programmer manually. It means the programmer could control the life cycle of a variable, rather then collected by operatring system automatically. It can always be dangerous if the programmer doesn't handle it properly

How to allocate dynamic array

- `malloc(size_t bytes)`: reserve a contiguous block of memory in heap that may be uninitialized, and returns a pointer to the first address of the allocated region
- `calloc(size_t n_member, size_t size)`: initializes memory contents to zero, and returns a pointer to the first address of the allocated region

- `realloc(void *space, size_t bytes)`: resize an existing memory allocation that was previously allocated on the heap. It first deallocates previous region, then allocates memory having the new size and copies old content into the new region up to the lesser of old and new sizes. Finally, it returns a pointer to the first element of the allocated memory

```
// 1D version
int* a=(int *)calloc(n, sizeof(int));
a=realloc(a, 2*n*sizeof(int));
free(a);
// 2D version: allocate a n*n array
int** A=malloc(n*sizeof(int *));
for(int i=0;i<n;++i)
{
    A[i]=malloc(n* sizeof(int));
}
for(int i=0;i<n;++i)
    free(A[i]);
free(A);
```

Pass it to function

```
// 1D case
void func_1D(int* array, int size);
// 2D case
void func_2D(int** array, int row, int col);
```

Avoid misbehave

- case 1: free and realloc can only be done on the original pointer

```
int main()
{
  int* a=(int *)malloc(10* sizeof(int));
  a+=1; // misbehave
  a=realloc(a,8* sizeof(int));  // error occurs
  //free(a);
  return 0;
}
```

- case 2: your variable never knows the size of the allocated memory

```c
int main()
{
  int* a=(int *)malloc(10* sizeof(int));
  printf("%lu", sizeof(a));  // output 8
  free(a);
  return 0;
}
```

  - memory allocator: keeps track of which bytes are currently allocated and which are available for use (it knows the size)
  - the return variable of malloc or calloc itself cannot know the size
- case 3: avoid dangling pointer and lost memory

```c
int* a=(int *)calloc(10, sizeof(int));
int* b=(int *)calloc(10, sizeof(int));
a=b; // then the memory previously pointed by a is lost
free(a);
free(b); // this may cause double free error
// after free, a and b are dangling pointer
```

  - dangling pointer: pointers that pointing to a memory location that has been freed (set to NULL to get rid of dangling pointer)
  - memory leak: use `valgrind` to check memory leak

Array and pointer

General

Similarity:

- Array can be interpreted as the pointer pointing the first element
- `*(array+i)` is equivalent to `array[i]` and `array+i` is equivalent to `&array[i]`

Differences:

- Array (not dynamic array) is static, so it cannot be resized
- Array name is not a variable, but the pointer is

```c
int a[10];
int* p=a;
a++; // not valid
p++; // OK
```

C-style string

- As array
  - An character array ends with '\0' (0 in ASCII)
  - Always leave a space for '\0'

    ```c
    char country_name[4]="GER"; // in this case country_name[3]=='\0', resembles the end of the string
    char name[]="1234"; // you can find the length of this array is 5
    // do not write char country_name[3]="GER";
    ```

  - If your char array does not end with '\0', there might be some problem if you use functions like `strcat`, `strcmp`, `strcpy` etc.
- As pointer

```c
// avoid writing the following
char *var = "12345"; // because here we have not allocate memory for pointer var
// you can use
char country_name[4]="GER";
char* name=country_name;
// or
char* str = malloc(10 * sizeof(char));
memset(....);
```

Important exercises

hw6: ex4,ex5

Citation

- RC-week9-Checklist
- malloc