Review

C++ classes have **6 implicitly defined** (built-in) member functions:

- Default constructor (i.e. implicit default (=0-parameter) constructor)
  - The moment you create any constructor of your own, this implicit default constructor is removed → Be careful with that!
- Destructor

- Copy constructor
- Copy assignment operator=

- Move constructor
  - Beyond the scope of this course
- Move assignment operator=
  - Beyond the scope of this course

Review

What do C++'s implicit destructor / assignment operator / copy constructor do?

- For primitive fields - shallow copy:
  - Copy constructor and operator= copy the bytes.
  - Destructor does nothing

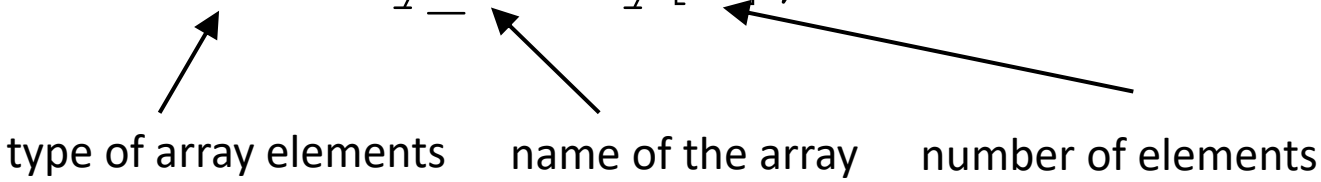
- For non-primitive fields
  - Copy constructor and "operator=" → The built-in implementation uses the CC and "operator=" provided by that non-primitive fields
  - The destructor ***always*** calls that type's destructor

- Note: A pointer is a primitive type.

Arrays

- An array is a **primitive** type. A vector is a non-primitive type.
- Array: A variable that can store multiple values of the same type, for example:

```
int my_array[5];
```



The above definition allocates the following memory

1st element	2nd element	3rd element	4th element	5th element

Arrays (cont.)

- Array size is the number of bytes in the array.
  - Typically, the array below has 20 bytes, since each element is an int (4 bytes)

```
int my_array[5];
```

- The **array size is fixed** and is determined when declaring the array (this is different from vectors).
- An array element can be used as if it's a regular variable, using index operator (aka the array operator):

```
my_array[0] = 79;
cout << my_array[0];
int i = 3;
my_array[i] = 63;
```

- Arrays must be accessed via individual elements:

```
cout << my_array; // won't print the array  elements
```

Arrays (cont.)

- Each element in an array is assigned a unique *subscript*.
- Subscripts start at 0
- The last element's subscript is *n*-1 where *n* is the number of elements in the array.

Array initialization

- Arrays can be initialized with an initialization list:

```
const int SIZE = 5;
int arr[SIZE] = {79,82,91,77,84};
```

- The values are stored in the array in the order in which they appear in the list.
- The initialization list cannot exceed the array size.

- Alternatively, the compiler can determine the size:

```
int arr[] = {79,82,91,77,84}; //size=5
```

Two-dimensional arrays

- Arrays can be two-demensional:

```
int exams[4][3];
```

- Use 2 subscripts the access the array

```
exams[2][2] = 95;
```

- Two-dimensional arrays are initialized row-by-row:

```
int exams[2][2] = { {84, 78},
                    {92, 97} };
```

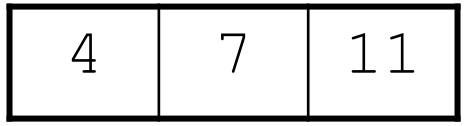
exams[0][0]	exams[0][1]	exams[0][2]
exams[1][0]	exams[1][1]	exams[1][2]
exams[2][0]	exams[2][1]	exams[2][2]
exams[3][0]	exams[3][1]	exams[3][2]

84	78
92	97

Arrays and pointers

- The array name holds the starting address of the array

```
int vals[] = {4, 7, 11};
```



Assume starting address of `vals`: `0x4a00`

```
cout << vals << endl;      // displays 0x4a00
cout << vals[0] << endl;   // displays 4
```

Arrays and pointers – cont.

- Array name can be used as a constant pointer:

```
int vals[] = {4, 7, 11};
cout << *vals << endl;     // displays 4
```

- Pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[1] << endl; // displays 7
```

Arrays and pointers

- Hence, arrays work very much like pointers to their first element, and an array can always be implicitly converted to a pointer of the proper type, i.e. a ***pointer can be assigned any value, whereas an array can only represent the same elements it pointed to during its instantiation***, hence:

```
int x[20];
int *px;

px = x;
```

valid

```
x = px;
```

Not valid

- Memory allocated:
  - Arrays: memory allocated to hold the number of elements inside the array.
  - Pointers: memory allocated to hold one address.

Arrays and pointers – example

```
#include <iostream>
using namespace std;

int main()
{
    int numbers[5];
    int* p;
    p = numbers;  *p = 10;
    p++;  *p = 20;
    p = &numbers[2];  *p = 30;
    p = numbers + 3;  *p = 40;
    p = numbers;  *(p + 4) = 50;
    for (int n = 0; n < 5; n++)
        cout << numbers[n] << " ";
    return 0;
}
```

Array and pointers – example

```
#include <iostream>
using namespace std;

int main()
{
    int numbers[5];
    int* p;
    p = numbers;  *p = 10;
    p++;  *p = 20;
    p = &numbers[2];  *p = 30;
    p = numbers + 3;  *p = 40;
    p = numbers;  *(p + 4) = 50;
    for (int n = 0; n < 5; n++)
        cout << numbers[n] << " ";
    return 0;
}
```

```
10, 20, 30, 40, 50,
```

Let’s implement a Vector class

At a minimum, we need:

- Default constructor
- Constructor with size and value
- push_back()
- size() – it’s a getter → must use **const**
- Operator[]

- pop_back()
- Explicit the constructor
- Copy control:
  - Destructor
  - copy constructor
  - operator=
- clear()
- back()

Implementing the constructor:

- What inputs do we need? size and values
- We need to allocate memory of a certain size
- Attributes:
  - Capacity
  - Size
  - Pointer to allocated memory

Implementing the destructor

- We need to deallocate the memory

Implementing the push_back(value)

1) Test if space needed – if true:
   - If capacity is zero, delete old and allocate an array of size one, setting capacity to one.
   - Otherwise allocate a “larger” array and copy the elements
     How much larger?
     Doubling is common, but some argue about the exact algorithm.
2) Add new item (i.e. copy its value to array)
3) Increment size.