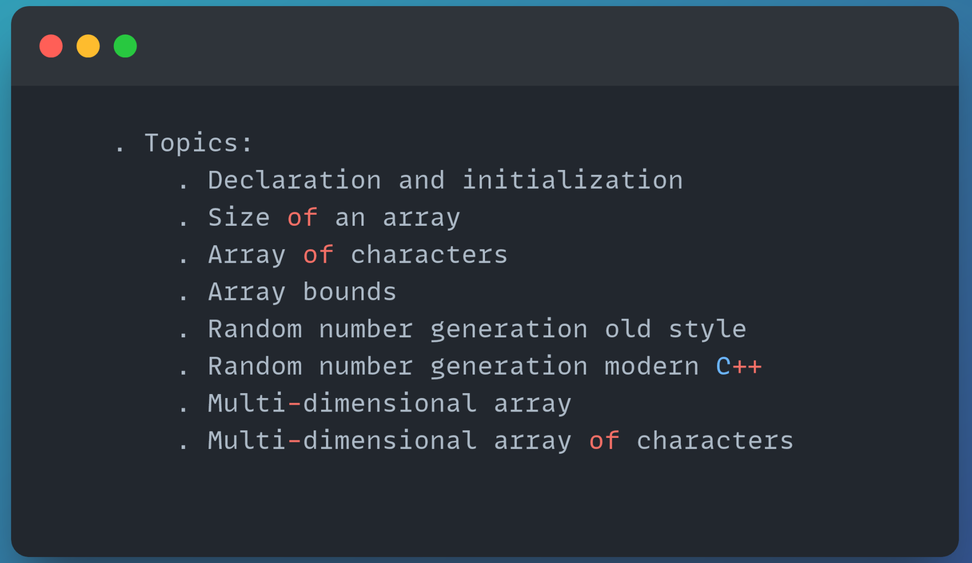
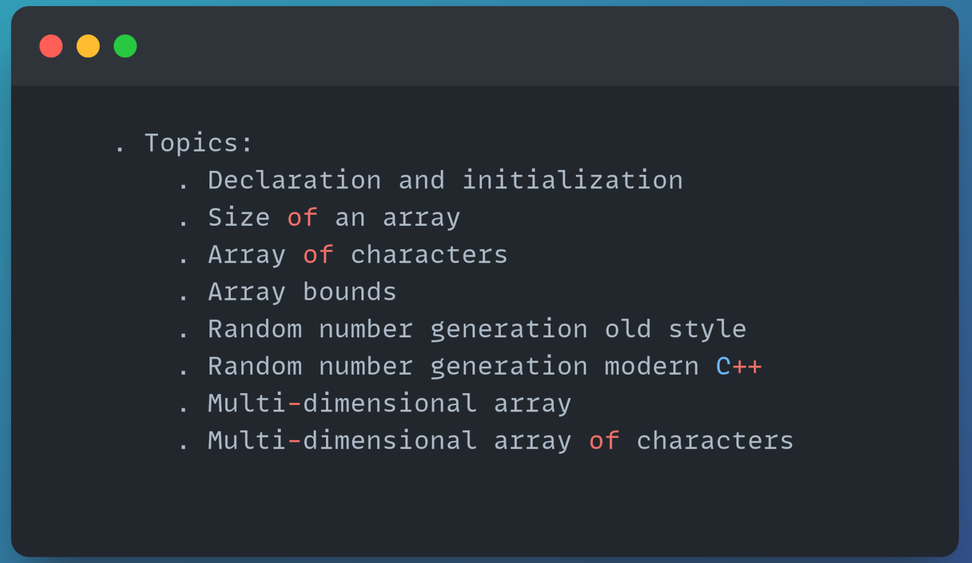
Raw arrays



Attention

In modern C++, it's generally **not recommended** to use raw arrays due to their limitations and potential safety risks. Instead, you should prefer standard library containers like std::vector, std::array, or other container types, as they provide better safety, flexibility, and ease of use.

Raw arrays



Attention

In modern C++, it's generally **not recommended** to use raw arrays due to their limitations and potential safety risks. Instead, you should prefer standard library containers like std::vector, std::array, or other container types, as they provide better safety, flexibility, and ease of use.

Raw arrays

```cpp
//Declaration and initialization
constexpr size_t array_size {5};

// Declare an array of ints
int scores [array_size]; // Junk data

// Read data
fmt::println( "scores [0]: {}", scores[0] );
fmt::println( "scores [1]: {}", scores[1] );

// Read with a loop
for( size_t i {0} ; i < array_size ; ++i){
    fmt::println("scores [{}]: {}", i, scores[i] );
}

scores[0] = 20;
scores[1] = 21;
scores[2] = 22;

//Print the data out
for( size_t i {0} ; i < array_size ; ++i){
    fmt::println("scores [{}] : {}", i, scores[i] );
}
```

Raw arrays

```
// Declare and initialize at the same time
 double salaries[5] {12.7, 7.5, 13.2, 8.1, 9.3};
 for(size_t i{0}; i < 5; ++i){
     fmt::println("salaries [{}]: {}", i, salaries[i] );
 }

 // If you don't initialize all the elements, those you leave out
 // are initialized to 0
 int families[5] {12, 7, 5};
     for(size_t i{0}; i < 5; ++i){
     fmt::println("families [{}]: {}", i, families[i] );
 }

 // Omit the size of the array at declaration
 int class_sizes[] {10,12,15,11,18,17,23,56};
 // Will print this with a range based for loop

 for(auto value : class_sizes){
     fmt::println("value: {}", value);
 }

 // Read only arrays
 const int birds[] {10,12,15,11,18,17,23,56};
 birds[2] = 8;   //Error
```

Raw arrays

```
// Sum up scores array, store result in sum
int scores[]{ 2, 5, 8, 2, 5, 6, 9 };
int sum{ 0 };

for (int element : scores) { sum += element; }
fmt::println("Score sum: {}", sum);
```

Size of a raw array

```cpp
//method1
int scores[]{ 1, 2, 5 };
int count{ std::size(scores) };// std::size( C++17)

fmt::println("sizeof(scores) : {}", sizeof(scores));    //12
fmt::println("sizeof(scores[0]) : {}", sizeof(scores[0]));  //4
fmt::println("count : {}", count);  //3
```

```cpp
//method2
int scores[]{ 1, 2, 5 };
int count {sizeof(scores)/sizeof(scores[0])};
```

Size of a raw array

```cpp
//method1
int scores[]{ 1, 2, 5 };
int count{ std::size(scores) };// std::size( C++17)

fmt::println("sizeof(scores) : {}", sizeof(scores));    //12
fmt::println("sizeof(scores[0]) : {}", sizeof(scores[0]));  //4
fmt::println("count : {}", count);  //3
```

```cpp
//method2
int scores[]{ 1, 2, 5 };
int count {sizeof(scores)/sizeof(scores[0])};
```

Arrays of characters

```
// Declare array
char message[5]{ 'H', 'e', 'l', 'l', 'o'}; //No space for null terminator. BAD. May crash your program
//char message[6]{ 'H', 'e', 'l', 'l', 'o', '\0' }; // Good. Null terminator included
//char message [] { 'H', 'e', 'l', 'l', 'o', '\0' }; // Good. Null terminator included
//char message[] { 'H', 'e', 'l', 'l', 'o' }; // Deduced size 5. No space for null terminator. BAD.
//char message[6]{ 'H', 'e', 'l', 'l', 'o' }; // Good. Null terminator is auto filled in. I wouldn't rely on this though.

int data[5] {1,2,3,3,3};
fmt::println( "data: {}",  data );    // Error: fmt knows how to print arrays of characters. Not arrays of any other type.
fmt::println("message: {}", message); // print the array through fmt::print or std::cout


// Print out the array through looping
fmt::println( "message : ");
for( auto c : message){
    fmt::println("{}", c );
}

//Change characters in our array
message[1] = 'a';

//Print out the array through looping
fmt::println( "message : ");
for( auto c : message){
    fmt::println("{}", c );
}
```

C-Strings

```cpp
// If a character array is null terminated, it's called as C-String
char message1 [] {'H','e','l','l','o','\0'};
fmt::println( "message1 : {}", message1 );
fmt::println( "sizeof(message1) : {}", sizeof(message1) );  // 6

char message2 [6] {'H','e','l','l','o'};    // The null terminator is auto filled in.
fmt::println( "message2 : {}", message2 );
fmt::println( "sizeof(message2) : {}", sizeof(message2) );  // 6

char message3 [] {'H','e','l','l','o'}; // This is not a c string ,
                                        //as there is not null character
//fmt::println( "message3 : {}", message3 );  // Bad. May crash your program
fmt::println( "sizeof(message3) : {}", sizeof(message3) );  // 5

// String literal
char message4 [] {"Hello"}; // The null terminator is auto filled in.
fmt::println( "message4 : {}", message4 );
fmt::println( "sizeof(message4) : {}", sizeof(message4) );  // 6
```

Built in arrays: No bound checking

```
int numbers[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };

// Read beyond bounds : May read garbage or crash your program
fmt::println("numbers[12] : {}",  numbers[12]);

// Write beyond bounds. The compiler allows it. But you don't own
// the memory at index 12, so other programs may modify it and your
// program may read bogus data at a later time. Or you can even
// corrupt data used by other parts of your ptogram

numbers[129] = 1000;
fmt::println("numbers[129]: {}", numbers[129]);

fmt::println("Program ending....");
```

Random numbers

```cpp
std::srand(std::time(0));// Seed

int main(){

    int random_num = std::rand();
    fmt::println("random_num : {}",random_num ); // 0 ~ RAND_MAX

}
```

Random numbers

```cpp
std::srand(std::time(0));// Seed

int main(){

    int random_num;

    //Generate in a given range [ 1~10 ]
    int random_num = std::rand() % 10 + 1;// [1~10]
    fmt::println("random_num : {}", random_num);// 0 ~ 10
    for (size_t i{ 0 }; i < 20; ++i) {
        random_num = std::rand() % 10 + 1;
        fmt::println("random_num {} : {}", i, random_num);// 0 ~ RAND_MAX

}
```

Random numbers

Using std::rand and std::srand for generating random numbers is considered outdated in modern C++. The C++ 11 standard introduced a more robust and flexible random number generation library in the <random> header. Here are some drawbacks of std::rand:

- **Predictability**: The numbers generated by std::rand are not very random and can be predictable if the seed is known.
- **Limited range**: std::rand generates numbers in the range [0, RAND_MAX], which may not be sufficient for some applications.
- **Global state**: std::rand and std::srand use global state, which can be problematic in multi-threaded programs
- **Lack of flexibility**: The <random> header library provides more flexibility and control over the distribution and range of random numbers.

Random numbers

```cpp
// Random numbers ( C++17 )
std::random_device rd;  // Provides seeds with less predictability
std::mt19937 mersenne{ rd() };  // This is the generator, initialized with a random seed
std::uniform_int_distribution die{ 1, 6 };  // We want uniform distribution.

for (size_t count{ 0 }; count < 48; ++count) {
    fmt::println("{}", die(mersenne));
}
```

Using random numbers

```cpp
//We have a collection of predictions
//We want to randomly select one of them and print it out

// The predictions are stored as strings in a vector
std::vector<std::string> predictions { "a lot of kids running in the backyard!",
    "a lot of empty beer bootles on your work table.",
    "you Partying too much with kids wearing weird clothes.",
    "you running away from something really scary",
    "clouds gathering in the sky and an army standing ready for war",
    "dogs running around in a deserted empty city",
    "a lot of cars stuck in a terrible traffic jam",
    "you sitting in the dark typing lots of lines of code on your dirty computer",
    "you yelling at your boss. And oh no! You get fired!",
    "you laughing your lungs out. I've never seen this before.",
    "Uhm , I don't see anything!"
};

//Use the marsenne twister to generate a random number and store that in a variable named index.
std::random_device rd;  // Provides seeds with less predictability
std::mt19937 mersenne{ rd() };  // Initialize our mersenne twister with a random seed
const int size(predictions.size()-1); // The distribution expects constant expressions in it's initializer.
std::uniform_int_distribution distr{ 0, size };  // We want uniform distribution.`
auto index = distr(mersenne);

fmt::println("Prediction :I see {}", predictions[index]);
```