Integer Modifiers, Characters, Text and Auto

```
/*
    . Topics:
        . Initialzation:
            . Braced initialization
            . Function style initialization
            . Assignment initialization
            . Integer modifiers
            . Floating point numbers:
                . Precision
                . Scientific notation
            . Infinity and nan
            . Characters and text
            . auto
*/
```

Braced initialization: Recommended

```
int elephant_count; // Initialized to junk.

int lion_count{};// Initializes to zero

int dog_count{ 10 };// Initializes to 10

int cat_count{ 15 };// Initializes to 15

// Can use expression as initializer
int domesticated_animals{ dog_count + cat_count };

//int new_number{doesnt_exist};

//int narrowing_conversion {2.9};//Error

fmt::println("Elephant count : {}", elephant_count);
fmt::println("Lion count : {}", lion_count);
fmt::println("Dog count : {}", dog_count);
fmt::println("Cat count : {}", cat_count);
fmt::println("Domesticated animal count : {}", domesticated_animals);
```

Functional Initialization

```
int apple_count(5);
int orange_count(10);
int fruit_count (apple_count + orange_count);
//int bad_initialization ( doesnt_exist3 + doesnt_exist4 );

//Information lost. less safe than braced initializers
int narrowing_conversion_functional (2.9);


fmt::println("Apple count : {}", apple_count);
fmt::println("Orange count : {}", orange_count);
fmt::println("Fruit count : {}", fruit_count);
fmt::println("Narrowing conversion : {}", narrowing_conversion_functional);//Will lose info
```

Assignment initialization

```
int bike_count = 2;
int truck_count = 7;
int vehicle_count = bike_count +truck_count;
int narrowing_conversion_assignment = 2.9;

fmt::println("Bike count: {}", bike_count);
fmt::println("Truck count: {}", truck_count);
fmt::println("Vehicle count: {}", vehicle_count);
fmt::println("Narrowing conversion: {}", narrowing_conversion_assignment);

//Check the size with sizeof
fmt::println("sizeof int: {}", sizeof(int));
fmt::println("sizeof truck_count: {}", sizeof(truck_count));
```

Integer modifiers: signed

```
// Integer modifiers
signed int value1{ 10 };
signed int value2{ -300 };

fmt::println("value1 : {}", value1);
fmt::println("value2 : {}", value2);
fmt::println("sizeof(value1) : {}", sizeof(value1));
fmt::println("sizeof(value2) : {}", sizeof(value2));

unsigned int value3{ 4 };
//unsigned int value4{-5}; // Compiler error.
```

Integer modifiers: short and long

```
// short and long
short short_var{ -32768 };//  2 Bytes
short int short_int{ 455 };//
signed short signed_short{ 122 };//
signed short int signed_short_int{ -456 };//
unsigned short int unsigned_short_int{ 456 };

int int_var{ 55 };// 4 bytes
signed signed_var{ 66 };//
signed int signed_int{ 77 };//
unsigned int unsigned_int{ 77 };

long long_var{ 88 };// 4 OR 8 Bytes
long int long_int{ 33 };
signed long signed_long{ 44 };
signed long int signed_long_int{ 44 };
unsigned long int unsigned_long_int{ 44 };

long long long_long{ 888 };// 8 Bytes
long long int long_long_int{ 999 };
signed long long signed_long_long{ 444 };
signed long long int signed_long_long_int{ 1234 };
unsigned long long int unsigned_long_long_int{ 1234 };
```

Fractional numbers

```
// Fractional numbers
 // Declare and initialize the variables
 /*
 float number1{ 1.12345678901234567890f };// Precision : 7
 double number2{ 1.12345678901234567890 };// Precision : 15
 long double number3{ 1.12345678901234567890L };

 // Print out the sizes
 fmt::println("sizeof float: {}", sizeof(float));
 fmt::println("sizeof double: {}", sizeof(double));
 fmt::println("sizeof long double: {}", sizeof(long double));

 // Precision
 fmt::println("number1 is: {}", number1);// 7 digits
 fmt::println("number2 is: {}", number2);// 15'ish digits
 fmt::println("number3 is: {}", number3);// 15+ digits


 // Float problems : The precision is usually too limited
 // for a lot of applications
 float number4 = 192400023.0f;// Error : narrowing conversion
 //Anything we can do better here?

 fmt::println("number4 : {}", number4);
```

Scientific notation for floating types

```
// Scientific notation
// What we have seen so far in terms of floating point types
// is fixed notation. There is another notation, scientific
// that is handy if you have really huge numbers or small numbers
// to represent

fmt::println("-------------------------");

double number5{ 192400023 };
double number6{ 1.92400023e8 };
double number7{ 1.924e8 };// Can ommit the lower 00023
                          // for simplicity if our application allows that.
double number8{ 0.00000000003498 };
double number9{ 3.498e-11 };// multiply with 10 exp(-11)

fmt::println("number5 is : {}", number5);
fmt::println("number6 is : {}", number6);
fmt::println("number7 is : {}", number7);
fmt::println("number8 is : {}", number8);
fmt::println("number9 is : {}", number9);
```

Infinity and nan

```cpp
// Infinity and Nan
double number10{ -5.6 };
double number11{};// Initialized to 0
double number12{};// Initialized to 0

// Infinity
double result{ number10 / number11 };


fmt::println("{} / {} yields {}", number10, number11, result);
fmt::println("{} + {} yields {}", result, number10, result + number10);

// NaN
result = number11 / number12;

fmt::println("{} / {} = {}", number11, number12, result);
```

booleans

```cpp
bool red_light{ false };
bool green_light{ true };

if (red_light == true) {
    fmt::println("Stop!");
} else {
    fmt::println("Go through!");
}

if (green_light) {
    fmt::println("The light is green!");
} else {
    fmt::println("The light is NOT green!");
}

// sizeof()
fmt::println("sizeof(bool) : {}", sizeof(bool));

      // Printing out a bool
// 1 ⟶ true
// 0 ⟶ false
fmt::println("red_light: {}", red_light);
fmt::println("red_light: {}", red_light);
fmt::println("green_light: {}", green_light);
```

Characters and text

```
char character1{ 'a' };
char character2{ 'r' };
char character3{ 'r' };
char character4{ 'o' };
char character5{ 'w' };

fmt::println("{}", character1);
fmt::println("{}", character2);
fmt::println("{}", character3);
fmt::println("{}", character4);
fmt::println("{}", character5);

// One byte in memory: 2^8 = 256 different values (0 ~ 255)

char value = 65;// ASCII character code for 'A'
fmt::println("value : {}", value);// A
fmt::println("value(int) : {}", static_cast<int>(value));
```

Collections of characters

```cpp
// const std::vector<char> characters {'a', 'r', 'r', 'o', 'w'};
const std::array<char, 5> characters{ 'a', 'r', 'r', 'o', 'w' };
fmt::println("{}", characters[0]);
fmt::println("{}", characters[1]);
fmt::println("{}", characters[2]);
fmt::println("{}", characters[3]);
fmt::println("{}", characters[4]);

// Print all characters in one go: compiler error.
//fmt::println("{},characters");

// Using std::string
fmt::println("Printing the string");
std::string message{ "arrow" };
fmt::println("{}", message);


// Doing more stuff with std::string
std::string greeting{ "Hello" };
greeting.append(" there");
fmt::println("{}", greeting);
```

auto

```cpp
auto var1{ 12 };
auto var2{ 13.0 };
auto var3{ 14.0f };
auto var4{ 15.0l };
auto var5{ 'e' };

// int modifier suffixes
auto var6{ 123u };// unsigned
auto var7{ 123ul };// unsigned long
auto var8{ 123ll };// long long

fmt::println("var1 occupies : {} bytes", sizeof(var1));
fmt::println("var2 occupies : {} bytes", sizeof(var2));
fmt::println("var3 occupies : {} bytes", sizeof(var3));
fmt::println("var4 occupies : {} bytes", sizeof(var4));
fmt::println("var5 occupies : {} bytes", sizeof(var5));
fmt::println("var6 occupies : {} bytes", sizeof(var6));
fmt::println("var7 occupies : {} bytes", sizeof(var7));
fmt::println("var8 occupies : {} bytes", sizeof(var8));
```