fit@hcmus

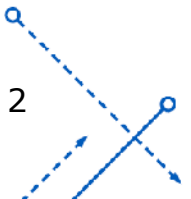
PROGRAMMING TECHNIQUES

Week 2: Pointers and Dynamic Memory (2)

Nguyễn Hải Minh - 02/2023

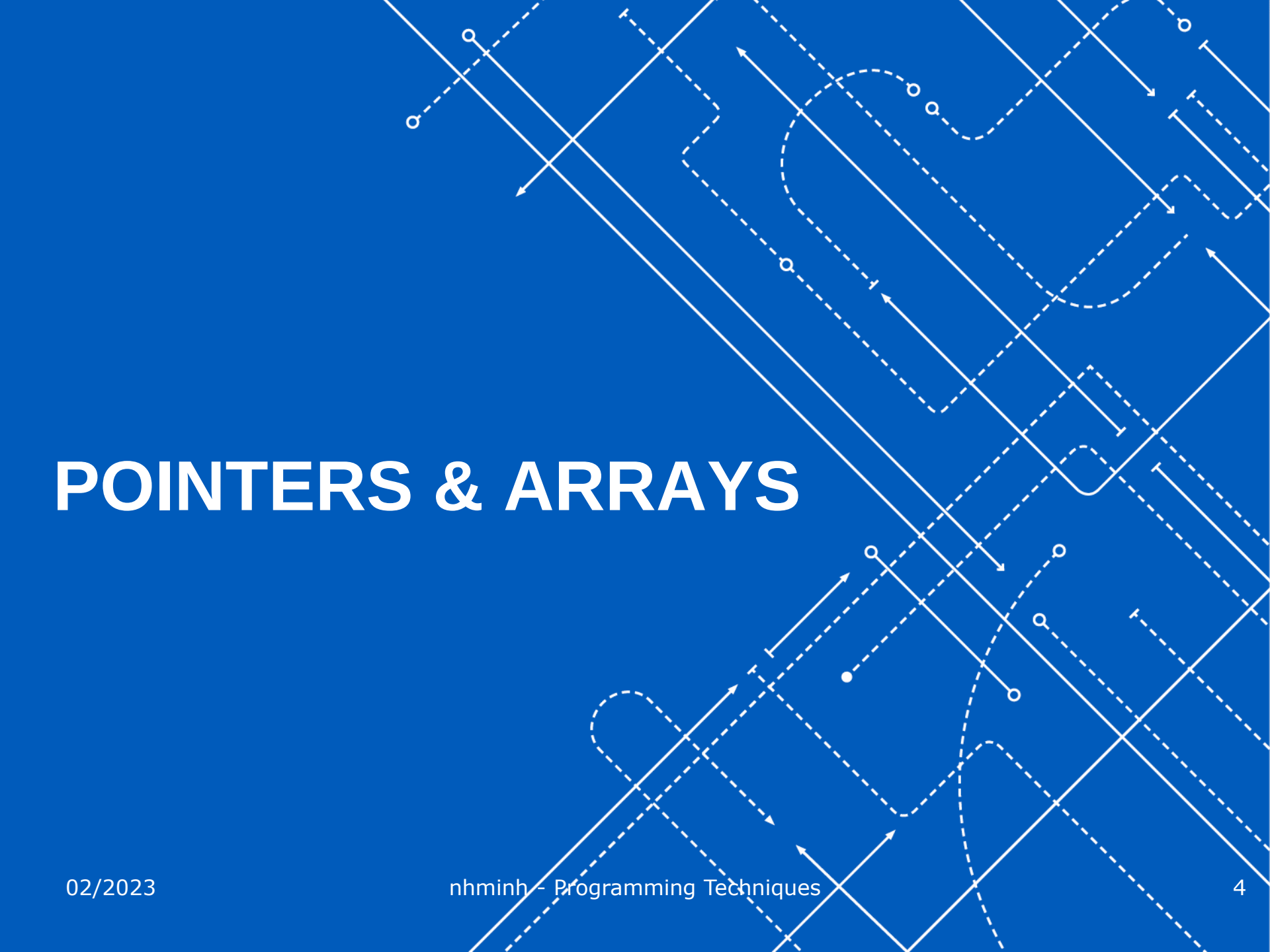
Content

- Review of Pointers
- Pointers and Arrays
- Pointers and Strings
- Dynamic Structures



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?


POINTERS & ARRAYS

Pointers and Arrays

- We have learnt using pointer to allocate memory for an array.
- It is also possible to use pointers to point to data that is stored sequentially in memory.
- We can treat a pointer to data stored sequentially in memory as an array.
- All operations on arrays have an equivalent pointer representation.
- We can take advantage of this to improve our programs' performance when operating on arrays.

Pointers and Arrays

- It is possible to define the behavior of the subscript operator `[]` entirely in terms of operations on a pointer.
- The first thing we need to know is that the identifier of an array is a constant pointer to the first element of that array.
- → It is a pointer to the same type as the elements of the array.
- → This means that we can initialize or assign an array name to a pointer, where the pointer points to data of the same type as the elements of our array.

Pointers and Arrays

```
int arr[7];  //arr is of type pointer to int
int* ptr;    //ptr is a pointer to int
ptr = arr;    //ptr now points to the array arr
```

- We now have two ways to access elements of an array, one using the name of the array (arr) and the other using a pointer (ptr).
- In this example, the name of the array (arr) is a constant pointer to an int. `//int* const`
- The pointer (ptr) is a variable pointer to an int that has been assigned the same address as arr.

Pointers and Arrays

- Since the value of arr has been assigned to ptr, the residual value of using either arr or ptr in an expression is the same in either case:
  - → it is the address of the first element of the array.
- When arr is used in an expression, that expression uses the value that the constant arr represents.
- When ptr is used in an expression, that expression uses the value currently assigned to variable ptr.
- We can apply the subscript operator to this residual value (an expression of type pointer to an int) in order to access the elements of the array.

Pointers and Arrays

```
int *ptr;                    //ptr is a pointer to an int
ptr = arr;                    //same as ptr = &a[0]
for(int i=0; i<7; i++)
 if(arr[i] != ptr[i])
   cout <<"Oops - big trouble in River City“ << endl;
```

- The relationship between pointers and arrays is defined by the following identity, where E1 is a pointer (either an array name or a pointer expression) and E2 is an integer expression.

```
E1[E2] ==  *((E1)+(E2))
```

Pointers and Arrays

- This identity means that the subscript operation is equivalent to adding the index to the pointer expression and then dereferencing the result.
- Understanding this identity allows us to decompose array subscripting operations into pointer operations.
- Array and pointer operations can be the same, even though the declarations for arrays and pointers are different.

```
arr[3] = 42;    //this stores 42 w/array subscripting
*(arr+3)=42;    //same thing using pointer operations
*(3+arr)=42;    //addition is communitive
```

All of the above works as well if ptr were used instead!

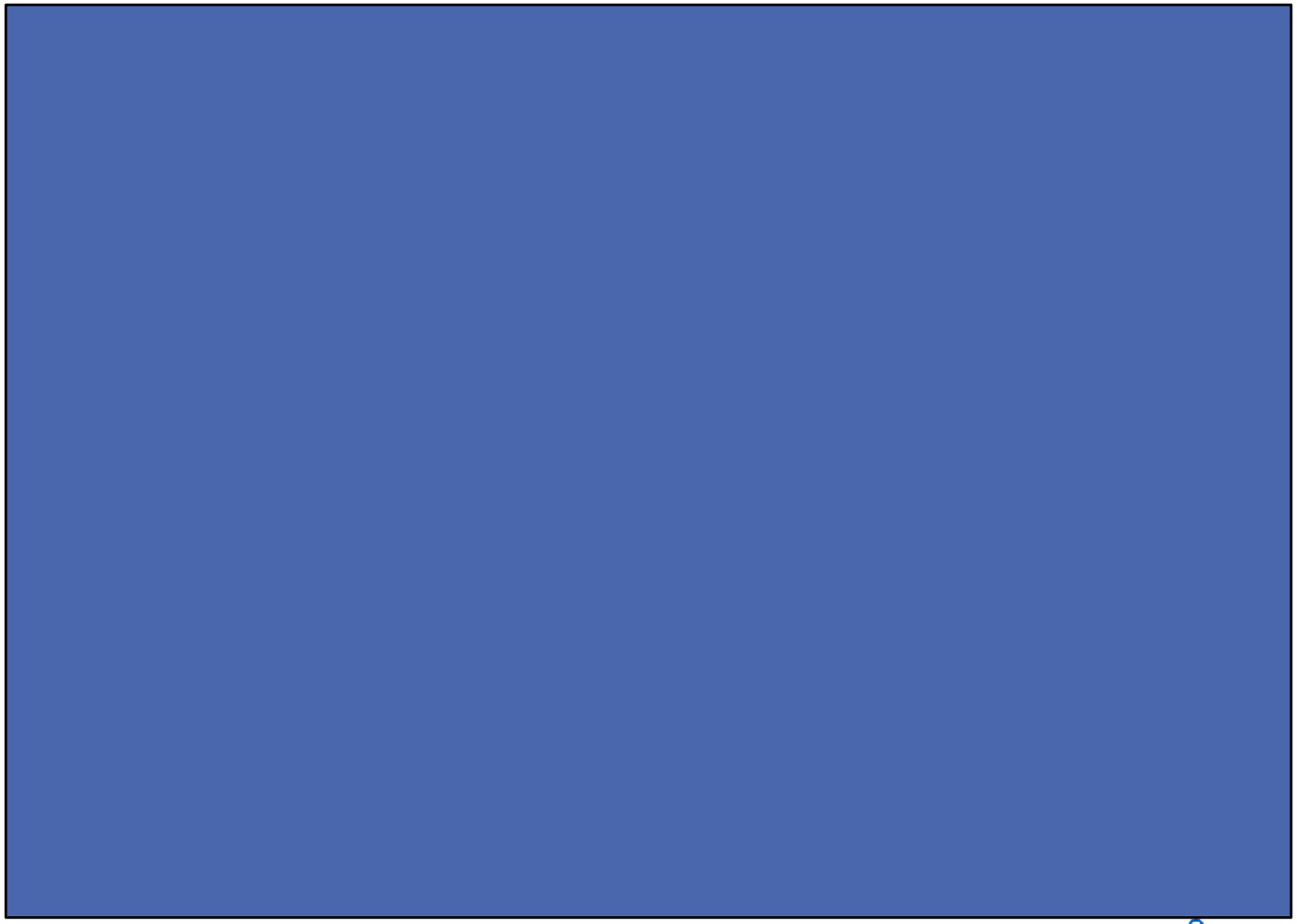
Pointers and Arrays

- We have seen that the name of an array can be replaced with a pointer to the first element of the array.
- The only difference is that the name of an array is a constant and cannot be modified, whereas a pointer can be defined as a variable and therefore can be modified.
- The process of modifying a pointer variable is called **pointer arithmetic**.

Pointer Arithmetic

- Walk through the following in class:

```
int a[10];
int* p=a;
int* q=&a[2];
p = q;

p = &a[5];
p+=3;
p-=8;

p = &a[5];
++p;
p++;
p = p + 2;
--p;
p--;
p = p - 2;
q = ++p;
q = p++;
```



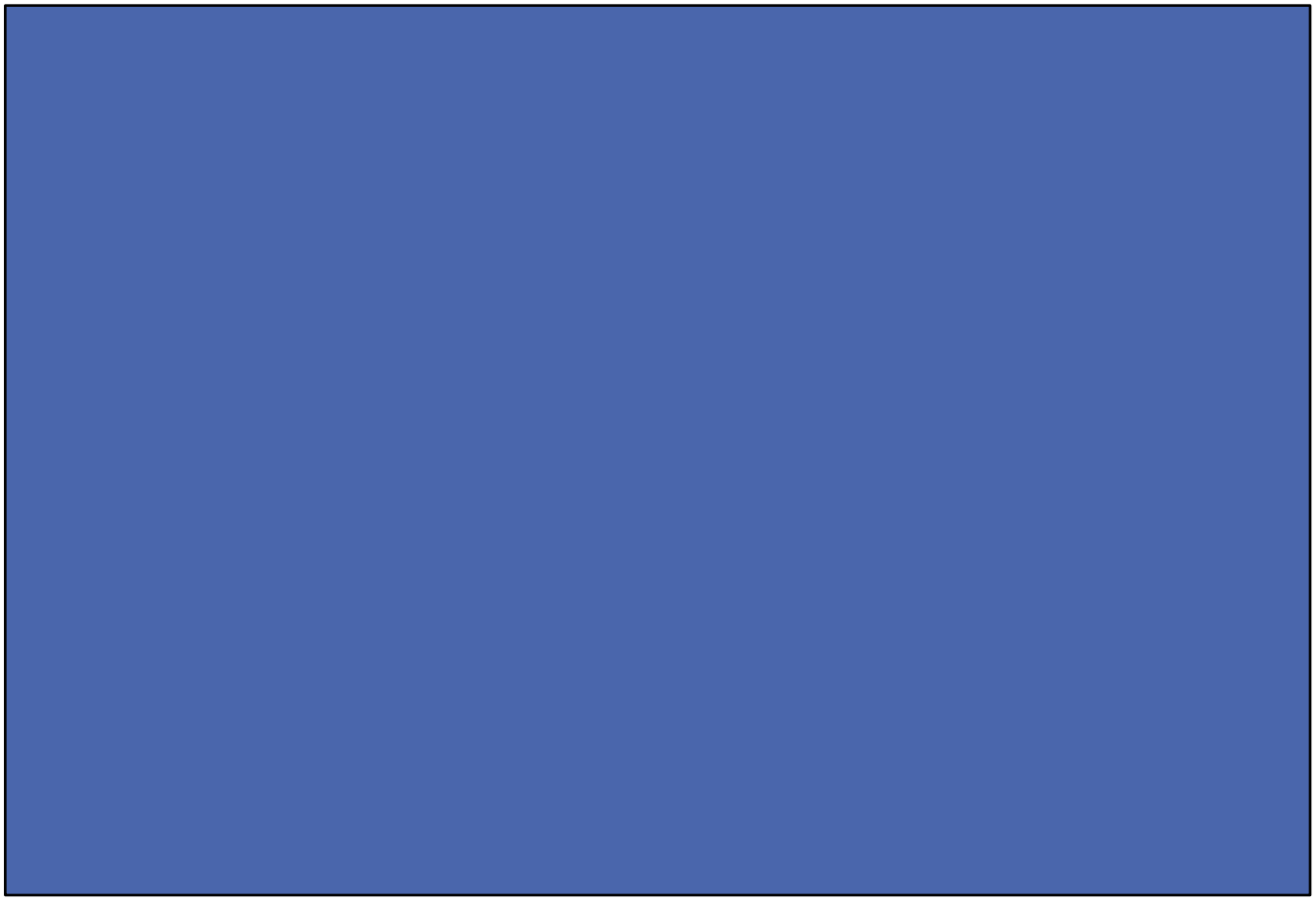
Pointer Arithmetic

- There are two key points in understanding pointer arithmetic.
- The first is that pointer variables can be modified whereas array names are constants and cannot be modified.
- The second is that pointer operations automatically take into account the size of the data pointed to, just like array subscripts do.
  - This means that operations such as addition and subtraction are independent of the size of the data.
  - When we add one to a pointer of some type, we point to the next element of that type.

Pointer Arithmetic

- Walk through the following in class:

```
int a[10];
int* p=a;
p = p + 1;
*p = *p + 1;
*p = *(p + 1);
p+=1;
*p+=1;
*(p+=1);
++p;
++*p;
*++p = 100;
p++;
*p++ = 200;
(*p)++;
```



Arrays of Arrays

- Arrays can be formed from any type of data, even other arrays!
- When each element is an array, we define an array of arrays.
- With an array of arrays, each element is an array of some type.
- Arrays of arrays are sometimes called multidimensional arrays in C++.
- This is not strictly correct because each dimension represents a different type, rather than each dimension representing the same type.

Arrays of Arrays

- An array of arrays is defined just like an array of a fundamental type, except that the identifier is immediately followed by an additional pair of brackets ([]).
- The size of each element's array, called a subarray, is supplied within the second set of brackets as a literal, constant, or constant expression.

```
int array[3][2];
```

Arrays of Arrays

- To access elements of an array of arrays, we can use the subscript operator. To access the appropriate subarray, we follow the name of the array by an index in brackets. For example, `array[0]` accesses the first subarray. The value of this element is the first subarray of two integers. Its type is an array of integers (a pointer to an int).
- To access elements within a subarray, we follow the name of the array by the index of the subarray in brackets and then follow that by the index of the element within the subarray that we wish to access in brackets. For example, `array[0][0]` accesses the first integer in the first subarray.

Arrays of Arrays

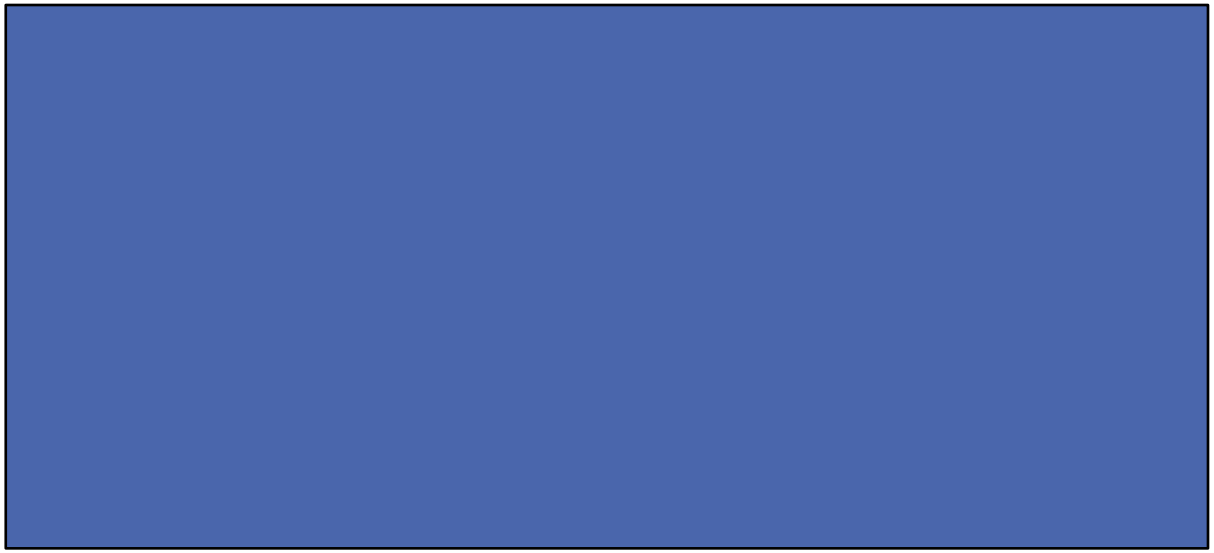
- The name of an array of arrays also represents a pointer expression. By itself, it has a value equal to the address of the first element of the array. The type is a pointer to the first element of the array. For example, the type of array is `int (*)[2]` (a pointer to an array of two integers).

```
int array[3][2];
int (*p1)[2]; //define pointer of same type as array
p1 = array;   //assign pointer to point to array
```

Array of Arrays

- Walk through the following in class:

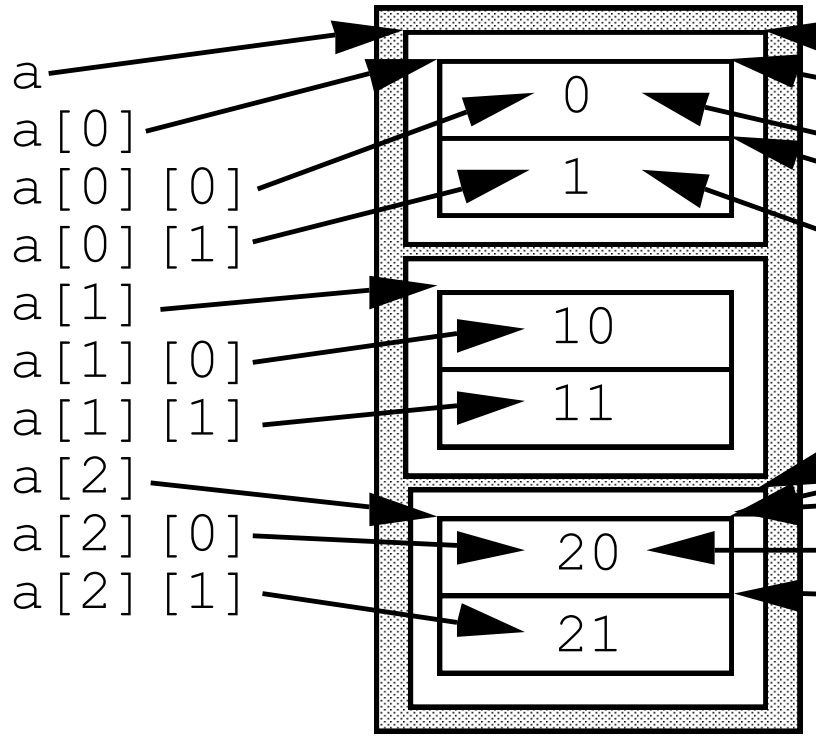
```
int array[6][2];
int (*p1)[2];
p1 = array;

int *p;
p = *p1;
p = array[0];
p = *(array+0);
p = *array;
```



- We can define and initialize a pointer to a subarray. The type of a subarray is a pointer to an element of the subarray. By defining a pointer of that type, we can use pointer arithmetic to access the subarray.

Array of Arrays

```
int a[3][2]={{0,1},{10,11},{20,21}};
```




```
p1=a          //int (*)[2]
*p1           //int*
**p1          //int
*p1+1         //int*
*(*p1+1)      //int
p1+2          //int (*)[2]
*(p1+2)       //int*
p=*(p1+2)     //int*
*p            //int
p+1           //int*
```

Arrays of Arrays

- If we were to print the value of the pointers p1 and p, their values would be the same even though they are different types.
- This is because the address of the first element of array is at the same address as the first element in the first subarray.
- However, when we add to or subtract from these two pointers, the results are significantly different.
- By adding one to pointer p1, we point to the next subarray of 2 integers.
- By adding one to pointer p, we point to the next int within the first subarray.


POINTERS & STRINGS

Pointers and Strings

- Consider the following code:

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

The name `flower` is the address of its 1st element

Address of character r

  - `cout` assumes that the address of a char is the address of a string, so it prints the character at that address and continues printing characters until it runs to a null character.
  - This implies that you can use a pointer-to-char variable as an argument to `cout` because it is also the address of a character.

Pointers and Strings

- Consider the following code:

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

This quoted string should also be an address.

Like an array name, it is the address of its 1st element.

- In C++, all are handled equivalently:
  - Strings in an array
  - Quoted string constants
  - Strings described by pointers

Passed along as an address (of the 1st character)

Pointers and Strings

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
  char animal[20] = "bear";    //animal holds bear
  const char* bird = “pigeon";//initialize a pointer-to-char
                              //to a string -> assign the address
                              //of pigeon to pointer bird
  char* ps;// uninitialized
  ps = animal;// set ps to point to string

  cout << animal << “\n";// display bear
  cout << bird << "\n";// display pigeon
  cout << ps << “\n" ; // display bear
  …
}
```

Pointers and Strings

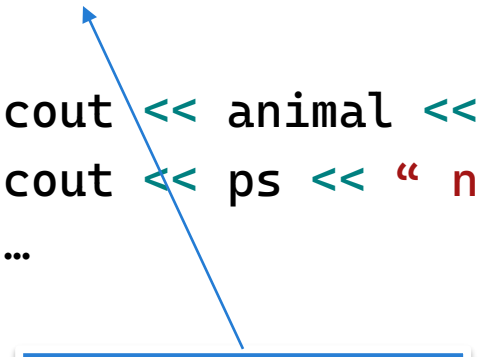
```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   …
   cout << animal << " is at " << (int *) animal << endl;
   cout << ps << " is at " << (int *) ps << endl;
   …
}
```

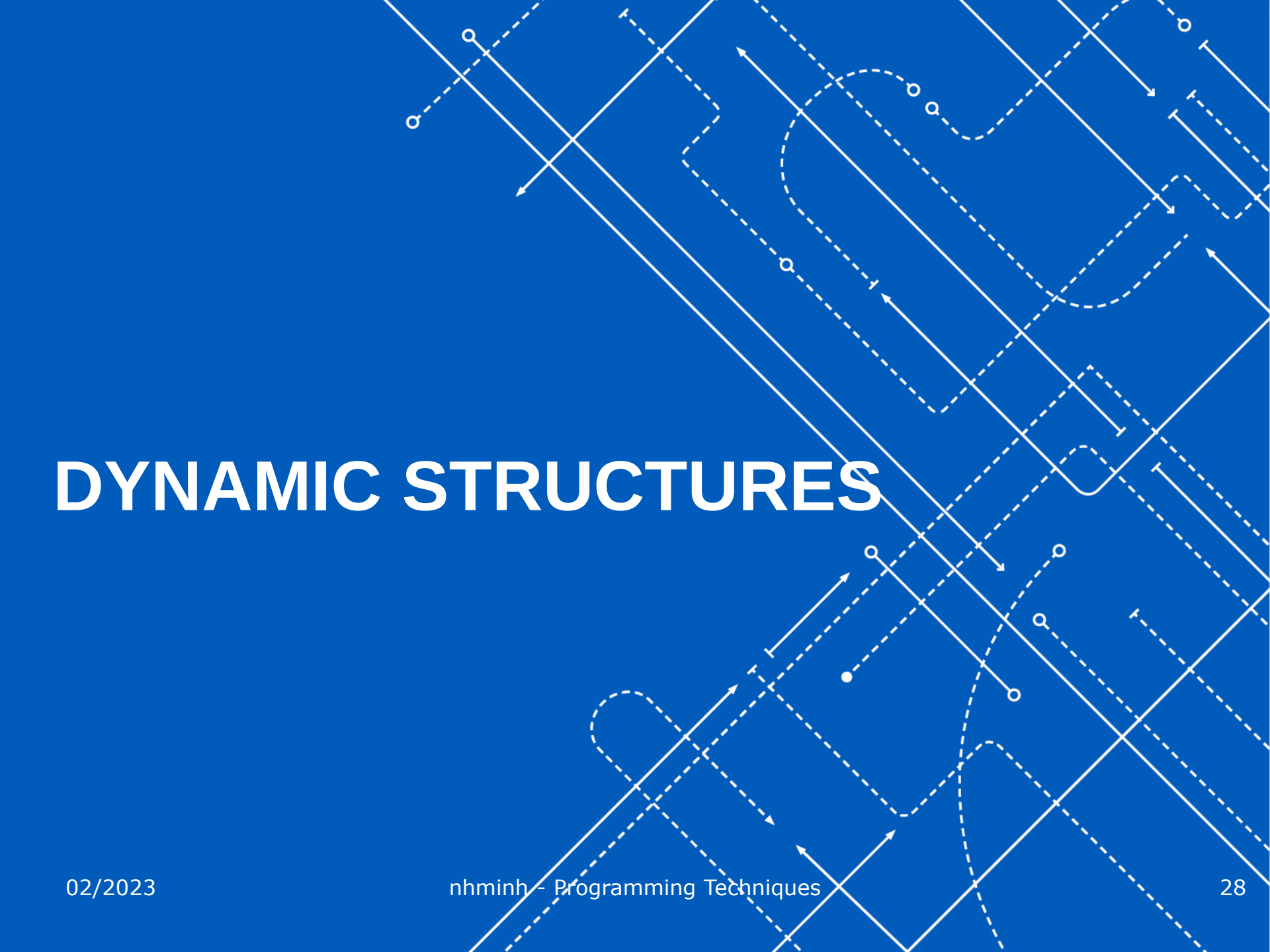
Address where the string is found

- `animal` and `ps` are pointers of type char → `cout` displays the pointed-to-string. If you want to see the address of the string, you have to type cast the pointer to another pointer type, such as `(int *)`
- *What if you want to get a copy of a string?*

Pointers and Strings

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   …
  int len = strlen(animal) + 1;
  ps = new char[len];// get new storage
  strcpy(ps, animal);// copy string to new storage

   cout << animal << " is at " << (int *) animal << endl;
   cout << ps << " now is at " << (int *) ps << endl;
   …

}
```




DYNAMIC STRUCTURES

Dynamic Structures

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

```
struct video {
   char title[20];
   char category[5];
   int quantity;
};
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.
- The member access operator (.) have the ***higher operator precedence*** than the dereference operator (*).
- 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

- 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!

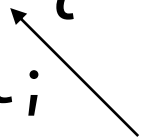
Dynamic Structures

- Ok, what about passing pointers to functions?
- Pass by value and 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;
}
```

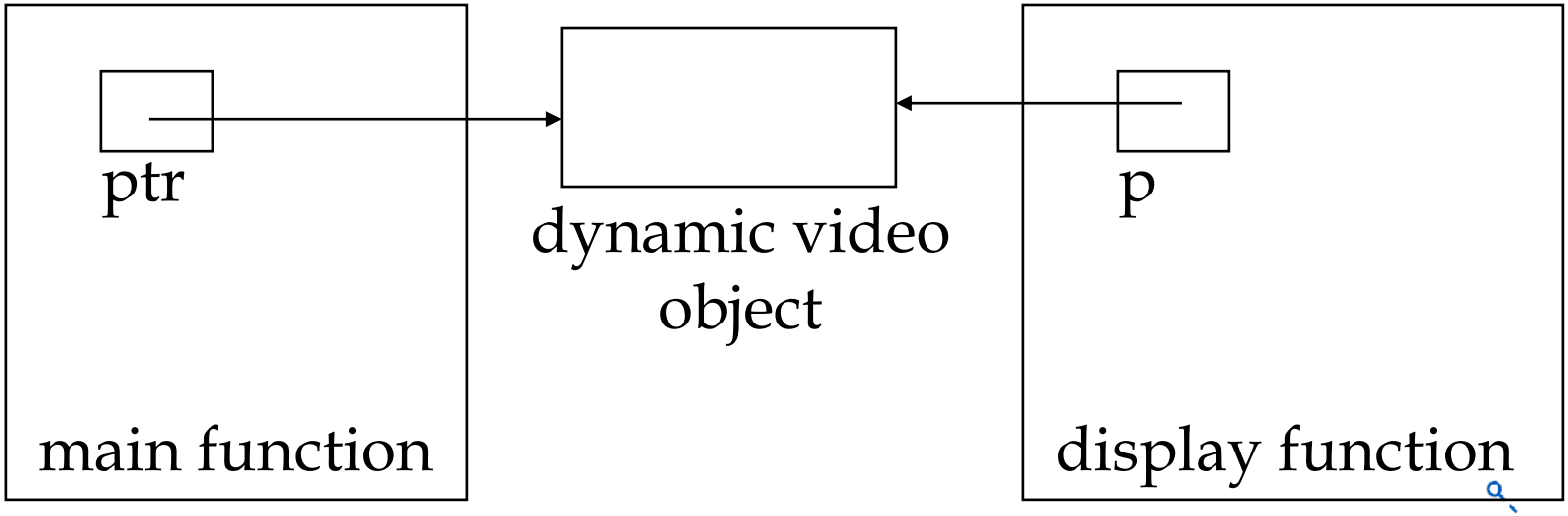


p is a pointer to a video object, passed by value. So, p is a local variable with an initial value of the address of a video object

Dynamic Structures

- Here is the pointer diagram for the previous example:

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



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)
- 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!

Next week's topic

- Double Pointers
- Pointers to Functions
- Homework:
  - Read your textbook: C++ Primer Plus page 361~


Q&A