Advanced Pointers

Nguyen Le Hoang Dung
nlhdung@fit.hcmus.edu.vn

Contents

- Lecture: Pointers and Dynamic Memory
  - Review
  - Dynamically allocating structures
  - Combining the notion of structures and pointers
  - Pointer of Pointer
  - Other types of pointers

Contents

- Lecture: Pointers and Dynamic Memory
  - **Review**
  - Dynamically allocating structures
  - Combining the notion of structures and pointers
  - Pointer of Pointer
  - Other types of pointers

Review of Pointers

- What is a pointer?
- How would you define a pointer variable, that can point to a float?
- Would this change if you wanted the pointer to reference an array of floats?
- Show how to dynamically allocate an array of 20 floats
- Show two ways of accessing element 19

Review of Pointers

- What operator allocates memory dynamically?
- What does it really mean to allocate memory? Does it have a name?
- Why is it important to subsequently deallocate that memory?
- What operator deallocates memory?

Contents

- Lecture: Pointers and Dynamic Memory
  - Review
  - **Dynamically allocating structures**
  - **Combining the notion of structures and pointers**
  - Pointer of Pointer
  - Other types of pointers

Dynamic Structures

- Let's take these notions and apply them to dynamically allocated structures
- What if we had a video structure, how could the client allocate a video dynamically?

```
video *ptr = new video;
```

- Then, how would we access the title?

```
*ptr.title
```
? Nope! WRONG

Dynamic Structures

- To access a member of a struct, we need to realize that there is a “precedence” problem.
- Both the dereference (*) and the member access operator (.) have the same operator precedence....and they associate from right to left
- So, parens are required:

```
(*ptr).title
```

Correct (but ugly)

Dynamic Structures

- A short cut (luckily) cleans this up:

`(*ptr).title` Correct (but ugly)

Can be replaced by using the indirect member access operator (->) ... it is the dash followed by the greater than sign:

`ptr->title` Great!

Dynamic Structures

- Now, to allocate an array of structures dynamically:

```
video *ptr;
ptr = new video[some_size];
```

- In this case, how would we access the first video's title?

```
ptr[0].title
```

  - *Notice that the -> operator would be incorrect in this case because ptr[0] is not a pointer variable. Instead, it is simply a video object. ptr is a pointer to the first element of an array of video objects*

Dynamic Structures

- Now, to allocate an array of structures dynamically:

```
video *ptr;
ptr = new video[some_size];
```

- In this case, how would we access the first video's title?

```
ptr[0].title
```

  - *Notice that the -> operator would be incorrect in this case because ptr[0] is not a pointer variable. Instead, it is simply a video object. ptr is a pointer to the first element of an array of video objects*

Dynamic Structures

- What this tells us is that the -> operator expects a pointer variable as the first operand.
  - In this case, ptr[0] is not a pointer, but rather an instance of a video structure. Just one of the elements of the array!
  - the . operator expects an object as the first operand...which is why it is used in this case!

Dynamic Structures

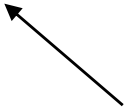
- Ok, what about passing pointers to functions?
- Pass by value & pass by reference apply.
  - Passing a pointer by value makes a copy of the pointer variable (i.e., a copy of the address).
  - Passing a pointer by reference places an address of the pointer variable on the program stack.

Dynamic Structures

- Passing a pointer by value:

```
video *ptr = new video;
display(ptr);
```

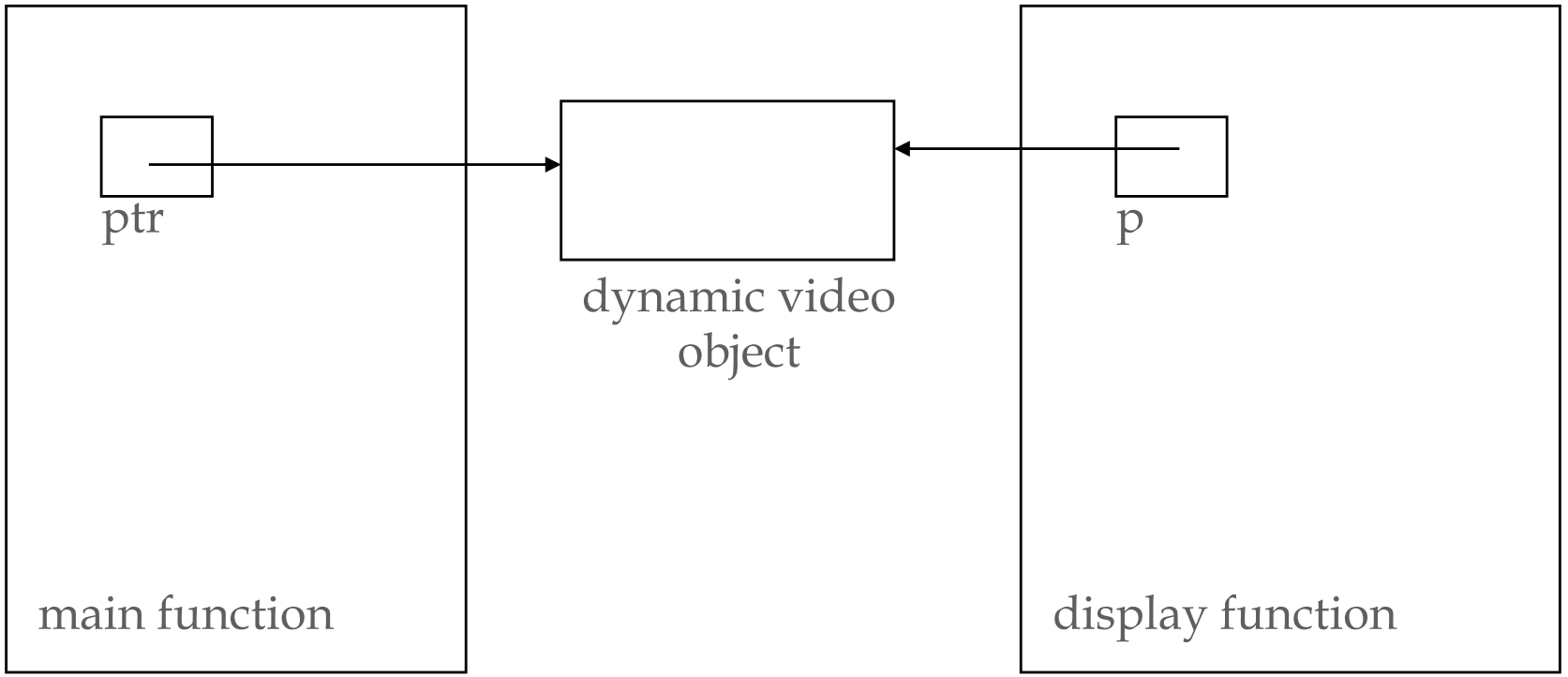
```
void display(video * p) {
  cout <<p->title <<endl;
}
```



p is a pointer to a video object, passed by value. So, p is a local variable with an initial value of the address of a video object

Dynamic Structures

- Here is the pointer diagram for the previous example:



Dynamic Structures

- Passing a pointer by reference allows us to modify the calling routine's pointer variable (not just the memory it references):

```
video *ptr;     set(ptr);  cout <<ptr->title;
```

```
void set(video * & p) {
   p = new video;
   cin.get(p->title,100,'\n');
   cin.ignore(100,'\n');
}
```

The order of the * and & is critical!

Dynamic Structures

- But, what if we didn't want to waste memory for the title (100 characters may be way too big (Big, with Tom Hanks)
- So, let's change our video structure to include a dynamically allocated array:

```
struct video {
   char * title;
   char category[5];
   int quantity;
};
```

Dynamic Structures

- Rewriting the set function to take advantage of this:

```
video *ptr;     set(ptr);
```

```
void set(video * & p) {
   char temp[100];
   cin.get(temp,100,'\n');
   cin.ignore(100,'\n');
   p = new video;
   p->title = new char[strlen(temp)+1];
   strcpy(p->title,temp);
}
```

watch out for where the +1 is placed!

Dynamic Structures

- But, what about that list of videos discussed earlier this term?
- Let's write a program to allocate this list of videos dynamically, at run time
- This way, we can wait until we run our program to find out how much memory should be allocated for our video array

Contents

- Lecture: Pointers and Dynamic Memory
  - Review
  - Dynamically allocating structures
  - Combining the notion of structures and pointers
  - **Pointer of Pointer**
  - Other types of pointers

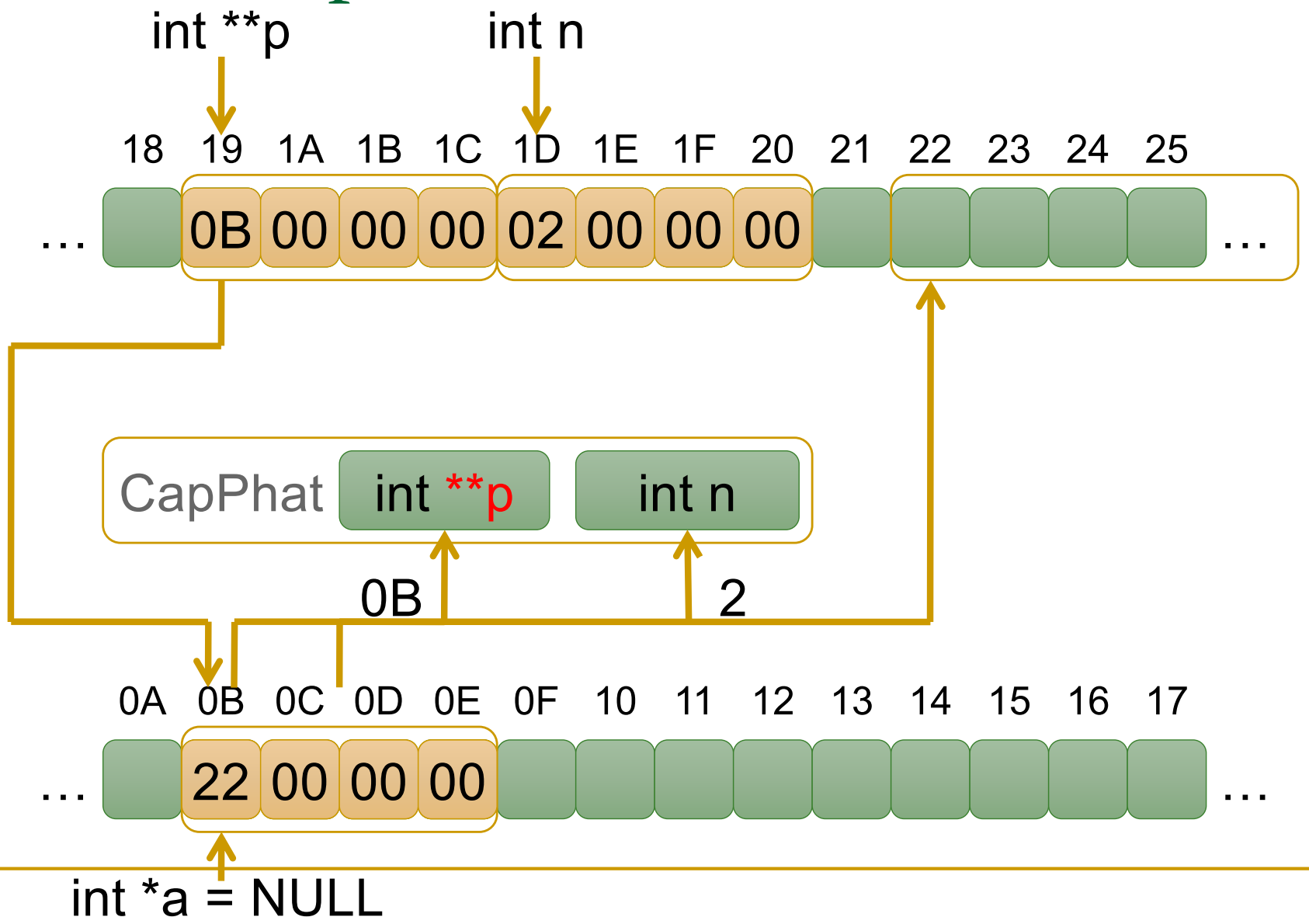
Pointer of pointer

- Address of pointer:
  - Variable has an address.
    - ***int*** has address ***int****
  - Pointer also has an address.
    - ***int**** has address type?
- Pointer of pointer:
  - A variable stores address of another pointer.
  - Declaration: *<pointer type> * <pointer name>;*

Pointer of pointer

- Declaration:
  - use *
- Initialization:
  - Use NULL
  - Use & operator

Pointer of pointer



Pointer of pointer

- Lưu ý

```
int x = 12;
int *ptr = &x;                  // OK
int k = &x; ptr = k;            // Lỗi

int **ptr_to_ptr = &ptr;        // OK
int **ptr_to_ptr = &x;          // Lỗi

**ptr_to_ptr = 12;              // OK
*ptr_to_ptr = 12;               // Lỗi

printf("%d", ptr_to_ptr);       // Địa chỉ ptr
printf("%d", *ptr_to_ptr);      // Giá trị ptr
printf("%d", **ptr_to_ptr);     // Giá trị x
```

Pointer of pointer

- Access memory content:
  - 1-level access: operator *
  - 2-level access: operator **
- Passing argument:
  - Pass-by-value.
  - Pass-by-reference.

=> Which values are changed in foo() ?

```
void foo(int **g, int** &h)
{
    (**g)++; (*g)++; g++;

    (**h)++; (*h)++; h++;
}

int main() {
    int a[10];
    int *p = a;
    int **q = &p;
    int **r = &p;

    foo(q, r);

    return 0;
}
```

Pointer of pointer

- Dynamic matrix:
  - Array of pointers:
    - Level-1 pointer is 1-dimensional dynamic array.
    - Level-2 pointer is 2-dimensional dynamic array.

```
void inputMatrix(int** &m, int& rows, int& cols)
{
    cout << "Input rows and cols: ";
    cin >> rows >> cols;

    m = new int*[rows];
    for (int i=0; i<rows; i++) {
        m[i] = new int[cols];
        for (int j=0; j<cols; j++) {
            cin >> m[i][j];
        }
    }
}
```

```
int main(int argc,
         const char * argv[]) {
    int **m;
    int rows, cols;

    inputMatrix(m, rows, cols);

    delete []m;

    return 0;
}
```

Contents

- Lecture: Pointers and Dynamic Memory
  - Review
  - Dynamically allocating structures
  - Combining the notion of structures and pointers
  - Pointer of Pointer
  - **Other types of pointers**

Other types of pointer

- Constant pointer:
  - Pointer points to only 1 address “for life”.
  - Declaration: ***<type> * const <pointer name>***

```
int x = 5, y = 6;
int * const p = &x;
p = &y; // Wrong.
```

  - The **name** of the **array** A is a **constant pointer** to the first element of the **arrays**

Other types of pointer

- Pointer to constant:
  - Memory content pointer points to cannot be changed.
  - Declaration: ***const <type> * <pointer name>***

```
int x = 5;
const int *p = &x;
*p = 6; // Wrong
```

Other types of pointer

- void pointer:
  - Pointer can store address of any types.
  - Declaration: ***void * <pointer name>***
  - Cast to specific type when accessing content.

```
void printList(void* p, int size){
    int *q = (int*)p;
    for (int i=0; i<size; i++) {
        cout << q[i] << endl;
    }
}
```

```
int main() {
    int a[10] = {1,2,3,4,5};
    int n = 5;

    printList(a, n);

    return 0;
}
```

Other types of pointer – Function pointer

- Function address:
  - Functions are also stored in memory.
  - Each function has an address.
- Function pointer stores address of function.
- Declaration:
  - <return type> (* <pointer name>) (<arguments>);
  - typedef <return type> (* <alias>) (<arguments>);
    <alias> <pointer name>;
- Functions have same address type if:
  - Same return type.
  - Same arguments.

Other types of pointer – Function pointer

```
typedef int (*funcPointer) (int a, int b);

int add(int a, int b) {
    return a+b;
}
int mul(int a, int b) {
    return a*b;
}

int cal(int m, int n, funcPointer p) {
    return p(m,n);
}
```

Other types of pointer – Function pointer

```
int main(int argc, const char * argv[]) {
    int x = 9;
    int y = 10;

    funcPointer p = add;
    cout << cal(x, y, p) << endl;

    p = mul;
    cout << cal(x, y, p) << endl;

    if (p == &mul) {
        cout << "Mul function";
    }

    return 0;
}
```

Other types of pointer – Function pointer

```
int (*getOperation(char op)) (int a, int b)  {
    if(op  == '*')
        return mul;
    if(op  == '+')
        return &add;
    return NULL;
}

funcPointer getOperation2(char op)  {
    if(op  == '*')
        return mul;
    if(op  == '+')
        return &add;
    return NULL;
}
```

Other types of pointer

- Pointer to fix-sized memory:
  - Address of static array:
    - What address type of static array?

```
int a[10];
int *p = a // p and a store address of a[0]
p = &a[0] // p and a store address of a[0]
??? q = &a;
```

Other types of pointer

- Pointer to fix-sized memory:
  - Pointer stores address of static array.
  - Declaration:

    <array type> (*<pointer name>)[<array size>];

```
int arr[10] = {1,2,3};
int *pointer = arr;
cout << pointer[0] << endl;

int (*pointer2)[10] = &arr;
cout << *pointer2[0] << endl;
```

Other types of pointer – Pointer to fix-sized memory

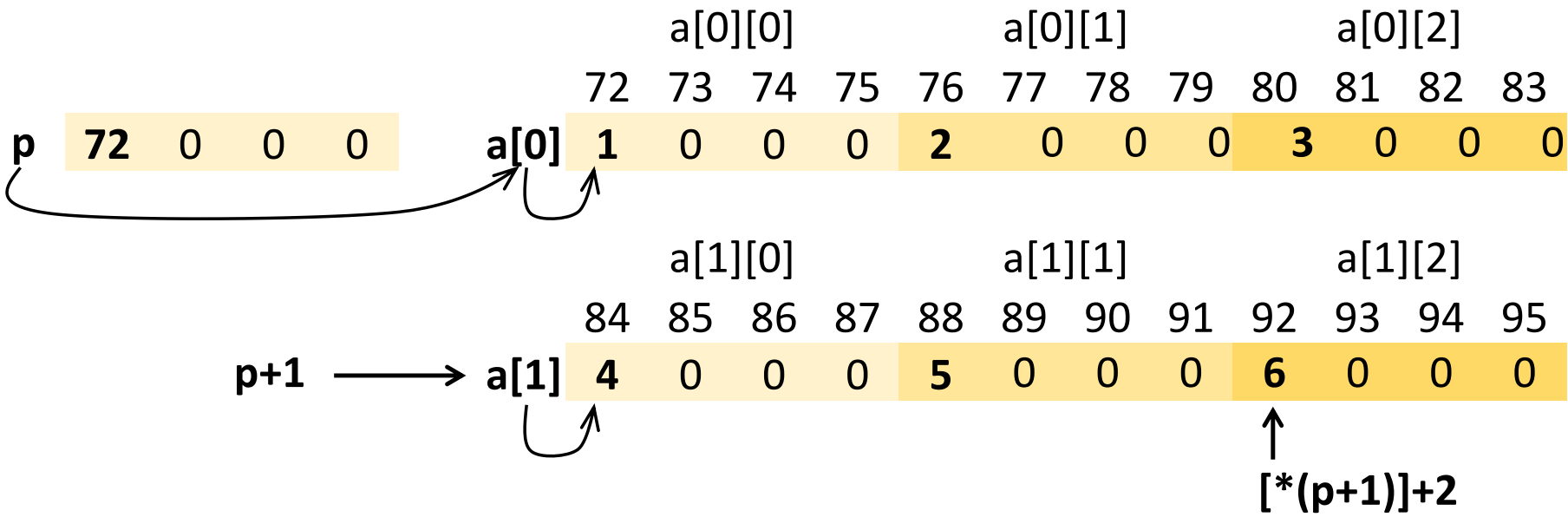
- Static 2-D array in C:
  - Is pointer to fix-sized 1-D array.
  - Stores address of the first row.

```
int main(int argc, const char * argv[]) {
    int a[3][4] = {{1,2,3},
                   {4,5,6}};
    a[0][0]=10; *a[0] = 10; **a = 10;

    a[1][0] = 40; *a[1] = 40; **(a+1) = 40;
    a[1][2] = 60; *(a[1]+2) = 60; *(*(a+1)+2) = 60;

    int (*p)[4] = a;
    cout << *(*(p+1)+2);
    return 0;
}
```

Other types of pointer – Pointer to fix-sized memory



Summary

- Dynamically allocating structures
- Combining the notion of structures and pointers
- Pointer of Pointer
- Other types of pointers

Practice 1

- Find the errors in the following code if any

```
int *x;
*x=100;
```

```
int x[3][12];
int *ptr[12];
ptr = x;
```

```
int a[] = {10, 20, 30, 40, 50};
for(int j=0; j<5; j++)
{
    cout << *(a++) << endl;
}
```

```
int **c = &c;
int **c = &*c;
int **c = **c;
```

Practice 2

- Given static 2-D array as follow:

```
int m[4][6];
```

- What types of addresses of the following variables?
  - a) m[1][3].
  - b) m[0].
  - c) m.
- Write code to access m[2][4] without using operator [ ]

Practice 3

- Write C program (use dynamic matrix) to do the followings:
  - Enter from keyboard matrix of M x N integers.
  - Get a list of primes from the input matrix.
  - Print the prime list to screen.

Practice 4

- Using **pointer** to write a program that can do the following functions:
  - Input for 10 integer number in array from keyboard,
  - Sort that array as ascending order.
  - Display that array.
  - Use function pointer that users can choose to sort that array in either ASC or DESC order.

References

- Kỹ thuật lập trình, Khoa CNTT, ĐHKHTN
- C++ Primer Plus, Stephen Prata, SAMS
- C++ Programming: An Object-Oriented Approach, 1st Edition, Behrouz A. Forouzan, Richard Gilberg, Hill Education, 2019
- Adam Drozdek, Data Structures and Algorithms in C++, 4th Edition, Cengage Learning, 2008
- Introduction to C++ Programming Concepts and Applications, John Keyser, Texas A&M University, 2019
- Nguyen Minh Huy, Programming Techniques and Practices slides, HCMUS
- Dang Binh Phuong, Programming Techniques and Practices slides, HCMUS