# std::array

```
Topics:
  . std::array
  . An easy to use, built in collection type
```

# std::array

```cpp
// Declaration and initialization
std::array<int, 5> arr = {1, 2, 3, 4, 5};

// Accessing elements using []
for (size_t i = 0; i < arr.size(); ++i) {
    fmt::println("arr[{}] = {}", i, arr[i]);
}

// Use range based for loop
for (const auto& element : arr) {
    fmt::print("{} ", element);
}

// Accessing elements using at()
for (size_t i = 0; i < arr.size(); ++i) {
    fmt::println("arr.at({}) = {}", i, arr.at(i));
}
```

## at() or []?

- at() performs bounds checking
- [] doesn't perform bounds checking
- Use at() when safety and error handling are priorities
- Use [] when performance is critical and you are certain the index will always be within bounds.

# Modify elements

```cpp
// Modifying elements in the array
std::array<int, 5> arr = {1, 2, 3, 4, 5};
arr[0] = 10;
arr.at(1) = 20;

// Printing modified array
for (size_t i = 0; i < arr.size(); ++i) {
    fmt::println("arr[{}] = {}", i, arr[i]);
}
```

# Methods: Fill

```cpp
//Filling an array with a value
std::array<int, 5> arr; // Array initialized with junk values

// Filling array with a single value
arr.fill(7);

for (const auto& element : arr) {
    fmt::print("{} ", element);
}
fmt::print("\n");
```

# Methods: front and back

```cpp
// Accessing the first and last elements
std::array<int, 5> arr = {1, 2, 3, 4, 5};
fmt::println("First element: {}", arr.front());
fmt::println("Last element: {}", arr.back());
```

# 2 dimensions

```cpp
// Multi-dimensional: 2D array (3x2)
std::array<std::array<int, 2>, 3> arr = {
            {
                {1, 2},
                {3, 4},
                {5, 6}
            }
};

for (const auto& row : arr) {
    for (const auto& element : row) {
        fmt::print("{} ", element);
    }
    fmt::print("\n");
}
```

# Comparing arrays

```cpp
// Comparing arrays
std::array<int, 3> arr1 = {1, 2, 3};
std::array<int, 3> arr2 = {1, 2, 3};
std::array<int, 3> arr3 = {1, 2, 4};
std::array<int, 3> arr4 = {0, 2, 3};

// Comparing arrays for equality
if (arr1 == arr2) {
    fmt::println("arr1 is equal to arr2");
} else {
    fmt::println("arr1 is not equal to arr2");
}

// Comparing arrays for inequality
if (arr1 != arr3) {
    fmt::println("arr1 is not equal to arr3");
} else {
    fmt::println("arr1 is equal to arr3");
}
// Lexicographical comparison
if (arr1 < arr3) {
    fmt::println("arr1 is less than arr3");
} else {
    fmt::println("arr1 is not less than arr3");
}
```

# Comparing arrays

```cpp
// Assigning one array to another
std::array<int, 5> arr1 = {1, 2, 3, 4, 5};
std::array<int, 5> arr2 = {6, 7, 8, 9, 10};

// Printing original arrays
fmt::println("Original arr1: ");
for (const auto& element : arr1) {
    fmt::print("{} ", element);
}
fmt::print("\n");

fmt::println("Original arr2: ");
for (const auto& element : arr2) {
    fmt::print("{} ", element);
}

// Assigning arr2 to arr1
arr1 = arr2;

// Printing arrays after assignment
fmt::println("arr1 after assignment: ");
for (const auto& element : arr1) {
    fmt::print("{} ", element);
}
fmt::print("\n");

fmt::println("arr2 after assignment: ");
for (const auto& element : arr2) {
    fmt::print("{} ", element);
}
fmt::print("\n");
```