Ch7: Class Templates array and vector

std::array
○○○○○○○○○○

Arrays and Loops
○○○○○○

Multidimensional Arrays
○○○

std::vector
○○○○

std::tuple, std::pair
○○○○○

std::array

std::array
Arrays and Loops

Multidimensional Arrays

std::vector

std::tuple, std::pair

Stony Brook University

std::array

An array is an aggregate data type that lets us access many variables of the <u>same type</u> through a single identifier. Arrays can be made from <u>any data type</u>.

- C++ Standard Library includes std::array and std::vector class templates to address the issues with built-in C-style fixed and dynamic arrays and make array management easier.

- std::array provides **<u>fixed array</u>** functionality and is defined in the <array> header, inside the std namespace.

❖ std::array **declaration:**

> std::array<type, arraySize> arrayName;

❖ std::array initialization:

It can be initialized using list initialization or initializer lists.

```cpp
#include <array>
int main() {
  std::array<int, 3> a; // declare an integer array with length 3
  std::array<int, 5> a2 { 9, 7, 5, 3, 1 }; // list initialization
  std::array<int, 5> a3 = { 9, 7, 5, 3, 1 }; // initializer list
}
```
▶

std::array **Declaration**

When declaring a std::array, the length of the array must be a **compile-time constant** (i.e., a constexpr variable). **Non-const variables** or **runtime constants** cannot be used.

```
#include <iostream>
#include <array>
int foo() {
    return 3;
}
int main() {
  int x1{3}; // x is a non-const variable
  // std::array<int, x1> a1{9, 7, 5}; // Error
  const int y1{ x1 }; // y isn't known until runtime, so y is a runtime constant.
  // std::array<int, y1> a2{9, 7, 5}; // Error

  const int x2{3}; // x is a const variable
  std::array<int, x2> a3{9, 7, 5}; // OK
  const int y2{ x2 }; // y is a compile-time constant.
  std::array<int, y2> a4{9, 7, 5}; // OK

  // std::array<int, foo()> a5{9, 7, 5}; // Error: value of foo() isn't known until runtime.
}
```

▶

Accessing Elements of std::array

- Each of the variables in an array is called an **element**.
- To access individual elements of an std::array, there are two methods:

  o Using subscript operator [·]. This does not do any bounds-checking.
  o Using at(·) function. This does (runtime) bounds-checking.

- In each method, a subscript (or index) tells the compiler which element we want.
- An array subscript can be a literal value, a variable (constant or non-constant), or an expression that evaluates to an integral type.

- Arrays count starting from 0. For an array of length N, the array elements are numbered 0 through N-1. This is called the array's range.

- Because at(·) does bounds checking, it is slower (but safer) than operator [·].

```cpp
#include <iostream>
#include <array>
int main() {
  std::array<int, 3> a {9, 7, 5};
  std::cout << a[0] << '\n';
  std::cout << a[1] << '\n';
  std::cout << a[2] << '\n';
  std::cout << a[3] << '\n'; // returns a value!
  std::cout << a.at(0) << '\n';
  // std::cout << a.at(3) << '\n'; // invalid, throws a runtime error
}
```

std::array **Initialization**

If there are less initializers in the list than the array can hold, the remaining elements are initialized to 0 for int, 0.0 for double, …

If there are more initializers in the list than the array can hold, the compiler will generate an error.

```cpp
#include <iostream>
#include <string>
#include <array>

int main() {
  std::array<int, 5> x; // self-initialize to zero
  x = { 0, 1, 2, 3, 4 }; // initializer list
  x = { 9, 8, 7 }; // elements 3 and 4 are set to zero.
  // x = { 0, 1, 2, 3, 4, 5 }; // Error: too many elements!
  x[0] = 11; // changing value of an element
  x.at(4) = 12; // changing value of an element

  std::array<int, 5> a{}; // explicit zero-initialization: {0,0,0,0,0}
  std::array<double, 5> b{}; // explicit zero-initialization: {0.0,0.0,0.0,…}
  std::array<std::string, 5> c{}; // Initialize all elements to an empty string

  std::cout << x[0] << '\n';
  std::cout << x[1] << '\n';
  std::cout << x[2] << '\n';
  std::cout << x[3] << '\n';
  std::cout << x[4] << '\n';
}
```

▶

std::array **Since C++17**

Before C++17, you cannot omit the array length even when providing an initializer. Since C++17, you can omit type and size (both together), when initializing an array.

```cpp
#include <array>

int main() {
  // Before C++17:
  std::array<int, 5> b { 9, 7, 5, 3, 1 };
  // std::array<int, > b { 9, 7, 5, 3, 1 }; // Error: array length must be provided
  // std::array<int> b { 9, 7, 5, 3, 1 }; // Error: array length must be provided

  // Since C++17:
  std::array c { 9, 7, 5, 3, 1 }; // The type is deduced to std::array<int, 5>
  std::array c1 { 9.7, 7.31 }; // The type is deduced to std::array<double, 2>
}
```
▶

Size and Sorting

- The size() member function can be used to retrieve the length of the std::array.

```cpp
#include <iostream>
#include <array>
int main() {
    std::array<double, 5> myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };
    unsigned int x{myArray.size()};
    std::cout << "length: " << x << '\n';
}
```
▶

- The std::sort function can be used to sort std::array. It is defined in the <algorithm> header, inside the std namespace.

```cpp
#include <iostream>
#include <array>
#include <algorithm> // for std::sort
int main() {
    std::array<int, 5> myArray { 7, 3, 1, 9, 5 };
    std::sort(myArray.begin(), myArray.end()); // sort the array forwards
    // std::sort(myArray.rbegin(), myArray.rend()); // sort the array backwards
    std::cout << myArray[0] << " " << myArray[1] << " " << myArray[2] << " " << myArray[3] << " " << myArray[4] << '\n';
}
```
▶

Passing std::array **to Functions**

Always pass std::array by **reference** or const **reference**. This is to prevent the compiler from making a copy of the std::array (which for large arrays can be very expensive) when the std::array was passed to the function.

```cpp
#include <iostream>
#include <array>
void printLength(const std::array<double, 5>& myArray) {
    std::cout << "length: " << myArray.size() << '\n';
}
int main() {
    std::array<double, 5> myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };
    printLength(myArray);
    std::array<int, 5> myArray2{ 9, 7, 5, 3, 1 };
    // printLength(myArray2); // Error: printLength expects a std::array<double, 5>
}
```
▶

We use a std::array as a function parameter, we must specify the element type and array length.

Passing std::array **of Different Types/Lengths to Functions**

Using **template functions**, we can pass std::array of different types and lengths to functions.

```cpp
#include <iostream>
#include <array>
#include <cstddef> // for size_t

// A template function
template <typename T, std::size_t size> // parameterize the element type and size
void printLength(const std::array<T, size>& myArray) {
    std::cout << "length: " << myArray.size() << '\n';
}

int main() {
    std::array<double, 5> a{ 9.0, 7.2, 5.4, 3.6, 1.8 };
    printLength(a);

    std::array<int, 3> b{ 9, 7, 5};
    printLength(b);

    std::array c{ 9.0, 7.2, 5.4, 3.6}; // Since C++17
    printLength(c);
}
```

▶

std::size_t

std::size_t is defined as an unsigned integral type, and it is typically used to represent the size or length of objects (or indices of arrays). For example, sizeof return a value of type std::size_t.

```
#include <iostream>
int main() {
    std::cout << sizeof(int) << '\n';
}
```

std::size_t is guaranteed to be unsigned and at least 16 bits, however, it is defined to be big enough to hold the size of the largest object creatable on your system (in bytes).

Arrays and Loops

std::array
○○○○○○○○○○

Arrays and Loops
●○○○○○

Multidimensional Arrays
○○○

std::vector
○○○○

std::tuple, std::pair
○○○○○

Stony Brook University

Arrays and Loops

We can use a loop variable as an array index to loop through all of the elements of our array and perform some calculation on them.

Ex: Determine the best score.

```cpp
#include <iostream>
#include <array>
#include <cstddef> // std::size_t
int main() {
  std::array scores { 84, 92, 76, 81, 56 };
  int maxScore{ 0 }; // keep track of our largest score
  for (std::size_t student{0} ; student < scores.size(); ++student) {
    if (scores[student] > maxScore) {
      maxScore = scores[student];
    }
  }
  std::cout << "The best score is " << maxScore << '\n';
}
```
▶

```cpp
#include <iostream>
#include <array>
#include <cstddef> // std::size_t
int main() {
  std::array myArray { 7, 3, 1, 9, 5 };
  for (std::size_t i{ 0 }; i < myArray.size(); ++i) {
    std::cout << myArray[i] << ' ';
  }
}
```
▶

❖ When using loops with arrays, always double-check your loop conditions to make sure you do not introduce **off-by-one errors**.

Remarks

Unsigned integers (like std::size_t) **wrap around** when you reach their limits. Therefore, decrementing an unsigned integer that is 0 already, causing a wrap-around to the maximum value.

Example: Printing an array in reverse order.

```
#include <array>
#include <iostream>

int main() {
  std::array myArray { 7, 3, 1, 9, 5 };

  // Print the array in reverse order.
  for (std::size_t i{ myArray.size() - 1 }; i >= 0; --i)
    std::cout << myArray[i] << ' ';
}
```
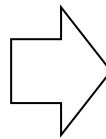▶

An infinite loop!

One possible solution. ⇒

```
#include <array>
#include <iostream>

int main() {
  std::array myArray { 7, 3, 1, 9, 5 };

  // Print the array in reverse order.
  for (std::size_t i{ myArray.size() }; i-- > 0; )
    std::cout << myArray[i] << ' ';
}
```
▶

Range-based for **Statement**

There is a simpler and safer type of loop called **Range-based** for **Statement** (or **for-each** loop) for cases where we want to iterate through every element in an array (or other list-type structure).

```
for (element_declaration : array) {
    statement(s);
}
```

The loop will iterate through each element in array, assigning the value of the current array element to the variable declared in element_declaration.

Note: element_declaration should have the same type as the array elements, otherwise type conversion will occur. Thus, the best practice is to use auto keyword, and let C++ deduce the type of the array elements for us.

```
#include <iostream>
#include <array>
int main() {
    std::array fibonacci{ 0, 1, 1, 2, 3, 5, 8, 13};
    for (auto number : fibonacci) {
        std::cout << number << ' ';
    }
}
```
▶

Range-based for Statement: Example

Ex: Rewriting the best score example using a range-based for statement.

```cpp
#include <iostream>
#include <array>
int main() {
  std::array scores { 84, 92, 76, 81, 56 };
  int maxScore{ 0 }; // keep track of our largest score
  for (auto score : scores) { // iterate over array scores, assigning each value in turn to variable score
    if (score > maxScore) {
      maxScore = score;
    }
  }
  std::cout << "The best score is " << maxScore << '\n';
}
```

▶

Range-based for Statement and References

Since each array element iterated over will be copied into variable element and copying array elements can be expensive (for non-fundamental data types), most of the time it is preferred to refer to the original element using a **const reference** (in a read-only fashion) to avoid having to make a copy for performance reasons.

```cpp
#include <iostream>
#include <array>
int main() {
  std::array myArray{ "peter", "likes", "frozen", "yogurt" };
  for (auto element : myArray) { // element will be a copy of the current array element
    std::cout << element << ' ';
  }
}
```
▶
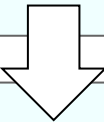
```cpp
#include <iostream>
#include <array>
int main() {
  std::array myArray{ "peter", "likes", "frozen", "yogurt" };
  for (const auto& element : myArray) { // element is a const reference to the currently iterated array element
    std::cout << element << ' ';
  }
}
```
▶

std::array
○○○○○○○○○○

Arrays and Loops
○○○○○○●

Multidimensional Arrays
○○○

std::vector
○○○○

std::tuple, std::pair
○○○○○

Range-based for Statement and Element Indices

Range-based for Statement do not provide a direct way to get the array index of the current element. One way to get the array index is to declare an index variable outside of the loop:

However, the scope of this index variable is too large. Since **C++20**, range-based for statement can be used with an init-statement just like the for statements.

```
for (init-statement; element_declaration : array) {
    statement(s);
}
```

The init-statement is a **local variable** for for statement, and only gets executed once when the loop starts.

```cpp
#include <iostream>
#include <array>
#include <cstddef> // std::size_t
int main() {
  std::array fibonacci{ 0, 1, 1, 2, 3, 5, 8, 13};
  std::size_t i{0};
  for (auto number : fibonacci) {
    std::cout << number << ", Index: " << i << '\n';
    ++i;
  }
}
```
▶

⬇

```cpp
#include <iostream>
#include <array>
#include <cstddef> // std::size_t
int main() {
  std::array fibonacci{ 0, 1, 1, 2, 3, 5, 8, 13};
  for (std::size_t i{0}; auto number : fibonacci) {
    std::cout << number << ", Index: " << i << '\n';
    ++i;
  }
}
```
▶

std::array
〇〇〇〇〇〇〇〇〇〇

Arrays and Loops
〇〇〇〇〇〇

Multidimensional Arrays
〇〇〇

std::vector
〇〇〇〇

std::tuple, std::pair
〇〇〇〇〇

Multidimensional Arrays

std::array
○○○○○○○○○○

Arrays and Loops
○○○○○○

Multidimensional Arrays
●○○

std::vector
○○○○

std::tuple, std::pair
○○○○○

Stony Brook
University

2D Arrays Using std::array

The elements of an array can be of any data type, including arrays! An array of arrays is called a **multidimensional array**.

A 2D array with m rows and n columns:

```
std::array < std::array<data_type, n >, m > array_name;
```

represents each row

Different ways of initialization 2D arrays:

```cpp
#include <iostream>
#include <array>
int main(){
  std::array<std::array<int, 4>, 2> arr1; // 2D array declaration
  std::array<std::array<int, 4>, 2> arr2 {}; // Zero initialization of a 2*4 2D array

  // Initialization a 2*4 2D array
  std::array<std::array<int, 4>, 2> arr3 {1, 2, 3, 4, 5, 6, 7, 8};
  std::array<std::array<int, 4>, 2> arr4 {{ {1, 2, 3, 4}, // row 0
                                            {5, 6, 7, 8} // row 1
                                         }}; // it needs additional set of {}
  std::array arr5 { std::array{1, 2, 3, 4},    // row 0, C++17 format
                    std::array{5, 6, 7, 8} };  // row 1
}
```
▶

Accessing Array Elements in 2D Array

The array elements can be accessed using the [·] operator or at(·) member function:

array_name[row_no][col_no]

array_name.at(row_no).at(col_no)

```cpp
#include <iostream>
#include <array>
int main() {
  std::array<std::array<int, 4>, 3> arr {{ {77, 85, 95, 4}, // row 0
                                           {3, 9, 82, 91},  // row 1
                                           {6, 43, 25, 95}  // row 2
                                         }};
  std::cout << arr[2][1] << '\n'; // prints the elment 43
  std::cout << arr.at(2).at(1) << '\n'; // prints the elment 43
}
```
▶

Notes:

- Arrays count starting from 0.
- at(·) does bounds checking, but operator [·] does not.

```
[0][0]  [0][1]  [0][2]  [0][3]  [0][4] // row 0
[1][0]  [1][1]  [1][2]  [1][3]  [1][4] // row 1
[2][0]  [2][1]  [2][2]  [2][3]  [2][4] // row 2
```

std::array
〇〇〇〇〇〇〇〇〇〇

Arrays and Loops
〇〇〇〇〇〇

Multidimensional Arrays
〇〇●

std::vector
〇〇〇〇

std::tuple, std::pair
〇〇〇〇〇

Stony Brook University

2D Arrays and Loops

```cpp
#include <iostream>
#include <array>
#include <cstddef> // std::size_t
int main (){
    std::array<std::array<int, 4>, 2> myArray {{ {1, 2, 3, 4}, // row 0
                                                 {5, 6, 7, 8}  // row 1
                                                }};
    for (std::size_t row{0}; row < myArray.size(); ++row) { // step through the rows
        for (std::size_t col{0}; col < myArray[0].size(); ++col){ // step through each element in the row
            std::cout << myArray[row][col] << ' ';
        }
    }

    std::cout << '\n'; // Range-based for Statement

    for (auto rowElement : myArray) {  // step through the rows
        for ( auto colElement : rowElement) {  // step through each element in the row
            std::cout << colElement << ' ';
        }
    }
}
```

▶

std::array
〇〇〇〇〇〇〇〇〇〇

Arrays and Loops
〇〇〇〇〇〇

Multidimensional Arrays
〇〇〇

std::vector
〇〇〇〇

std::tuple, std::pair
〇〇〇〇〇

std::vector

std::array	Arrays and Loops	Multidimensional Arrays	**std::vector**	std::tuple, std::pair
○○○○○○○○○○	○○○○○○	○○○	●○○○	○○○○○

Stony Brook University

std::vector

std::vector provides **dynamic array** functionality. This means you can create arrays that have their length set/change at **run-time** and std::vector will dynamically allocate memory for its contents as requested. It is defined in the <vector> header, inside the std namespace.

❖ std::vector **declaration:**

> std::vactor<type> arrayName;

❖ std::vector initialization: It can be initialized using list initialization, initializer lists, or parenthesis initialization.

The vector will self-resize to match the number of elements provided.

Brace Initialization {} is used for initialization with specific values. **Parenthesis Initialization** () is used for initialization to a specific size (the values will be value initialized).

```cpp
#include <vector>
int main() {
  std::vector<int> v1; // declares an integer vector
  v1 = { 0, 1, 2, 3, 4 }; // vector length is now 5
  v1 = { 9, 8, 7 }; // vector length is now 3
  std::vector<int> v2 { 9, 7, 5, 3, 1 }; // list initialization
  std::vector<int> v3 = { 9, 7, 5, 3, 1 }; // initializer list
  std::vector<int> v4 ( 3 ); // allocate 3 elements with values 0
  std::vector<int> v5 ( 3, 4 ); // allocate 3 elements with values 4
  // as with std::array, the type can be omitted since C++17
  std::vector v7 { 9, 7, 5, 3, 1 }; // deduced to std::vector<int>
}
```
▶

• Resizing a vector is computationally expensive, so you should strive to minimize the number of times you do so. If you need a vector with a specific number of elements, you can use parenthesis initialization to do so.

std::vector

- Just like std::array, accessing array elements can be done via [] operator (which does no bounds checking) or the at() function (which does bounds checking):

  Note: In either case, if you request an element that is off the end of the array, the vector will not automatically resize.

```cpp
#include <iostream>
#include <vector>
int main() {
    std::vector v { 9, 7, 5, 3, 1 };
    v[5] = 2; // no bounds checking
    v.at(3) = 4; // does bounds checking
    std::cout << v.at(0) << "\n";
    std::cout << v.size() << "\n";
}
```
▶

- Just like std::array, the size() member function can be used to retrieve the length of the std::vector.

- Just like std::array, always pass std::vector by **reference** or const **reference**.

```cpp
#include <iostream>
#include <vector>
void printLength(const std::vector<int>& v) {
    std::cout << "The length is: " << v.size() << '\n';
}
int main() {
    std::vector<int> v1{ 9, 7, 5, 3, 1 };
    printLength(v1);
    std::vector<int> v2 {};
    printLength(v2);
}
```
▶

std::array   Arrays and Loops   Multidimensional Arrays   **std::vector**   std::tuple, std::pair
○○○○○○○○○○   ○○○○○○   ○○○   ○○●○   ○○○○○

Stony Brook University

std::vector

Adding and
Removing Elements

Iterating Over
Elements

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Add an element to the end
    numbers.push_back(6);

    // Iterate and print the elements
    for (const auto& num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Remove the last element
    numbers.pop_back();

    // Print size
    std::cout << "Size: " << numbers.size() << std::endl;
}
```

▶

std::array
ooooooooooo

Arrays and Loops
oooooo

Multidimensional Arrays
ooo

std::vector
ooo●

std::tuple, std::pair
ooooo

Resizing std::vector

Vectors can be resized using member function resize(.) to be larger or smaller. Always, the existing element values are preserved. In resizing to a larger vector, new elements are initialized to the default value for the type (e.g., 0 for integers).

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector v{ 0, 1, 2 };
  v.resize(5); // set size to 5
  std::cout << "The length is: " << v.size() << '\n';

  for (auto i : v)
    std::cout << i << ' ';
}
```
▶

```
The length is: 5
0 1 2 0 0
```

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector v{ 0, 1, 2, 3, 4 };
  v.resize(3); // set size to 3
  std::cout << "The length is: " << v.size() << '\n';

  for (auto i : v)
    std::cout << i << ' ';
}
```
▶

```
The length is: 3
0 1 2
```

std::tuple, std::pair

std::tuple

In C++, std::tuple is a template class that can hold a **fixed number of elements** of **potentially different types**. It is defined in the <tuple> header and provides a way to group multiple values into a single object.

```cpp
#include <iostream>
#include <tuple>

int main() {
    std::tuple<int, double, std::string> t1; // Default initialization
    std::tuple<int, double, std::string> t2 {10, 15.5, "Hello"};
    auto t3 = std::make_tuple(20, 25.5, "World");

    int i = std::get<0>(t3);
    double d = std::get<1>(t3);
    std::string s = std::get<2>(t3);

    std::cout << "i: " << i << ", d: " << d << ", s: " << s << std::endl;
}
```

▶

std::array
○○○○○○○○○○

Arrays and Loops
○○○○○○

Multidimensional Arrays
○○○

std::vector
○○○○

std::tuple, std::pair
○●○○○○

std::tuple

Returning multiple values from a function.

```cpp
#include <iostream>
#include <tuple>

std::tuple<int, double, std::string> get_tuple() {
    return std::make_tuple(1, 2.5, "example");
}


int main() {
    auto t = get_tuple();
    std::cout << "Tuple values: " << std::get<0>(t) << ", " << std::get<1>(t) << ", " << std::get<2>(t) << std::endl;
}
```

▶

std::pair

In C++, std::pair is a simple container defined in the <utility> header that holds two **potentially different type** objects as a single unit. A pair is a specific case of a std::tuple with exactly two elements.

```cpp
#include <iostream>
#include <utility>

int main() {
    std::pair<int, std::string> p1;
    std::pair<int, std::string> p2{10, "Hello"};
    auto p3 = std::make_pair(20, "World");

    p1.first = 5;
    p1.second = "Hi";

    int number = p2.first;      // Access first element
    std::string text = p2.second; // Access second element

    std::cout << "p1: " << p1.first << ", " << p1.second << std::endl;
    std::cout << "p2: " << number << ", " << text << std::endl;
    std::cout << "p3: " << p3.first << ", " << p3.second << std::endl;
}
```
▶

std::array
○○○○○○○○○○

Arrays and Loops
○○○○○○

Multidimensional Arrays
○○○

std::vector
○○○○

std::tuple, std::pair
○○○●○

std::pair

Returning multiple values from a function.

```cpp
#include <iostream>
#include <utility>

std::pair<int, std::string> get_id_name() {
    return std::make_pair(1, "John");
}

int main() {
    std::pair<int, std::string> id_name = get_id_name();
    std::cout << "ID: " << id_name.first << ", Name: " << id_name.second << std::endl;
}
```

▶

std::array
○○○○○○○○○○

Arrays and Loops
○○○○○○

Multidimensional Arrays
○○○

std::vector
○○○○

std::tuple, std::pair
○○○○●

Common C++ Standard Library classes

std::string, std::vector, std::array, std::tuple, and std::pair are all class types. Therefore, when you create an object of any of these types, you are instantiating a class object. Moreover, when you call a function using these objects, you are calling a member function.

```cpp
#include <string>
#include <array>
#include <vector>
#include <iostream>

int main() {
    std::string s { "Hello, world!" }; // instantiate a string class object
    std::array<int, 3> a { 1, 2, 3 }; // instantiate an array class object
    std::vector<double> v { 1.1, 2.2, 3.3 }; // instantiate a vector class object
    std::cout << "length: " << s.length() << '\n'; // call a member function
}
```
▶