vg101: Introduction to Computer Programming

RC 5

CHEN Xiwen

2019/6/21

Arrays

An **ordered** collection of data values of the **same** type.

- Declaration: `type name[size] (= {values})`

```c
int d[10];
char str[10];
double f[10];
```

- Initialization `<demoArray.c>`

```c
int a[10];
int b[10] = {0};
int c[10] = {1};

int n[5];
for (int i = 0; i < 5; i++) {
    n[i] = i + 1;
}
```

Note: Values will not set to default if not initialized, which is different from MATLAB.

Q: Then how to perform initialization on a batch of elements?

=> `memset` (`char` array)

1. Library `#include <string.h>`
2. Function declaration:

```c
void *memset(void *dest, int ch, size_t, count);
/*
 dest: pointer to the destination;
 ch: fill byte;
 count: number of bytes to fill;
*/
```

3. Common usage `<demoMemset.c>`

```c
#include <stdio.h>
#include <string.h>

int main() {
```

```
    char s1[10];
    char s2[] = "memset is useful";

    memset(s1, '-', 9);
    memset(s2 + 7, '=', 2);

    printf("%d\n", s1[9] == '\0'); // 1

    puts(s1);
    puts(s2);
    return 0;
}
```

Note: `memset` counts data size in byte.

```
// a common usage;
char str[20];
memset(str, 0, 20 * sizeof(char));
```

Q: What is the output of the following, and why?

```
char str[20];
memset(str, 'a', 20 * sizeof(char));
```

- C-style string `<demoCharArray.c>`

```
#include <stdio.h>

int main() {
    // "Hello, world.\0";
    char s1[20] = "Hello, world.";
    char s2[20];
    int i;
    for (i = 0; i < 5; i++) {
        s2[i] = s1[i];
    }
    // s2[i] = '\0';
    puts(s2);
    return 0;
}
```

1. Represented as `char` array
2. Use `\0` at the end of the string
3. `\0` allows `printf` and `puts` to decide the end of the string
4. What is the consequence of ignoring `\0`?

- Accessing by index **---Do not exceed index bound!**
- Function argument

1. Argument in declaration: need to specify size.

```
void print_array(int a[], int size);
```

2. Outside function call: as above.

```
int a[10] = {0};
print_array(a, 10);
```

3. Pass by address: will modify original data if it is modified in the function call.
`<demoArrayFunc.c>`

```
#include <stdio.h>
// a: pass by address;
// size: pass by value;
void set_array(int a[], int size) {
    for (int i = 0; i < size; i++) {
        a[i] = i + 1;
    }
    // will not affect the value of size;
    size = 7;
}

int main() {
    int a[10] = {0};
    int size = 10;
    set_array(a, size);

    for (int i = 0; i < 10; i++) {
        printf("%d\n", a[i]);
    }
    printf("%d\n", size);
    return 0;
}
```

We will hopefully have a deeper understanding about address after discussing pointers.

- Two-dimensional array (matrix in MATLAB)
    1. Array of array
    2. Stored as one-dimensional array in memory
    3. Can be extended to higher-dimensional arrays

H3 Pointers

Accessing memory by address

- Address in memory: bytes
- What are the values of pointers?

```
#include <stdio.h>
```

```
2
3  int main() {
4      int a = 0;
5      float b = 0.;
6      char c = 'a';
7      char s[] = "string";
8
9      int* pa = &a;
10     float* pb = &b;
11     char* pc = &c;
12     char* ps = s;
13
14     printf("int: %p\nfloat: %p\nchar: %p\nstring: %p\n", pa, pb, pc,
   ps);
15
16     return 0;
17 }
```

- Reference `&` and dereference `*`

  Consider the following cases, what can we say about the following statements? (Declarations are omitted.) `<demoPtr.c>`

```
1  int* p;
2
3  p = v1;
4  *p = v2;
5  // &p = v3;
6
7  v4 = *p;
8  v5 = &p;
```

- Declaration

```
1  int x, y;
2  int* px, py; // px is `int*`, py is `int`
3  int *px, *py; // both `int*`;
```

- Assignment `int* p;`

  1. Assign by address: `p = &a;`
  2. Assign by value:
     1. **First:** `p = &a;`, **then** `*p = 4;`
     2. How about `int* p; *p = 3;`? `<demoPtr.c>`

  Understanding: What does computer do in declaration?

  1. Declare data type
  2. Calculate memory usage
  3. Allocate memory to the variable

Q: Case for pointers?

When declaring a pointer, the computer calculates the memory usage of the *pointer*, but does not allocate the memory usage of the data that the pointer points to. A pointer is an address afterwards. This also allows us to use dynamic memory allocation.

- `NULL` pointers
  1. `NULL == 0`
  2. Safe memory
  3. Cannot be dereferenced
  4. Used to check whether the data pointed by the pointer has been deleted.
  5. Good practice to assign a pointer to `NULL` after deleting the values it points to.
- Understanding arrays `<demoPtr.c>`
  1. The name of an array is a pointer, pointing to the first element of the array.

     ```
     1  // int* a;
     2  int a[5] = {1, 2, 3, 4, 5};
     3  int* b = &(a[0]);
     4  printf("%d\n", a == b);
     ```

  2. Arithmetic operations on pointers: because pointer is an address

     ```
     1  b = a + 3;
     2  printf("%d %d\n", *b, a[3] == *(a + 3));
     ```

  3. `a[index]` is equivalent to dereferencing a pointer `a`.
  4. `function(int a[], int size)` is equivalent to `function(int* a, int size)`
  5. `int a[10]`: `a` is `int*`, `int b[10][10]`: `b` is `int**`, `b[0]` is `int*`

H3 Dynamic Memory

- `malloc`, `calloc`

```
1  void* malloc(size_t size);
2  void* calloc(size_t num, size_t size);
```

1. `void*` means pointer to "any type" `<demoAlloc.c>`

   e.g.,

   ```
   1  typedef struct Complex {
   2      double imag;
   3      double real;
   4  } complex_t;
   5
   6  int* pm1 = (int*)malloc(2 * sizeof(int));
   7  // same as
   8  //      int* pm1 = malloc(2 * sizeof *pm1);
   9  complex_t* pm2 = (complex_t*)malloc(2 * sizeof(complex_t));
   ```

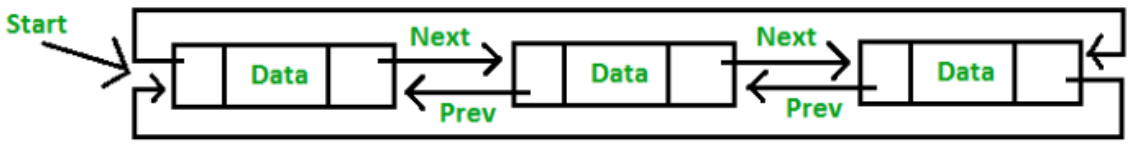
```
complext_t** pm3 = (complex_t**)malloc(2 * sizeof(complex_t*));

int pc1 = (int*)calloc(2, sizeof(int));
complex_t* pc2 = (complex_t*)calloc(2, sizeof(complex_t));
complex_t** pc3 = (complex_t**)calloc(2, sizeof(complex_t*));

// IMPORTANT!
free(pm1);
free(pm2);
free(pm3);
free(pc1);
free(pc2);
free(pc3);
```

- `realloc`

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

H3 Linked Lists

Application of pointers.



1. How to initialize an empty list?

   Create a single node with `next` and `prev` pointers pointing to itself.

2. How to insert element?

   Set:

   - New node's `next` pointer point to the next node.
   - New node's `prev` pointer point to the previous node.
   - Previous node's `next` pointer point to the new node.
   - Next node's `prev` pointer point to the new node.

   Does the sequence of the updates above matter?

3. How to remove element?

   Set:

   - Previous node's `next` pointer point to the deleted node's `next` node.
   - Next node's `prev` pointer point to the deleted node's `prev` node.
   - Deleted node's `prev` and `next` pointers point to itself.

   Does the sequence of the updates above matter?