STL, Containers and iterators

```
/*
    . STL, Containers and iterators
        .#1: Common containers: std::vector and std::array

        .#2: Iterators
            . Playing with the basics of iterators.
            . Manually moving forward and printing both std::vector and std::array with one print function.
            . we can adjust the beginning and end of the collection for printing.
            . reverse iterators

        .#3: Constant iterators, std::begin, and std::end

        .#4: Generic container elements:
            . working with a container without knowing what it contains.
*/
```

std::vector and std::array

```cpp
//std::vector
// Constructing vectors
  std::vector<std::string> vec_str{"The", "sky", "is", "blue", "my", "friend"};
  fmt::println("vec1[1]  : {}", vec_str[1]);
  print_vec(vec_str);

  std::vector<int> ints1;
  print_vec(ints1); // Won't print anything, the vector has no content

  std::vector<int> ints2 = {1, 2, 3, 4};
  std::vector<int> ints3{11, 22, 33, 44};

  std::vector<int> ints4(20, 55); // A vector with 20 items, all initialized to 55
  fmt::print("ints4 : ");
  print_vec(ints4);

  // Be careful about uniform initialization
  std::vector<int> ints5{20, 55}; // A vector with 2 items: 20 and 55

  // Accessing elements
  fmt::println("vec_str[2] : {}", vec_str[2]);
  fmt::println("vec_str.at(3) : {}", vec_str.at(3));
  fmt::println("vec_str.front() : {}", vec_str.front());
  fmt::println("vec_str.back() : {}", vec_str.back());

  // Using the data method
  print_raw_array(vec_str.data(), vec_str.size());
```

std::vector and std::array

```cpp
//std::vector (contd)
// Adding and removing stuff
 fmt::println("Adding and removing stuff : ");

 fmt::print("ints1 : ");
 print_vec(ints1);

 // Pushing back
 ints1.push_back(100);
 ints1.push_back(200);
 ints1.push_back(300);
 ints1.push_back(500);
 fmt::print("ints1 : ");
 print_vec(ints1);

 // Popping back
 ints1.pop_back();
 fmt::print("ints1 : ");
 print_vec(ints1);
```

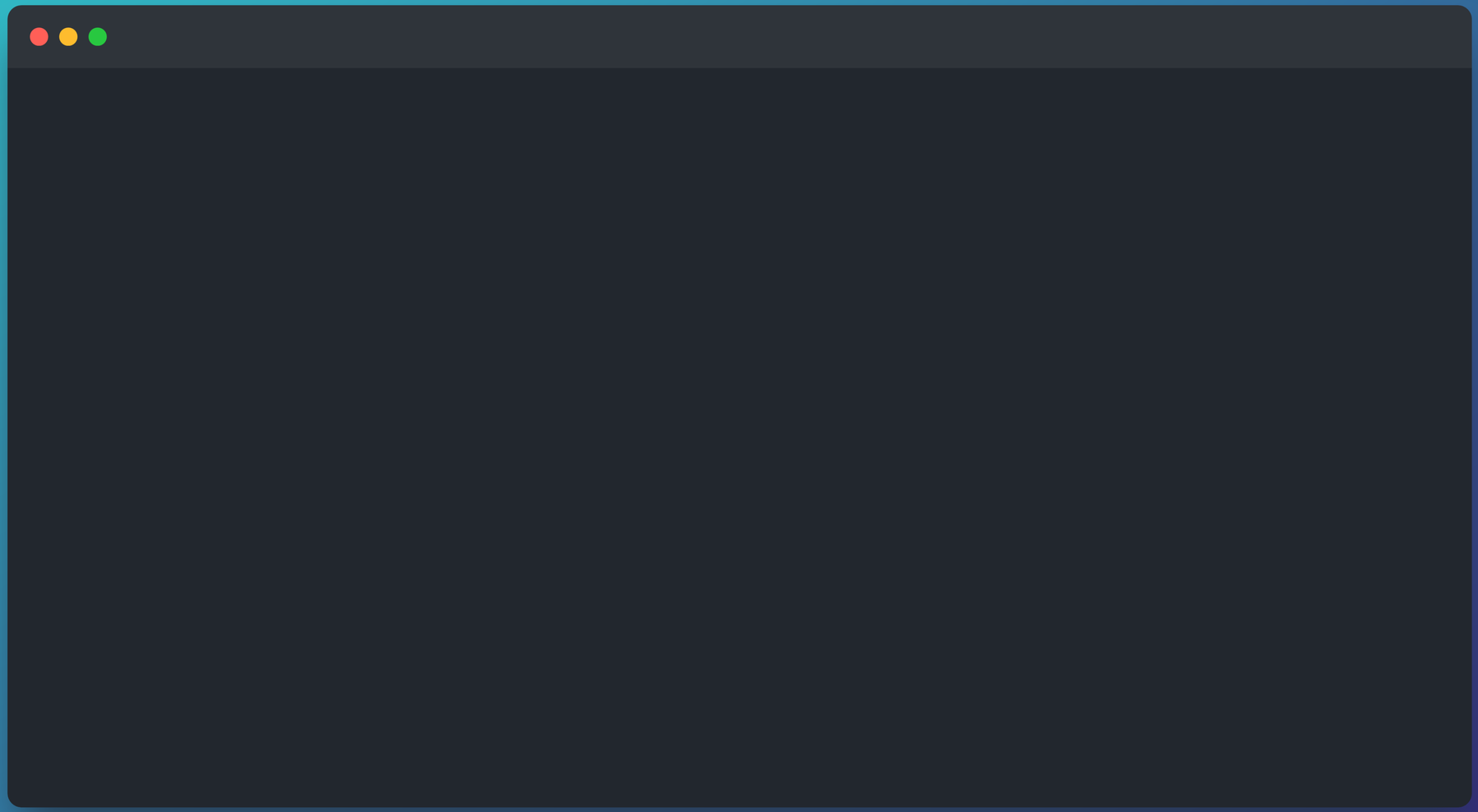
std::vector and std::array

```cpp
//std::array
std::array<int, 3> int_array1; // Will contain junk by default
std::array<int, 3> int_array2{1, 2}; // Will contain 1, 2, 0
std::array<int, 3> int_array3{}; // Will contain 0, 0, 0
std::array int_array4{1, 2}; // Compiler will deduce std::array<int, 2>
// std::array<int, 3> int_array5{1, 2, 3, 4, 5}; // Compiler error: More than enough elements
// Can deduce the type with auto.

fmt::print("int_array1 : ");
print_array(int_array1);

fmt::print("int_array2 : ");
print_array(int_array2);

fmt::print("int_array3 : ");
print_array(int_array3);

fmt::print("int_array4 : ");
print_array(int_array4);
```

std::vector and std::array

```cpp
//std::array (contd)
// Adding and removing stuff
// Can't really add stuff. Can specify content at initialization
// Can also fill the entire array with an element
fmt::println("");
fmt::println("Filling the array : ");
int_array1.fill(321);
int_array4.fill(500);
fmt::print("int_array1 : ");
print_array(int_array1);
fmt::print("int_array4 : ");
print_array(int_array4);

// Accessing elements
fmt::println("");
fmt::print("Accessing elements in an array: ");
fmt::println("int_array2[0] : {}", int_array2[0]);
fmt::println("int_array2.at(1) : {}", int_array2.at(1));
fmt::println("int_array2.front() : {}", int_array2.front());
fmt::println("int_array2.back() : {}", int_array2.back());
// data method
print_raw_array(int_array2.data(), int_array2.size());
```

std::vector and std::array: Demo time!



Iterators: Introduction

```
/*
    . Tinkering with functions that print the contents of a collection, using iterators.
        . Containers are manipulated in a unified way.
    . The main goals is to introduce iterators.
    . It can print both std::vector and std::array.
        . Any container, as long as it has begin() and end() functions, can be printed.
    . Most containers in the STL have these functions.
*/
```

Iterators: Introduction

```
std::vector ints1{ 11, 22, 33, 44 };
std::array ints2{ 100, 200, 300, 400 };

auto it_begin = ints1.begin();//begin returns an iterator pointing to the first element of the collection
auto it_end = ints1.end(); //end returns an iterator pointing to the element after the last element of the collection

fmt::println("first elt: {}", *it_begin);    //dereferencing the iterator gives the value
fmt::println("it == end_it: {}", (it_begin == it_end));

++it_begin;
fmt::println("second elt: {}", *it_begin);
fmt::println("it == end_it: {}", (it_begin == it_end));

++it_begin;
fmt::println("third elt: {}", *it_begin);
fmt::println("it == end_it: {}", (it_begin == it_end));

++it_begin;
fmt::println("fourth elt: {}", *it_begin);
fmt::println("it == end_it: {}", (it_begin == it_end));

/*
++it_begin; // Throws us out of the valid range
fmt::println("junk elt: {}", *it_begin);
fmt::println("it == end_it: {}", (it_begin == it_end));// true
*/
```

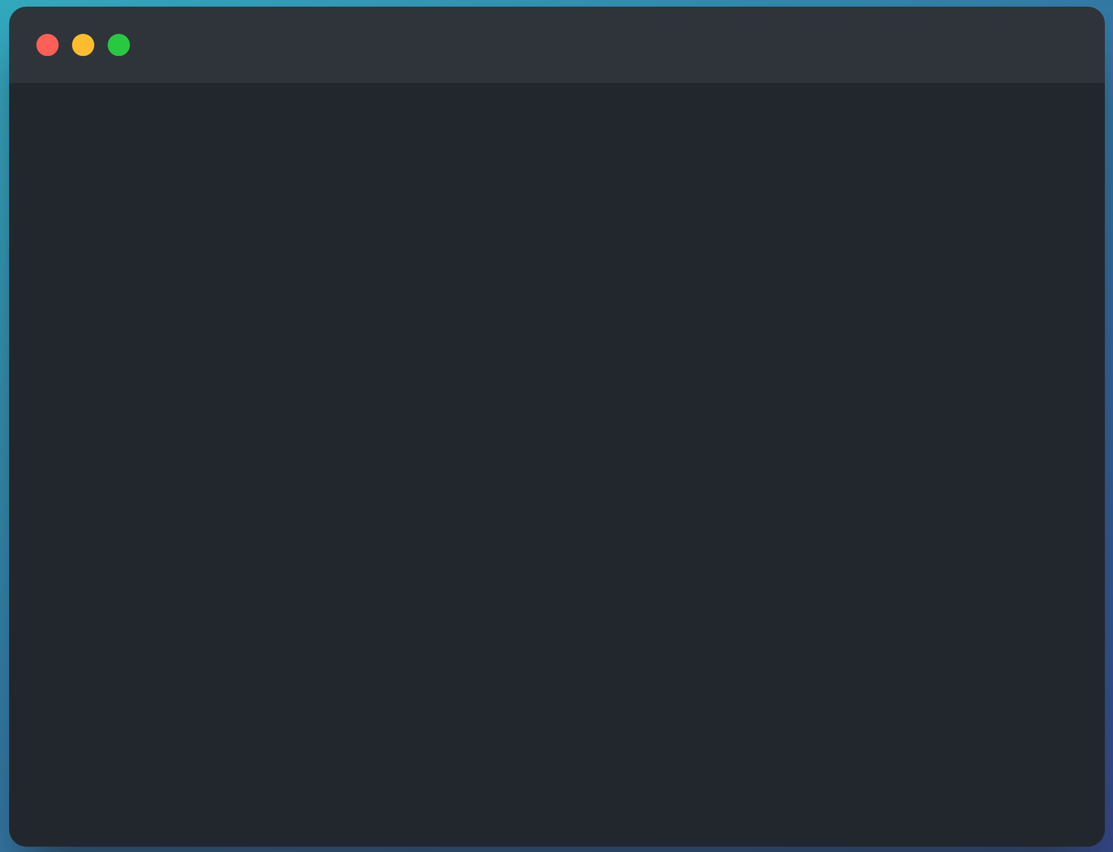
Iterators: Introduction

```cpp
export template<typename T>
void print_collection(const T &collection)
{
    auto it = collection.begin(); //begin returns an iterator pointing to the first element

    fmt::print(" [");
    while (it != collection.end()) {    //end returns an iterator pointing to the element after the last element
        fmt::print(" {}", *it); // iterators can be dereferenced to get the value, the idea is to make them behave like pointers
        ++it;   // iterators can be incremented
    }
    fmt::println(" ]");
}

// We can adjust the beginning and end of the collection for printing.
export template<typename T>
void print_collection(const T &collection, size_t begin_adjustment, size_t end_adjustment)
{
    // Adjudt begining and end
    auto start_point = collection.begin() + begin_adjustment;
    auto end_point = collection.end() - end_adjustment;

    fmt::print(" [");
    while (start_point != end_point) {
        fmt::print(" {}", *start_point);
        ++start_point;
    }
    fmt::println(" ]");
}
```

Iterators: Introduction

```cpp
//Adjust the beginning and end
fmt::println("Adjusting begining and end");
std::vector ints3{ 11, 22, 33, 44, 55, 66, 77 };
std::array ints4{ 100, 200, 300, 400, 500, 600 };

fmt::print("ints3: ");
containers_iterators_02::print_collection(ints3, 1, 3);
fmt::print("ints4: ");
containers_iterators_02::print_collection(ints4, 1, 1);

//Reverse iterators
fmt::println("Reverse iterators");
fmt::print("ints3: ");
auto rit = ints3.rbegin();
while (rit =/= ints3.rend()) {
    fmt::print(" {}", *rit);
    ++rit;
}
```

Iterators: Introduction - Demo time!



Constant iterators: cbegin and cend

```cpp
//Constant iterators
std::vector<int> numbers{ 11, 22, 33, 44, 55, 66, 77 };

fmt::print("numbers : ");
containers_iterators_03::print_collection(numbers);

std::vector<int>::iterator it = numbers.begin();
while( it != numbers.end()){
  *it = 100;
  ++it;
}

fmt::print("numbers : ");
containers_iterators_03::print_collection(numbers);

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

// std::vector<int>::const_iterator c_it = numbers.cbegin();
auto c_it = numbers.cbegin();
while (c_it != numbers.end()) {
  //*c_it = 100; // You can't go through the iterator and modify the container data
  ++c_it;  // But you can move the iterator (increment, decrement)
}
```

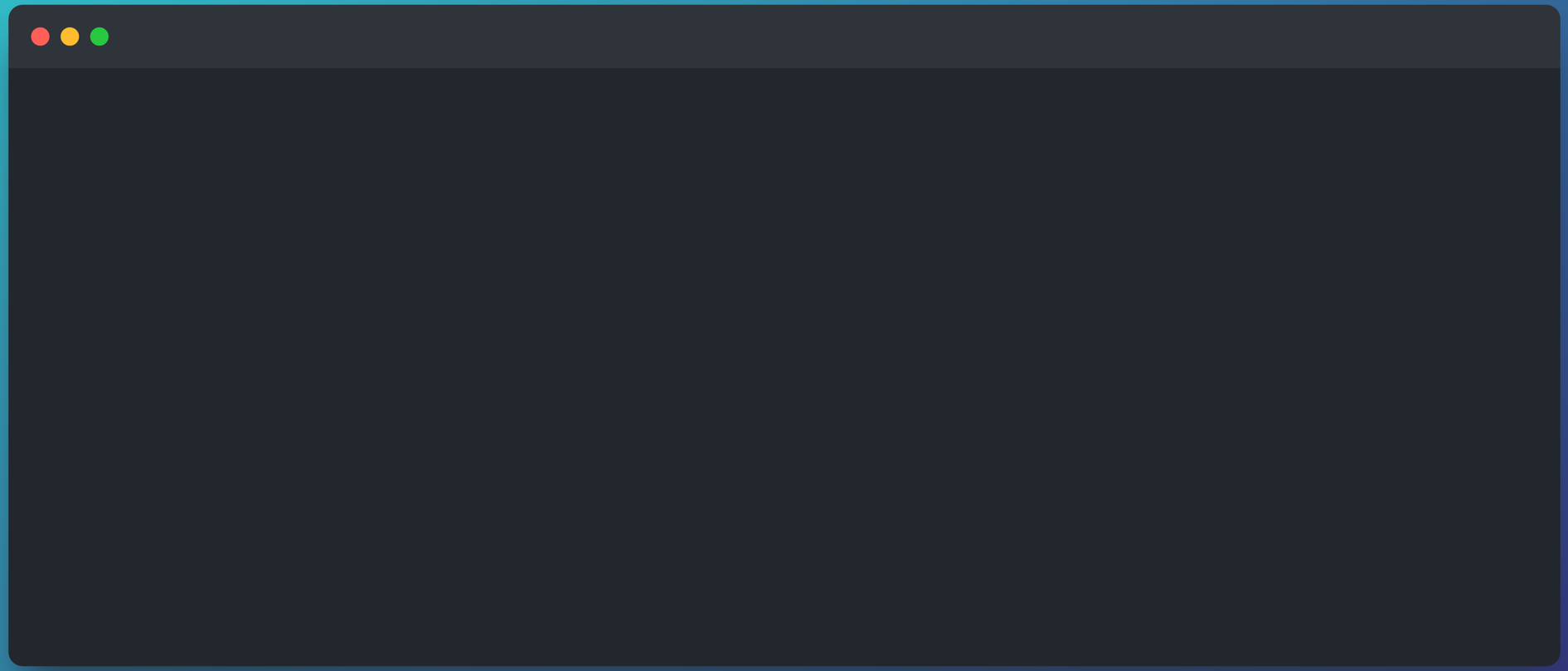
Constant reverse iterators

```cpp
// Constant reverse iterators
auto it1 = numbers.crbegin();
// std::vector<int>::const_reverse_iterator it1= numbers.crbegin();

while (it1 != numbers.crend()) {
  //*it1 = 600; // Compiler error, it1 is a const iterator, we can't modify container data through it.
  ++it1;
}
```

std::begin and std::end

```cpp
//std::begin and std::end
// std::vector<int> vi {1,2,3,4,5,6,7,8,9};
int vi[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
/*
fmt::print( " Collection : " ); // This will fail on raw arrays because they don't have begin and end functions
for(auto it = vi.begin(); it=≠ vi.end(); ++it){
  fmt::print( "{} ", *it );
}
*/
fmt::println("--------");

fmt::print(" Collection : ");  // This will work on raw arrays
for (auto it = std::begin(vi); it =≠ std::end(vi); ++it) {
  fmt::print("{} ", *it);
}
```

Demo time!



The iterator_traits template

```
/*
    .  The iterator traits template:
        . Problem:
            . Once we have an iterator, we want to know what type of data it points to.
            . But we don't know the type of the container.
        . Solution:
            . Use the iterator_traits template to get the type of the data the iterator points to.
*/
```

The iterator_traits template

```
export template <typename T>
void print_referenced_value(T it){

    //This syntax allows us to retrieve the value contained in the container, which is dynamic.
    //If the container contains int, val will be int, if the container contains std::string, val will be std::string, and so on.
    typename std::iterator_traits<T>::value_type val = *it;
    fmt::println("{}", val);
}

//We can use the technique to create a find function that works with any container: Pretty slick if you ask me.
export template <typename T>
auto custom_find(T begin, T end, typename std::iterator_traits<T>::value_type val){

    while(begin != end){
        if(*begin == val){
            return begin;
        }
        ++begin;
    }
    return end;
}
```

The iterator_traits template

```cpp
//Printing a single referenced value.
std::vector<int> numbers{ 11, 22, 33, 44, 55, 66, 77 };
//Works on vectors
containers_iterators_04::print_referenced_value(numbers.begin() + 2);

//works on arrays
std::array<int, 5> arr{ 1, 2, 3, 4, 5 };
containers_iterators_04::print_referenced_value(arr.begin() + 2);


//Custom find
auto result = containers_iterators_04::custom_find(numbers.begin(), numbers.end(), 33);
if (result != numbers.end()) {
    fmt::println("Found: {}", *result);
}
else {
    fmt::println("Not found");
}
```