Characters and strings

```
/*

    . Characters and strings
        . Handling characters
            . The cctype header
        . Handling C-strings
        . Handling std::string
        . Escape sequences and raw string literals
        . Handling std::string_view
        . And more


*/
```

Handling characters

```
characters_and_strings::is_al_num();
characters_and_strings::is_alphabetic();
characters_and_strings::is_blank();
characters_and_strings::is_upercase();
characters_and_strings::is_digit();
characters_and_strings::change_to_lowercase();
```

Alphabetic

```
// Check if character is alphabetic
fmt::println( "std::isalpha : ");
fmt::println(   "C is alphabetic : {}", std::isalpha('e') ); // 1
fmt::println(   "^ is alphabetic : {}", std::isalpha('^') ); // 0
fmt::println(   "7 is alphabetic : {}", std::isalpha('7') ); // 0

if(std::isalpha('e')){
    fmt::println( "e is alphabetic" );
}else{
    fmt::println( "e is NOT alphabetic" );
}
```

Alphanumeric

```cpp
// Check if character is alphanumeric
fmt::println( "std::isalnum : ");
fmt::println( "C is alphanumeric : {}", std::isalnum('C') );
fmt::println( "^ is alphanumeric : {}", std::isalnum('^') );

//Can use this as a test condition
char input_char {'*'};
if(std::isalnum(input_char)){
    fmt::println( "{} is alhpanumeric.", input_char );
}else{
    fmt::println(  "{} is not alphanumeric.", input_char );
}
```

Blank

```cpp
// Check if a character is blank
char message[] {"Hello there. How are you doing? The sun is shining."};
fmt::println( "message : {}", message );

//Find and print blank index
size_t blank_count{};
for (size_t i{0} ; i < std::size(message); ++i){
    //fmt::println( "Value : " << message[i] );
    if(std::isblank(message[i])){
        fmt::println( "Found a blank character at index : [{}]",  i );
        ++blank_count;
    }
}
fmt::println( "In total, we found {} blank characters.", blank_count);
```

Uppercase

```cpp
// Check if character is lowercase or uppercase
fmt::println( "std::islower and std::isupper : ");
fmt::println("");
char thought[] {"The C++ programming language is one of the most used on the planet."};
size_t lowercase_count{};
size_t upppercase_count{};

//Print original string for ease of comparison on the terminal
fmt::println( "Original string  : {}" , thought );

for( auto character : thought){
    if(std::islower(character)){
        fmt::print("{} ", character);
        ++lowercase_count;
    }
    if(std::isupper(character)){
        ++upppercase_count;
    }
}
fmt::println("");
fmt::println( "Found {} lowercase characters and {} uppercase characters.", lowercase_count, upppercase_count);
```

Digit

```cpp
// Check if a character is a digit
fmt::println("");
fmt::println( "std::isdigit : ");

char statement[] {"Mr Hamilton owns 221 cows. That's a lot of cows! The kid exclamed."};
fmt::println( "statement : {}", statement );

size_t digit_count{};

for(auto character : statement){
    if(std::isdigit(character)){
        fmt::println( "Found digit : {}", character );
        ++digit_count;
    }
}
fmt::println( "Found {} digits in the statement string", digit_count );
```

Change case

```cpp
    // Turning a character to lowercase using the std::tolower() function
    fmt::println("");
    fmt::println("std::tolwer and std::toupper: ");
    char original_str[]{ "Home. The feeling of belonging" };
    char dest_str[std::size(original_str)];

    // Turn this to uppercase. Change the array in place
    for (size_t i{}; i < std::size(original_str); ++i) {
        dest_str[i] = std::toupper(original_str[i]);
    }

    fmt::println("Original string: {}", original_str);
    fmt::println("Uppercase string: {}", dest_str);

    // Turn this to lowercase. Change the array in place
    for (size_t i{}; i < std::size(original_str); ++i) {
        dest_str[i] = std::tolower(original_str[i]);
    }
    fmt::println("Lowercase string: {}", dest_str);
}
```

Handling C-strings

```
#include <cstring>
handling_c_strings::cstring_strlen();
handling_c_strings::cstring_strcmp();
handling_c_strings::cstring_strncmp();
handling_c_strings::cstring_find_first_occurence_version_1();
handling_c_strings::cstring_find_first_occurence_version_2();
handling_c_strings::cstring_find_last_occurence();
handling_c_strings::cstring_concatenation_version_1();
handling_c_strings::cstring_concatenation_version_2();
handling_c_strings::cstring_strncat();
handling_c_strings::cstring_strcpy();
handling_c_strings::cstring_strncpy();
```

Handling C-strings

```
export void cstring_strlen(){
    //std::strlen : Find the length of a string
    // real arrays and those decayed into pointers
    const char message1 [] {"The sky is blue."};

    //Array decays into pointer when we use const char*
    const char* message2 {"The sky is blue."};
    fmt::println( "message1 : {}",  message1 );
    fmt::println( "message2 : {}",  message2 );

    //strlen ignores null character
    fmt::println( "strlen(message1) : {}", std::strlen(message1) );

    // std::sizeof includes the null character
    fmt::println( "sizeof(message1) : {}",  sizeof(message1) );

    //strlen still works with decayed arrays
    fmt::println( "strlen(message2) : {}" , std::strlen(message2) );

    //std::size prints size of pointer
    fmt::println( "sizeof(message2) : {}",  sizeof(message2) );
}

//The rest are seen in the code session.
```

Handling std::string

```
handling_std_strings::std_string_declaration();
handling_std_strings::std_string_concatenation();
handling_std_strings::std_string_access_characters();
handling_std_strings::std_string_size_and_capacity();
handling_std_strings::std_string_modify();
```

Handling std::string

```
export void std_string_declaration(){
    std::string full_name;// Empty string
    std::string planet{ "Earth. Where the sky is blue" };// Initialize with string literal
    std::string prefered_planet{ planet };// Initialize with other existing string
    std::string message{ "Hello there", 5 };// Initialize with part of a string literal.
                                            // Contains hello
    std::string weird_message(4, 'e');// Initialize with multiple copies of a char
                                      //  contains eeee
    std::string greeting{ "Hello World" };
    std::string saying_hello{ greeting, 6, 5 };// Initialize with part of an existing std::string
                                               //  starting at index 6, taking 5 characters.
                                               // Will contain World.

    // Changing std::string at runtime
    planet = "Earth. Where the sky is blue Earth. Where the sky is blue Earth. Where ";
    fmt::println("planet: {}", planet);


    // Use a raw array
    const char *planet1{ "Earth. Where the sky is blue Earth." };
    planet1 = "Earth. Where the sky is blue Earth. Where the sky is blue Earth. Where ";
    std::string planet2{ planet1 };
    fmt::println("planet2: {}", planet2);
}

//The rest are seen in the code session.
```

Escape sequences and Raw string literals

```cpp
//The problem
//New line character
for(size_t i{0} ; i  < 10 ; ++i){
    fmt::print( "Hello\n");
}

// Escape the double quote character
fmt::println( "He said \"Get out of here immediately!\"" );

// Simulating a todo list and systemn paths
std::string todo_list{ "\tClean the house\n\tWalk the dog\n\tDo laundry\n\tPick groceries" };
std::string windows_path{ "D:\\sandbox\\testProject\\hello.txt" };
std::string linux_path{ "/home/username/files/hello.txt" };
std::string hint{ " \"\\\\\" escapes a backslash character like \\." };

fmt::println( "todo_list: " );
fmt::println( "{}", todo_list );
fmt::println( "windows_path: {}" , windows_path );
fmt::println( "linux_path: {}" , linux_path );
fmt::println( "hint: {}" , hint );
```

Raw string literals

```cpp
//The solution: Raw string literals
std::string to_do_list {R"(
Clean the house
Walk the dog
Do laundry
Pick groceries)"};

// Raw string literals with assignments
std::string to_do_list =   R"(
        Clean the house
        Walk the dog
        Do Laundry
        Pick groceries
)";

// Raw string literals with assignments: cstring
const char* c_string {  R"(
        Clean the house
        Walk the dog
        Do Laundry
        Pick groceries
    )"} ;
```

Raw string literals

```
//Fixing other escaped strings

std::string windows_path1 {R"(D:\sandbox\testProject\hello.txt)"};
std::string linux_path1 {R"(/home/username/files/hello.txt)"};
std::string hint1 {R"("\\" escapes a backslash character like \.)"};

fmt::println( "windows_path1 : {}" , windows_path1 );
fmt::println( "linux_path1 : {}" , linux_path1 );
fmt::println( "hint1 : {}" , hint1 );

// Problematic raw string literals
std::string sentence{ R"--(The message was "(Stay out of this!)")--" };
fmt::println("sentence: {}", sentence);
```

std::string_view

the problem

```cpp
export void string_view_the_problem(){
    // Showing the problem
    std::string string {"Hello"};
    std::string string1 {string}; // Copy
    std::string string2 {string}; // Copy
}
```

the solution

```cpp
export void string_view_the_solution(){
    // Using string_view
    std::string_view sv0
    {"Hellooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo"};
    std::string_view sv1 {sv0}; // View viewing the hello literal
    std::string_view sv2 {sv1}; // Another view viewing hello

    fmt::println( "Size of string_view: {}" , sizeof(std::string_view) );
    fmt::println( "size of sv1: {}" , sizeof(sv1) );

    fmt::println( "sv0: {}" , sv0 );
    fmt::println( "sv1: {}" , sv1 );
    fmt::println( "sv2: {}" , sv2 );
}
```

std::string_view: constructors

```cpp
export void string_view_construction(){
    // Constructing string_view's
    std::string string3 {"Regular std::string"};
    const char * c_string {"Regular C-String"};
    const char char_array[]{"Char array"}; // Null terminated
    char char_array2[]{'H','u','g','e'}; // Non null terminated char array

    std::string_view sv3{"String litteral"};
    std::string_view sv4{string3};
    std::string_view sv5{c_string};
    std::string_view sv6{char_array};
    std::string_view sv7{sv3};    //From another string view
    std::string_view sv8{char_array2,std::size(char_array2)};//Non null terminated char array
                                                             //Need to pass in size info
    fmt::println( "sv3: {}" , sv3 );
    fmt::println( "sv4: {}" , sv4 );
    fmt::println( "sv5: {}" , sv5 );
    fmt::println( "sv6: {}" , sv6 );
    fmt::println( "sv7 (constructed from other string_view): {}" , sv7 );
    fmt::println( "Non null terminated string with std::string_view: {}" , sv8 );
}
```

std::string_view: change propagation

```
export void std_string_view_visualizes_original_string(){
    // Changes to the original string are reflected in the string_view
    char word [] {"Dog"};
    std::string_view sv9{word};

    fmt::println( "word : {}" , sv9 );

    fmt::println( "Changing data: " );
    //Change the data
    word[2] = 't';

    fmt::println( "word : {}" , sv9 );
}
```

std::string_view view window

```
export void std_string_view_change_view_window(){
    // Changing the view window : SHRINKING
    const char * c_string1 { "The animals have left the region" };
    std::string_view sv10{c_string1};

    fmt::println( "sv10 : {}" , sv10 );

    sv10.remove_prefix(4); // Removes "The"

    //Prints : animals have left the region
    fmt::println( "View with removed prefix(4) : {}" , sv10 );

    sv10.remove_suffix(10); // Removes "the region"

    //Prints : animals have left
    fmt::println( "View with removed suffix(10) : {}" , sv10 );

    //Changing the view doesn't change the viewed string :
    fmt::println( "Original sv10 viewed string : {}" , c_string1 );
}
```

std::string_view lifetime

```
export void std_string_view_lifetime(){
    // String_view shouldn't outlive whatever it is viewing
    std::string_view sv11;

    {
        std::string string4{"Hello there"};
        sv11 = string4;
        fmt::println( "INSIDE --- sv11 is viewing: {}" , sv11 );
        //string4 goes out of scope here.
    }
    fmt::println( "OUTSIDE --- sv1 is viewing: {}" , sv11 );
}
```

std::string_view: the data method

```
export void std_string_view_data(){
    // data
    /*
    std::string_view sv13 {"Ticket"};
    fmt::println( "sv13: {}" , sv13 );
    fmt::println( "std::strlen(sv13.data()): {}" , std::strlen(sv13.data()) );
    */


    // Don't use data() on a modified view (through remove_prefix or remove_suffix)
    std::string_view sv14 {"Ticket"};
    sv14.remove_prefix(2);
    sv14.remove_suffix(2);

    //Length info is lost when you modify the view.
    //Using the data method is dangerous at this point.
    fmt::println( "{} has {}  characters(s) " ,sv14,  std::strlen(sv14.data()));
    fmt::println( "sv14.data() is {}" , sv14.data() );
    fmt::println( "sv14 is {}" , sv14 );
}
```

std::string_view

```
export void std_string_view_non_null_terminated_strings(){
    // Don't view non null terminated strings
    char char_array3[] {'H','e','l','l','o'};
    std::string_view sv15 {char_array3,std::size(char_array3)};

    fmt::println( "{} has {} characters(s)" ,sv15, std::strlen(sv15.data()) );
    fmt::println("sv15.data is : {}" ,  sv15.data() );
    fmt::println( "sv15 is: {}" , sv15 );
}

export void std_string_view_behaviors(){
    // std::string behaviors(methods) are available in string_view
    std::string_view sv16{ "There is a huge forest in that area" };

    fmt::println("sv16 is {}", sv16.length(), " characters long");
    fmt::println("The front character is: {}", sv16.front());
    fmt::println("The back character is: {}", sv16.back());
    fmt::println("Substring: {}", sv16.substr(0, 22));
}
```