

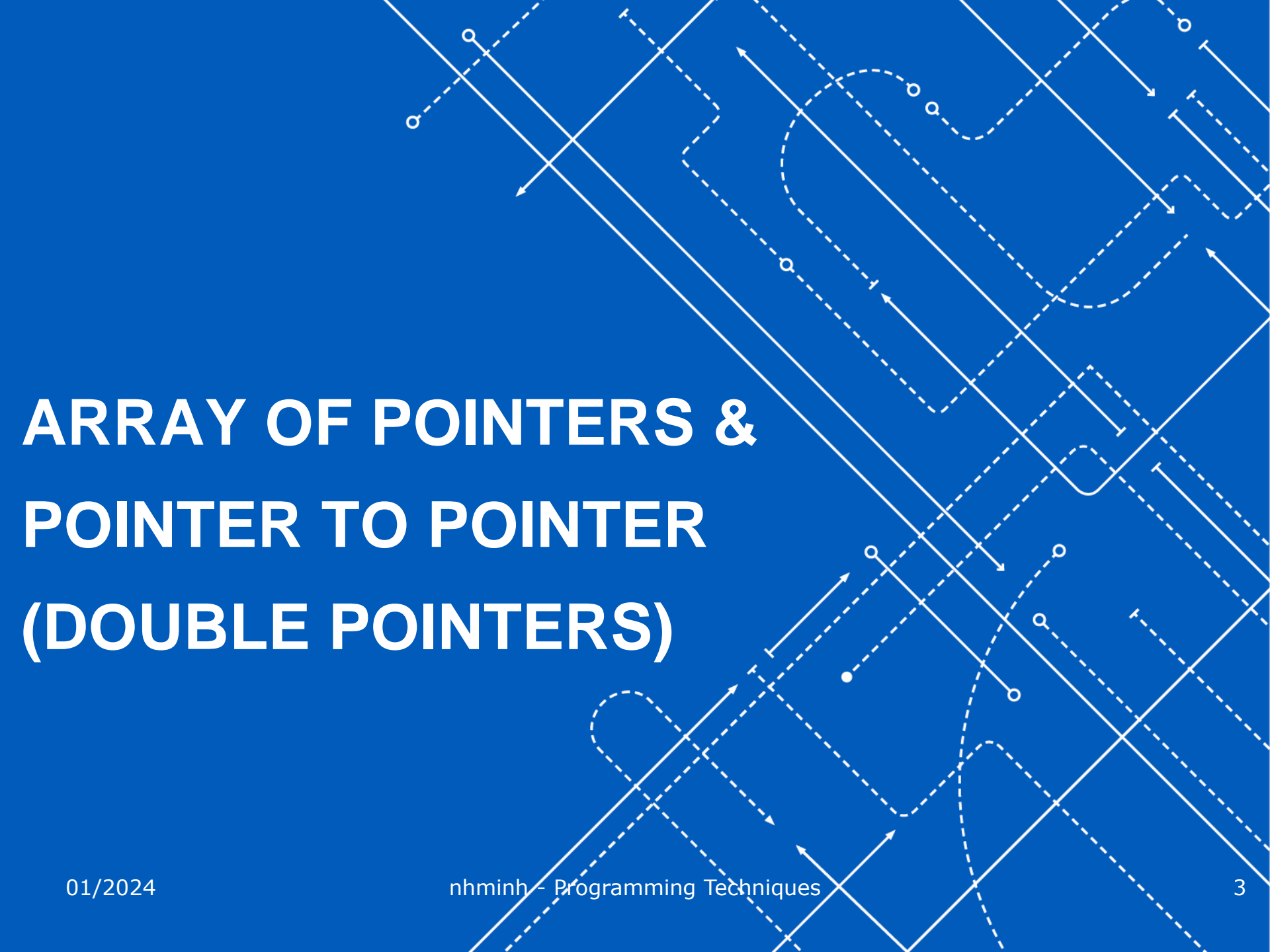
# PROGRAMMING TECHNIQUES

## Week 3: Pointers and Dynamic Memory (3)

# Content

---

- Array of Pointers vs Pointer to Pointer (Double Pointers)
- Pointers to Functions

The background of the slide is a solid blue color. Overlaid on this background is a complex, abstract pattern of white lines and arrows. The pattern consists of numerous straight lines of varying lengths and orientations, some of which are dashed. These lines are interconnected by arrows, some of which are also dashed. The overall effect is a sense of movement and connectivity, reminiscent of a network diagram or a stylized map. The lines and arrows are scattered across the entire slide, with a higher density in the upper right and lower right areas.

# ARRAY OF POINTERS & POINTER TO POINTER (DOUBLE POINTERS)

# Pointer to Pointer (Double Pointers)

- Pointer is a type of variable that *points to another variable or object*.
- So, can we use a pointer to store address of another pointer?

```
int* ptr1;
int* ptr2 = &ptr1;
```

type int \*\*

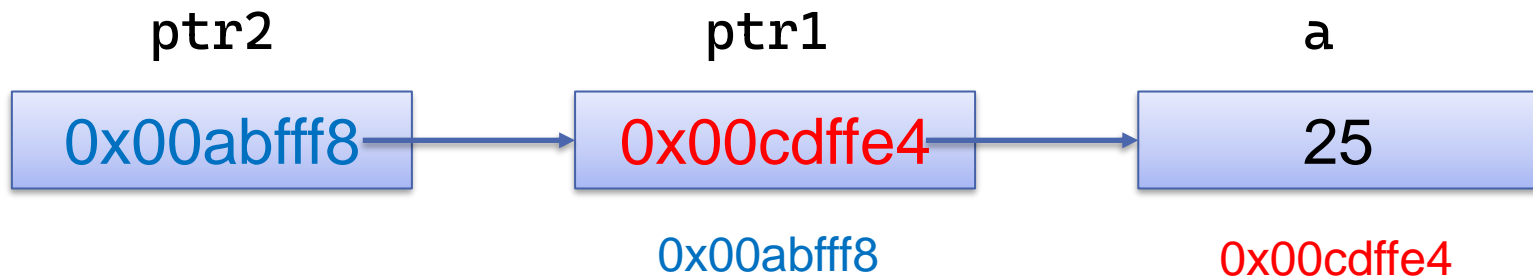
→ Error: a value of type “int \*\*” cannot be used to initialize an entity of type “int \*”

- We need to use Pointer to Pointer (Double Pointers)

# Pointer to Pointer (Double Pointers)

- Double pointers are pointers that *point to other pointers*. In other words, they are pointers that store the memory address of a pointer variable.

```
int a = 25;
int* ptr1 = &a;
int** ptr2 = &ptr1;
```



# Array of Pointers vs Double Pointers

- What is the difference between `arr` and `ptr`?

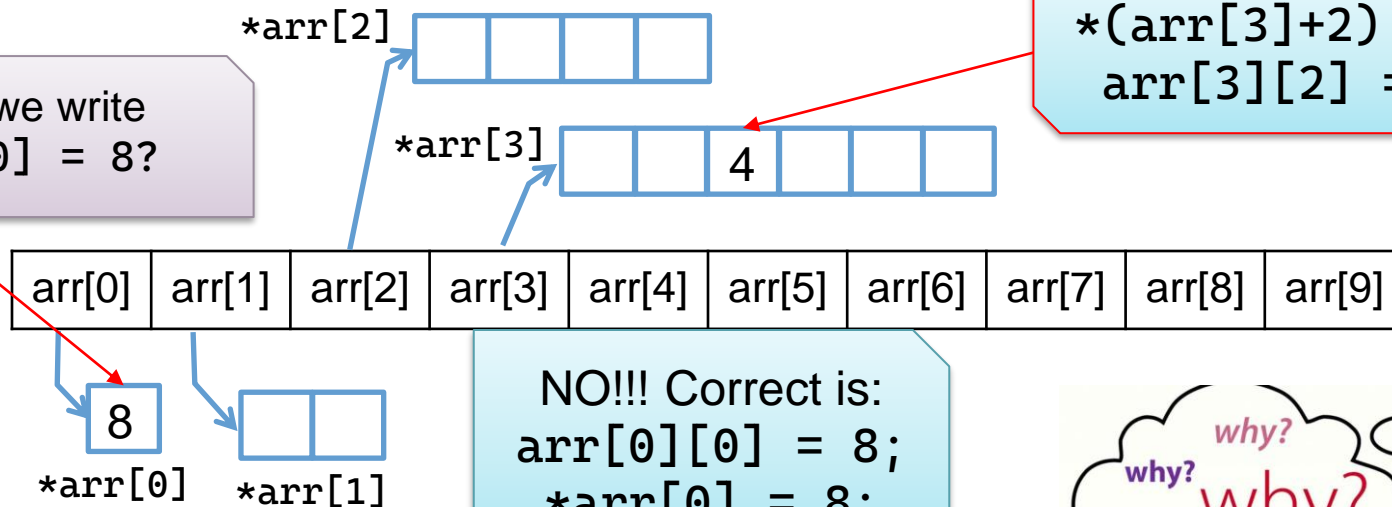
```
int* arr[10];
int** ptr;
```

- `arr` is an array of 10 pointers to integers. Each element of `arr` is a pointer to an integer value. It means **`arr[i]` is a pointer** and can be dynamically allocated.
  - We have 10 static pointers `arr[i]` (only the memories that they point to can be dynamical)
- `ptr` is a double pointer that points to another pointer variable.
  - We have only 1 static pointer that points to another pointer variable.
  - But we can use this pointer (`ptr`) to dynamically allocate memory for the integer pointer that it points to (or even allocate an array of pointers dynamically!!!)

# Array of Pointers

```
int* arr[10];
arr[0] = new int;
arr[1] = new int[2];
arr[2] = new int[4];
arr[3] = new int[6];
...
```

Can we write  
`arr[0] = 8;`



`*(arr[3]+2) = 4;`  
`arr[3][2] = 4;`

NO!!! Correct is:  
`arr[0][0] = 8;`  
`*arr[0] = 8;`



# Array of Pointers – Deallocate

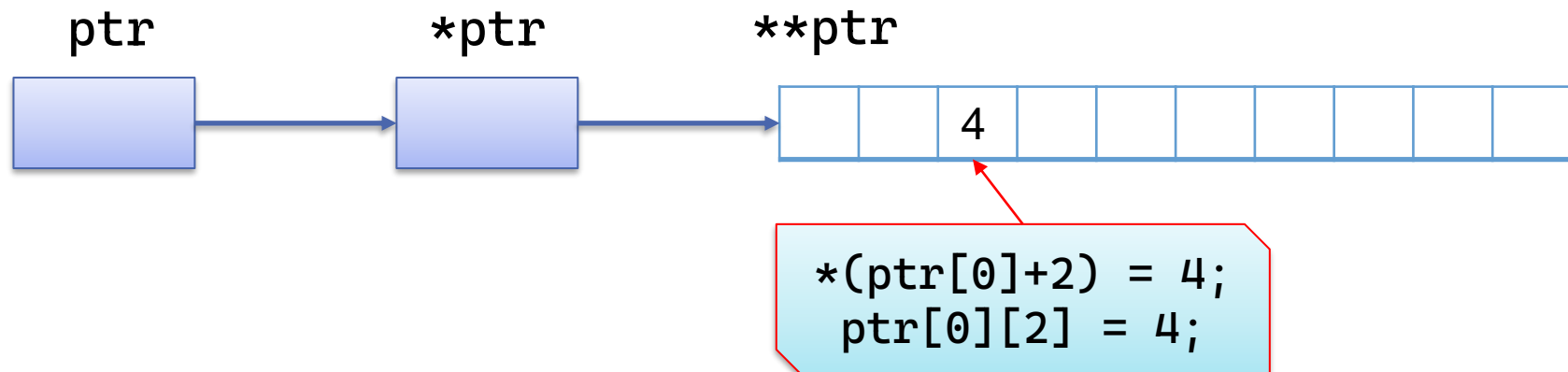
- To deallocate an array of pointers in C++, you need to free the memory allocated for each pointer in the array.

```
int* arr[10];  
for (int i = 0; i < 10; i++)  
    arr[i] = new int[i + 1]; //allocate 10 dynamical arrays  
for (int i = 0; i < 10; i++)  
{  
    delete[] arr[i]; //deallocate dynamic memories  
    arr[i] = NULL; //avoid using a dangling pointer  
}  
delete[] arr; //arr is a static variable, we don't need to delete it
```



# Double Pointers

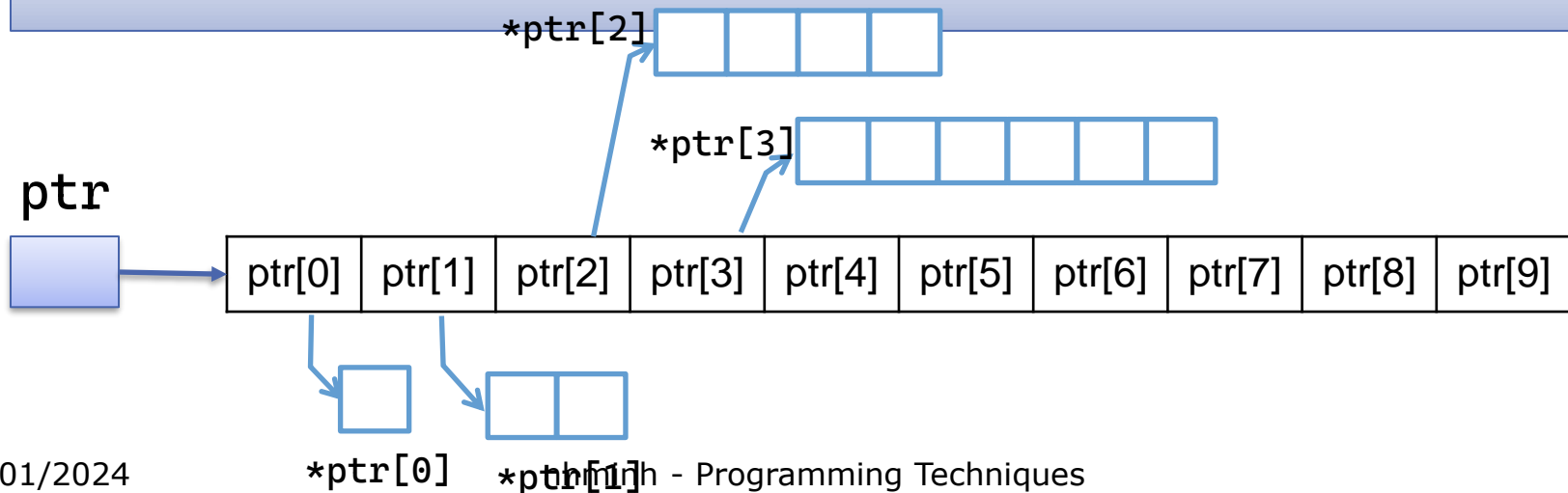
```
int** ptr;  
ptr = new int*;  
*ptr = new int[10];
```



# Double Pointers

- We can use double pointer to allocate an array of pointers dynamically

```
int** ptr;
ptr = new int*[10]; //allocate an array of 10 pointers dynamically
ptr[0] = new int;
ptr[1] = new int[2];
ptr[2] = new int[4];
ptr[3] = new int[6];
...
```



# Double Pointers – Deallocate

- Double pointers in C++ are often used to manage dynamically allocated memory. To deallocate the memory pointed to by a double pointer, you need to follow a few steps:
  - Deallocate the memory pointed to by the pointer: Use the `delete` operator to free the memory pointed to by the double pointer.
  - Set the pointer to `null`: After deallocating the memory, set the pointer to null to prevent it from being used again accidentally.

```
int** ptr;
ptr = new int*[10]; //allocate an array of 10 pointers dynamically
//...allocate memories for the double pointer...
//deallocate memories
for (int i = 0; i < 10; i++)
    delete[] ptr[i];
delete[] ptr;
ptr = NULL;
```

# Usages of Array of Pointers

1. Dynamic memory allocation

2. Arrays of strings:

```
const char* strs[] = {"Hello", "world", "!"};
```

3. Multi-dimensional arrays

```
int* arr[3];  
for (int i = 0; i < 3; i++)  
    arr[i] = new int[4];
```

4. Pointers to functions

5. Data structures: Arrays of pointers are often used to store pointers to data structures.

```
video * arr[10];
```

# Usages of Pointer to Pointer

1. Dynamic memory allocation: for example, create a 2D array.
2. Pointer parameters to functions that need to modify a pointer variable.:

```
void createList(ListNode** head);
```

3. Creating an array of pointers

```
int* arr[3];
```

```
int** ptr = arr;
```



# POINTERS TO FUNCTIONS

Obtain the address of a function

Declare a pointer to a function

Use a pointer to invoke the function

# Pointers to Functions

- Functions, like data items, have addresses.
  - A function's address is the memory address at which the stored machine language code for the function begins.
  - It's possible to write a function that takes the address of another function as an argument.
  - It leaves open the possibility of passing different function addresses at different times.

*→ The first function can use different functions at different times*

# Obtain the address of a function

- ❑ You just use the function name without trailing parentheses.
- ❑ If `think()` is a function, then `think` is the address of the function.
- ❑ To pass a function as an argument, you pass the function name.
- ❑ Be careful:

```
process(think); //pass the address of think() to process()
thought(think()); //pass the return value of think() to thought()
```



# Declaring a Pointer to a Function

- A pointer to a function has to specify to what type of function the pointer points:
  - the function's return type
  - the function's argument list

```
double square_root(int); //prototype
double exponent(int);    //prototype
double (*pf) (int); //pf points to a function that takes one int
                      //argument and returns a double type
```

- Be careful: *Parentheses have a higher precedence than the \* operator*

```
double* pf (int); //pf is a function that returns a pointer
```

# Using a Pointer to Invoke a Function

- Recall, (\*pf) plays the same role as a function name. Thus, all you have to do is use (\*pf) as if it were a function name:

```
double (*pf) (int);  
pf = square_root;  
double x = (*pf)(5);  
double y = square_root(5);  
cout << x << " & " << y << endl;
```



# Using a Pointer to Invoke a Function

- Use **typedef** keyword allows you to create a type alias:

```
typedef double real;
```

- The technique is to declare the alias as if it were an identifier and to insert the keyword typedef at the beginning. So you can do this to make **p\_func** an alias for the function pointer type

```
typedef double (*p_func)(int);  
p_func p1 = square_root;  
p1(5); //same as square_root(5);
```

# Next Week

---

## □ Dynamic Structures – Linked List

