Coding Review (I2P 2019)

Basic Array and Pointers in C v1.0

Review of array and pointers.

Structure

We'll start from reviewing some basics about structures.

If we want to make a 2D game with different objects, we may need to have its position coordinates and size stored in arrays. Something like:

```
float enemy_x[MAX];
float enemy_y[MAX];
float enemy_w[MAX];
float enemy_h[MAX];
float player_x;
float player_y;
float player_w;
float player_h;
```

However it gets really confusing once the variable count increases. So Structures come into rescue:

```
struct object {
    float x, y, w, h;
};

struct object enemy[MAX];
struct object player;
```

and then, we can access the coordinates easily like:

```
player.x = 0;
```

The `struct object` declaration is quite long and kind of redundant, so we can use `typedef`:

```
struct object {
    float x, y, w, h;
};
typedef struct object Object;
```

```
Object enemy[MAX];
Object player;
```

We can even combine the declaration of `struct` with `typedef`.

```
typedef struct object {
    float x, y, w, h;
} Object;
Object enemy[MAX];
Object player;
```

and even omit the `object` type:

```
typedef struct {
    float x, y, w, h;
} Object;
Object enemy[MAX];
Object player;
```

By using Structures, we can define a special type for variable declaration afterwards. More advanced usages such as bit-fields, union, enum can provide more flexible controls on Structures.

Arrays

For using static arrays, we can only declare it with constant length.

1. Static array initialization

   What is the result of:

   ```
   #include <stdio.h>
   #define MAX 3

   int a[MAX];
   int b[MAX] = {};
   int c[MAX] = {0};
   int d[MAX] = {1};
   int e[MAX] = {1, 1, 1};

   int main(void) {
       for (int i = 0; i < MAX; i++)
           printf("%d ", a[i]);
       for (int i = 0; i < MAX; i++)
           printf("%d ", b[i]);
       for (int i = 0; i < MAX; i++)
           printf("%d ", c[i]);
       for (int i = 0; i < MAX; i++)
   ```

```
        printf("%d ", d[i]);
    for (int i = 0; i < MAX; i++)
        printf("%d ", e[i]);
    return 0;
}
```

Answer: (Write your answer here!)

2. Multi-dimensional Static Array

   What is the result of:

```
#include <stdio.h>
#define N 3
#define M 2

int a[N][M] = {{1}, {2, 3}};

int main(void) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            printf("a[%d][%d] is %d\n", i, j, a[i][j]);
        }
    }
    return 0;
}
```

   Answer: (Write your answer here!)

3. Variable-Length Arrays (VLA)

```
#include <stdio.h>
int main(void) {
    int N;
    scanf("%d", &N);
    int a[N];
    return 0;
}
```

   You may see some code that declares an array like above, where `N` is not a constant value. This is called Variable-Length Array, and is only supported after C99. There are some drawbacks for using VLAs, to avoid confusion, you can use the dynamic array as shown in the next section.

Pointers

Misusing pointers can pose a serious security threat to programs. You'll often see Buffer Overflow security fixes in Google Chrome, Mozilla Firefox, or any other applications. These security issue are mostly caused by pointers. Thus, in many high-level languages, the usage of pointer is forbidden.

If pointers are so dangerous, why pointers are used in the first place? There's some cases that requires pointers:

- Function returning more than one value
- Send arguments more efficiently
- Access dynamic allocated memory
- Implement data structures like linked lists, trees, etc.
- Call functions through a variable
- etc.

1. Function returning more than one value

   Before starting let's take a look at the snippet below:

   ```
   int a, b;
   a = 1;
   b = a;
   ```

   For the second line, `a` is at the left hand side of the assignment operator. The value `1` is written to `a`'s memory address.

   For the third line, `a` is at the right hand side of the assignment operator. The value of `a` is read from its memory address (I'll ignore registers and cache here to avoid complexity) and written to `b`'s memory address.

   If we want to change `a`'s value through a function call, we can only:

   ```
   int a;
   a = func_a();
   ```

   But if we want to change multiple variables' value, we cannot do something like:

   ```
   a, b = func()
   ```

   in Python. Instead, we need to find a way to send its address into the function, and let the function's code change its value. But something like:

   ```
   int a, b;
   func(a, b);
   ```

   doesn't work since the value are read from the memory addresses before sending into the function, if we change the variables inside the function, it only changes the value of the local variables inside that function. So, we send their memory addresses instead:

```
int a, b;
func(&a, &b);
```

which is how `scanf` does its work, and the return value can now be used to indicate whether the function runs successfully.

In the case above, we wanted to change multiple variables' value. What if the value we want to change is a memory address?

```
int* a;
a = malloc(sizeof(int));
free(a);
```

If we don't want to use the return value of the function, then it'll become something like:

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

void my_malloc(int** a, size_t size) {
    *a = malloc(size);
}

int main(void) {
    int* a;
    my_malloc(&a, sizeof(int));
    free(a);
    return 0;
}
```

Sending the pointer `a` doesn't work here since we can only change the value in `a`'s memory address. But we wanted to change `a`'s memory address itself. So we need to send `a`'s memory address into the function.

This double pointer allocation is commonly used in CUDA programming, and also used a lot when interacting with Windows APIs.

2. Send arguments more efficiently

   What's the output of:

```
#include <stdio.h>
#define MAX 100

typedef struct {
```

```
    int content[MAX];
} LargeArray;

void func(LargeArray la) {
    printf("sizeof la is %zu\n", sizeof la);
}

void funcp(LargeArray* pla) {
    printf("sizeof pla is %zu\n", sizeof pla);
    printf("sizeof la is %zu\n", sizeof *pla);
}

int main(void) {
    LargeArray la;
    func(la);
    funcp(&la);
    return 0;
}
```

Answer: (Write your answer here!)

3. Access dynamic allocated memory

   We can allocate dynamic resources for later usages. For example a 1d dynamic array:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

int main(void) {
    int* a;
    a = malloc(sizeof(int) * MAX);
    free(a);
    return 0;
}
```

Remember to `free` any `malloc`ed resources is a good coding practice. Forgetting to free allocated resources result in memory leak, which is really bad for products and applications.

For making games, we may want to allocate some resources when doing certain operations, and this operation is repetitively called. If we forgot to free these resources, the program will keep asking more memory spaces from the OS, and finally use up all memory and crashes. (Sometimes the OS kills the program, but sometimes the OS crashes due to Out Of Memory issue (OOM)) A dangerous code is shown below, which may crash your OS:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000000
```

```c
void game_update() {
    int* a;
    a = malloc(sizeof(int) * MAX);
    // Do something...
    // and forget to free this resource.
    // free(a);
}

int main(void) {
    while (1) {
        game_update();
    }
    return 0;
}
```

4. Implement data structures like linked lists, trees, etc.

   Linked List:

```c
typedef struct _node {
    int value;
    struct _node* next;
} Node;
```

   Binary Tree:

```c
typedef struct _node {
    int value;
    struct _node *left, *right;
} Node;
```

5. Call functions through a variable

   What is the result of:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000000

void add(int a, int b) {
    printf("add : %d\n", a+b);
}
void mult(int a, int b) {
    printf("mult: %d\n", a*b);
}

int main(void) {
```

```
    void (*fptr) (int, int);
    fptr = &add;
    fptr(2, 3);
    fptr = &mult;
    fptr(2, 3);
    return 0;
}
```

Answer: (Write your answer here!)

More on Pointers and Array

Arrays is not equal to Pointers, they are alike but not the same. But most of the time, they can be used in the same way.

This part is the most interesting part, we'll start from reviewing basic operators.

Operator review:

- `&var` returns `var`'s memory address.
- `*ptr = c` assign `c` to the address where `ptr` is pointing.
- `a = *ptr` takes out the value where `ptr` is pointing and store it in `a`.
- `ptr++` Move the pointer forward for `sizeof(*ptr)`-byte, that is, to move n-bytes, where n is the size of the type that `ptr` is pointing to.

  e.g. `int* ptr`, if `ptr == 0x00`, `ptr+1 == 0x04`, assuming `sizeof(int)` is 4.
- `&a` is an alias of `&a[0]`.
- `arr[i]` is an alias of `*(a+i)`.

  So, `i[arr]` is actually valid!
- `arr[i][j]` is an alias of `*(*(a+i)+j)`.
- `struct_ptr->var` is an alias of `(*struct_ptr).var`.

1. Size of Pointers and Arrays.

   What is the output of:

```
#include <stdio.h>
#include <stdlib.h>
#define N 2
#define M 3

int main(void) {
    // Declaration
    int i, j, *tmp;
    int a[N*M];
```

```c
    int b[N][M];
    int c[M][N];
    int *pa = malloc(sizeof(int) * M * N);
    int **pb = malloc(sizeof(int*) * N);
    int **pc = malloc(sizeof(int*) * M);
    // Allocate second level resources all at once to reduce allocation
overhead.
    tmp = malloc(sizeof(int) * M * N);
    for (i = 0; i < N; i++)
        pb[i] = tmp + i * M;
    tmp = malloc(sizeof(int) * M * N);
    for (i = 0; i < M; i++)
        pc[i] = tmp + i * N;
    // Print
    printf("sizeof(int)       : %2zu\n", sizeof(int));
    printf("sizeof(int[M])    : %2zu\n", sizeof(int[M]));
    printf("sizeof(int[N])    : %2zu\n", sizeof(int[N]));
    printf("sizeof(int[N][M]) : %2zu\n", sizeof(int[N][M]));
    printf("sizeof(int[M][N]) : %2zu\n", sizeof(int[M][N]));
    printf("sizeof(void*)     : %2zu\n", sizeof(void*));
    printf("sizeof(int*)      : %2zu\n", sizeof(int*));
    printf("sizeof(int**)     : %2zu\n", sizeof(int**));
    printf("sizeof   a        : %2zu\n", sizeof a);
    printf("sizeof  *a        : %2zu\n", sizeof *a);
    printf("sizeof   b        : %2zu\n", sizeof b);
    printf("sizeof  *b        : %2zu\n", sizeof *b);
    printf("sizeof **b        : %2zu\n", sizeof **b);
    printf("sizeof   c        : %2zu\n", sizeof c);
    printf("sizeof  *c        : %2zu\n", sizeof *c);
    printf("sizeof **c        : %2zu\n", sizeof **c);
    printf("sizeof   pa       : %2zu\n", sizeof pa);
    printf("sizeof  *pa       : %2zu\n", sizeof *pa);
    printf("sizeof   pb       : %2zu\n", sizeof pb);
    printf("sizeof  *pb       : %2zu\n", sizeof *pb);
    printf("sizeof **pb       : %2zu\n", sizeof **pb);
    printf("sizeof   pc       : %2zu\n", sizeof pc);
    printf("sizeof  *pc       : %2zu\n", sizeof *pc);
    printf("sizeof **pc       : %2zu\n", sizeof **pc);
    // Free
    free(pa);
    free(pb[0]);
    free(pb);
    free(pc[0]);
    free(pc);
    return 0;
}
```

Answer: (Write your answer here!)

2. Array as Arguments

   What is the output of:

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

void func1(int a[]) {
    printf("%zu\n", sizeof a);
}
void func2(int *a) {
    printf("%zu\n", sizeof a);
}

int main(void) {
    int a[100];
    func1(a);
    func2(a);
    return 0;
}
```

Answer: (Write your answer here!)

3. Pointer Compatibility

```c
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2;
int *arp[3];
```

Which of the operations below are valid?

```c
pt = &ar1[0][0];
pt = ar1[0];
pt = ar1;
pa = ar1;
pa = ar2;
p2 = &pt;
*p2 = ar2[0];
p2 = ar2;
pa = arp;
```

Answer: (Write your answer here!)

Constants

- `const int* ptr` is the same as `int const* ptr`, which protects the value pointed.

```
*ptr = 0; // Not allowed
ptr = &a; // Allowed
```

- `int * const ptr`, which protects the pointer variable itself.

```
*ptr = 0; // Allowed
ptr = &a; // Not allowed
```

- `const int * const ptr` is the same as `int const * const ptr`.

```
*ptr = 0; // Not allowed
ptr = &a; // Not allowed
```

Interpretation on Syntaxes

- Arrays

  For `int a[2][3]`, it can be interpret from the inside to the outside. So `a` is an array with 2 cells. Each of the cell contains a int[3] element, that is, an array with 3 cells storing int.

- Pointers

  You may interpret it in 2 ways:

  1. `int* ptr`, `ptr` is a variable, and its type is `int*`, that is, a variable storing memory address.
  2. `int (*ptr)`, `ptr` is a pointer, and the memory address represents a `int`.

  For `int (*pa)[3]`, it can also be interpret from the inside to the outside. So `pa` is a pointer, pointing to a int[3] element.

  For `int *ap[3]`, it is an array of 3 cells. Each cell stores a int*, that is, a pointer-to-int.

There are much more tricky problems on pointers and arrays, but by thinking hard on the questions above and understanding the underlying reason of the results should be quite enough.

If you have trouble understanding a C pointer statement, try out this website:

- cdecl

This should be the last assignment for this semester, hope you have better understandings of C language through finishing these six assignments.

Epilogue

If you feel frustrated when trying to understand these concept, it should be a good sign. Since it means that you reinforced these important concepts that cannot be learned just by writing codes for NTHU Online Judge.

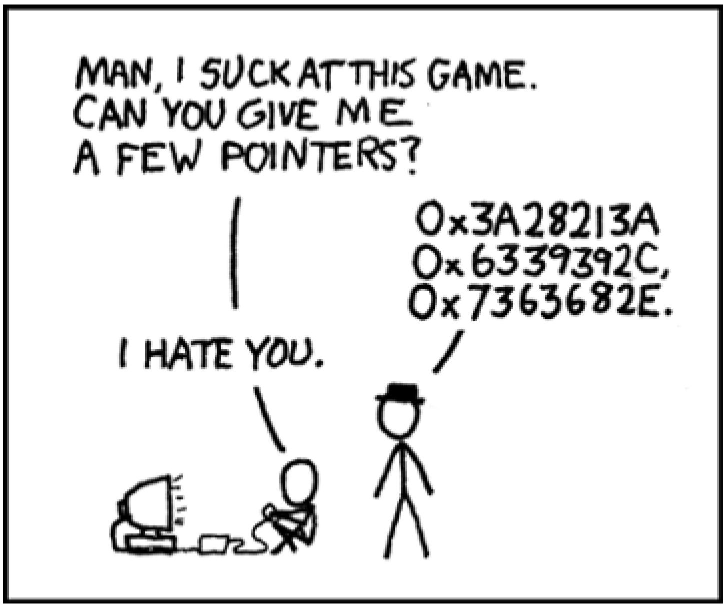

Photo Credit: Pointers on XKCD.

This is the first time for Introduction to Programming course to include these written assignments, so if there's any typo, please discuss on iLMS or email `j3soon@gapp.nthu.edu.tw`, I'll appreciate it.

If I'm still the TA for next semester, maybe we can try to reinforce the understanding of C++ as well... There are much more interesting concepts in OOP, Design Patterns, Templates... Anyway, good luck on your exam.