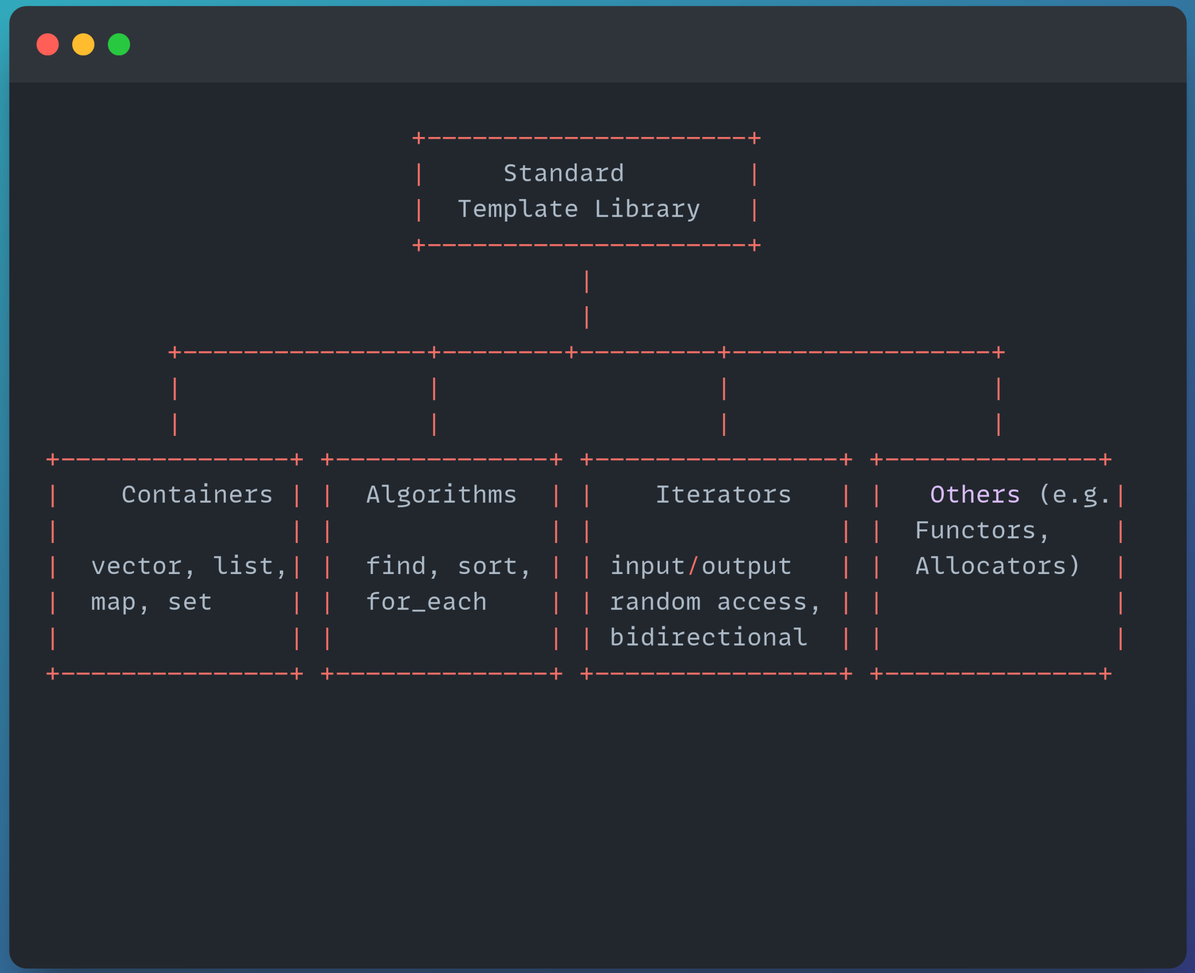
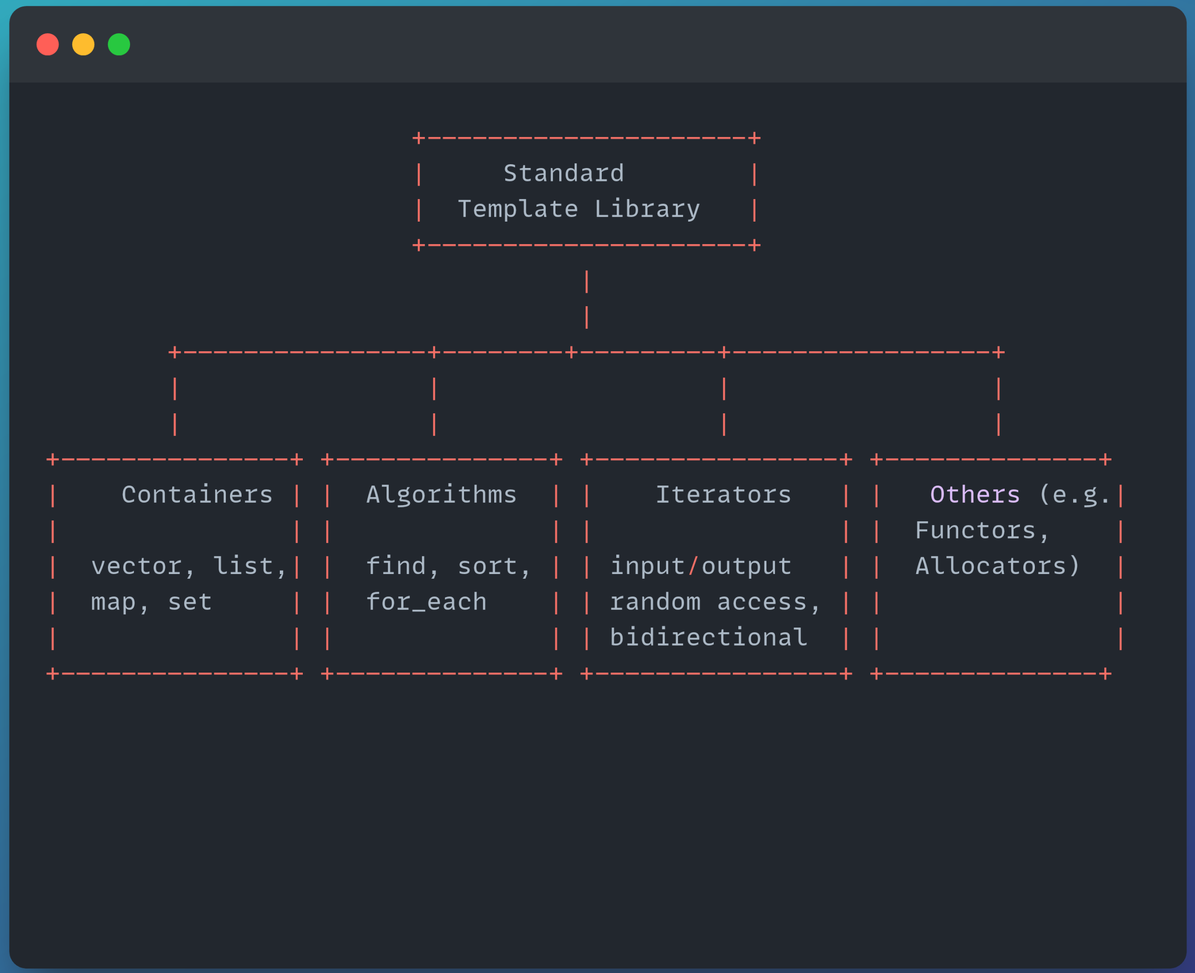
Ranges library

```
/*
    . Exploring the ranges library:

        .#1: Range algorithms

        .#2: Ranges library iterator pair algorithms

        .#3: Projections

        .#4: Views and range adaptors

        .#5: View composition and pipe operator

        .#6: Range factories
*/
```

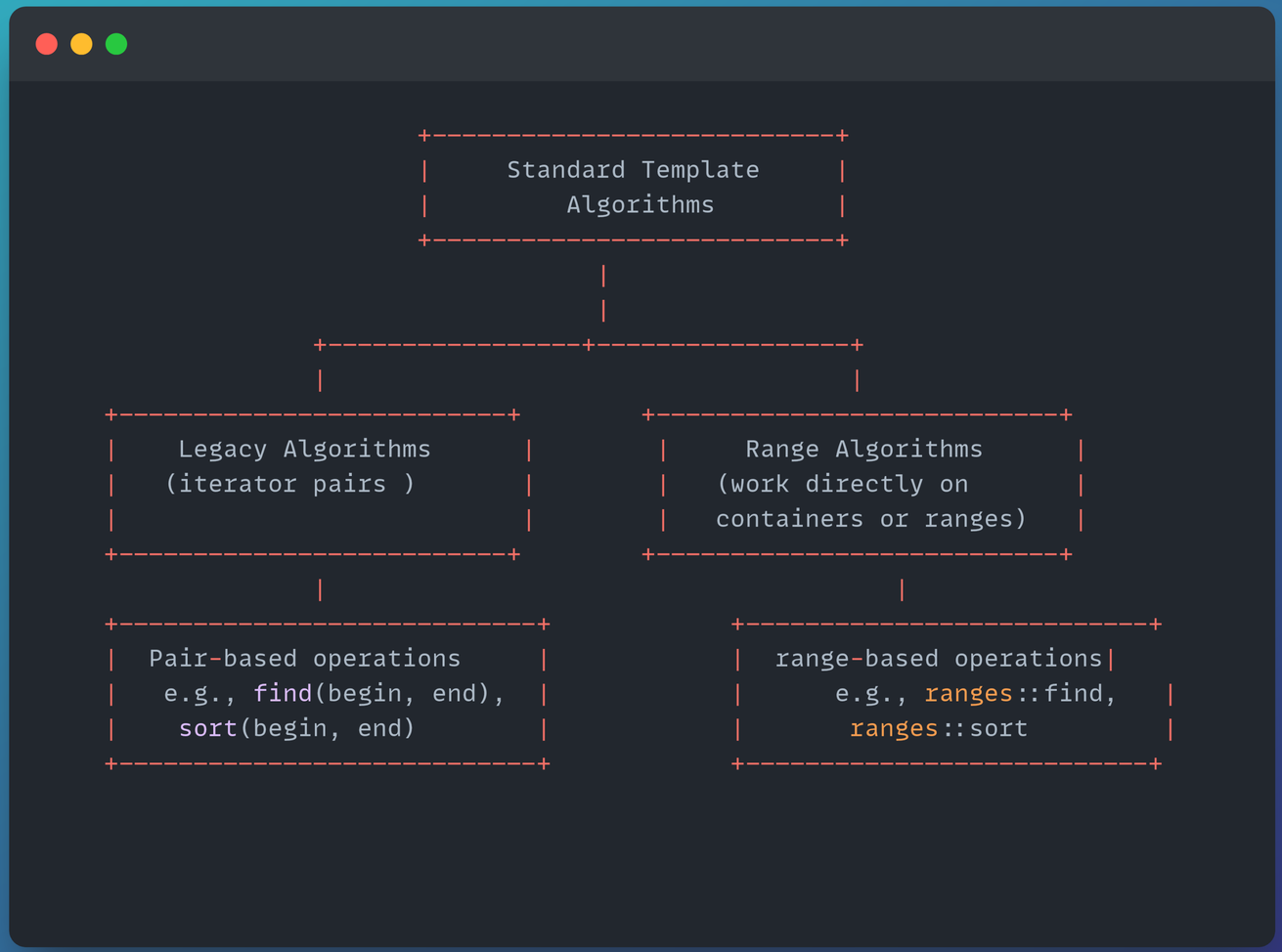
Range algorithms



Range algorithms



Range algorithms



std::all_of (iterator pair)

```cpp
std::vector<int> numbers{ 11, 2, 6, 4, 8, 3, 17, 9 };

auto odd = [](int n) { return n % 2 =/= 0; }; //The predicate

//Use ranges
auto result = std::all_of(numbers, odd); //Lives in the std::namesapce

if (result) {
      fmt::print("All elements in numbers are odd");
} else {
      fmt::print("Not all elements in numbers are odd");
}
```

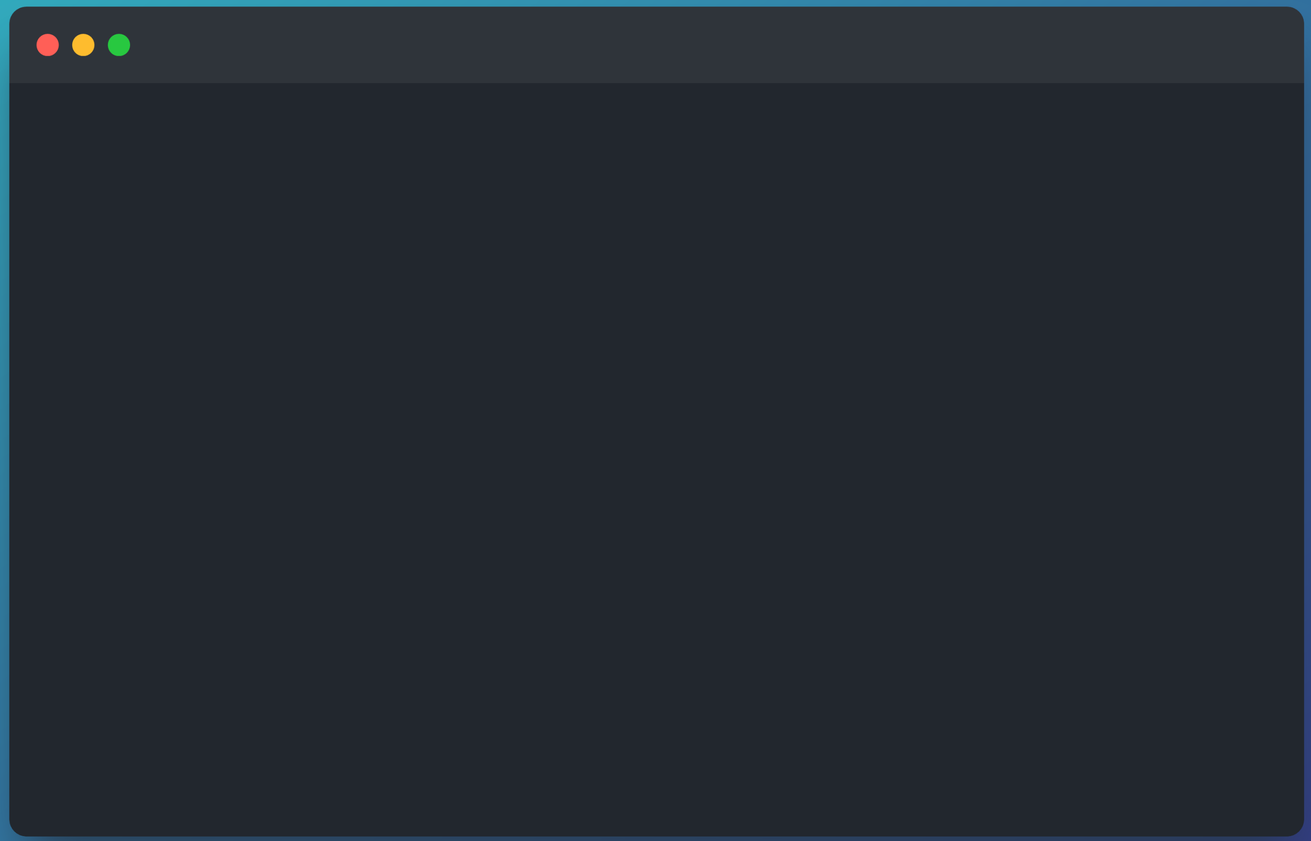
std::all_of (ranges)

```cpp
std::vector<int> numbers{ 11, 2, 6, 4, 8, 3, 17, 9 };

auto odd = [](int n) { return n % 2 == 0; }; //The predicate

//Use iterator pairs
auto result = std::ranges::all_of(numbers.begin(), numbers.end(), odd); //Lives in the std::ranges:: namespace

if (result) {
      fmt::print("All elements in numbers are odd");
} else {
      fmt::print("Not all elements in numbers are odd");
}
```

Other algorithms

```cpp
// For each
std::ranges::for_each(numbers, [](int &n) { n *= 2; });

// Sort
std::ranges::sort(numbers);

// Find
auto odd_n_position = std::ranges::find_if(numbers, odd);

if (odd_n_position != std::end(numbers)) {
  fmt::print("numbers contains at least one odd number: {}", *odd_n_position);
} else {
  fmt::print("numbers does not contain any odd number");
}

// Important, copying into outputstream on the fly
std::ranges::copy(numbers,std::ostream_iterator<int>(std::cout, " "));
```

Range algorithms: Demo time!



Ranges library: Iterator pair algorithms

```
/*
      . In the std::ranges namespace, we still have iterator pair algorithms
      . Why?
      . The std::ranges namespace provides constrained (With C++20 concepts) versions
        of the iterator pair alorithms in the std namespace
      . What you should do:
          . If you have to use iterator pair algorithms for some reason, prefer
            those in the std::ranges namespace
          . Otherwise, just use a range as we have seen in previous examples.
*/
```

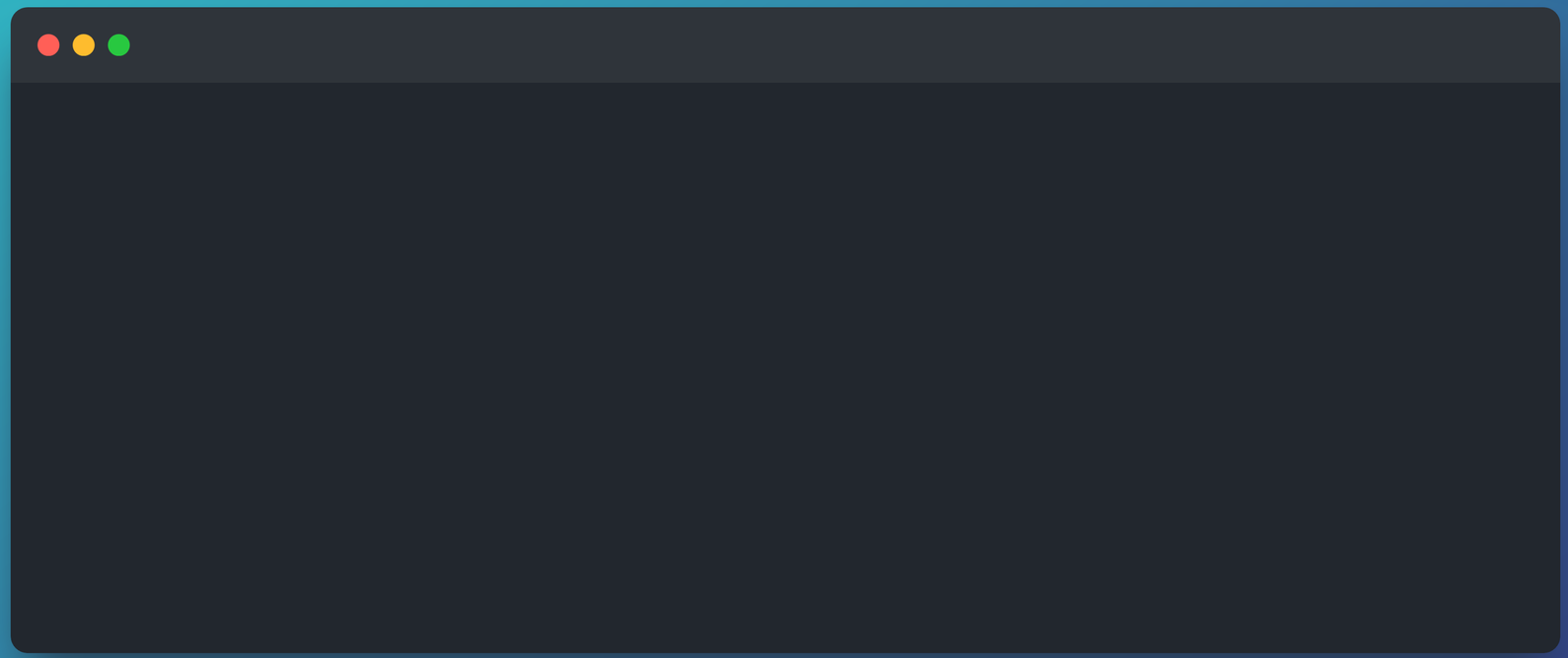
Ranges library: Iterator pair algorithms

```cpp
std::vector<int> numbers{ 11, 2, 6, 4, 8, 3, 17, 9 };

auto odd = [](int n) { return n % 2 =/= 0; };

// Ranges, iterator pair
auto result = std::ranges::all_of(numbers.begin(), numbers.end(), odd);

if (result) {
  fmt::println("All elements in numbers are odd");
} else {
  fmt::println("Not all elements in numbers are odd");
}

// For each
std::ranges::for_each(numbers.begin(), numbers.end(), [](int &n) { n *= 2; });

// Sort
std::ranges::sort(numbers.begin(), numbers.end());

// Find
auto odd_n_position = std::ranges::find_if(numbers.begin(), numbers.end(), odd);

if (odd_n_position =/= std::end(numbers)) {
  fmt::println("numbers contains at least one odd number : {}", *odd_n_position);
} else {
  fmt::println("numbers does not contain any odd number");
}
```

Ranges library: Iterator pair algorithms

```cpp
// Copy to the output stream
std::ranges::copy(numbers.begin(),numbers.end(),std::ostream_iterator<int>(std::cout, " "));


// Why you should prefer std::ranges algorithms from now on
fmt::println("");
std::vector<int> numbers_list{ 11, 2, 6, 4, 8, 3, 17, 9 };
fmt::print("numbers_list : ");
ranges_library_02::print_collection(numbers_list);

std::ranges::sort(numbers_list.begin(), numbers_list.end());
```

Ranges library: Iterator pair algorithms - Demo time!



Ranges library: Projections

```
/*
  Projections :
        . Usually the sorting is done based on operator< but you get one chance to write operator <
        . Sometimes you want to sort things based on another scheme ormember variable other than the one used by operator<
        . You can do that with projections.
        . For example, sorting based on y for Point can be achieved with a y projection as shown in the example
*/
```

Ranges library: Projections

```
//Points: compared based on distance from (0,0)
export class Point
{
    friend std::ostream& operator<< (std::ostream& out , const Point& p);
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    //Operators
    bool operator==(const Point& other) const;
    std::partial_ordering operator<=>(const Point& right) const;

private:
    double length() const;   // Function to calculate distance from the point(0,0)

public :
    double m_x{};
    double m_y{};
};
```

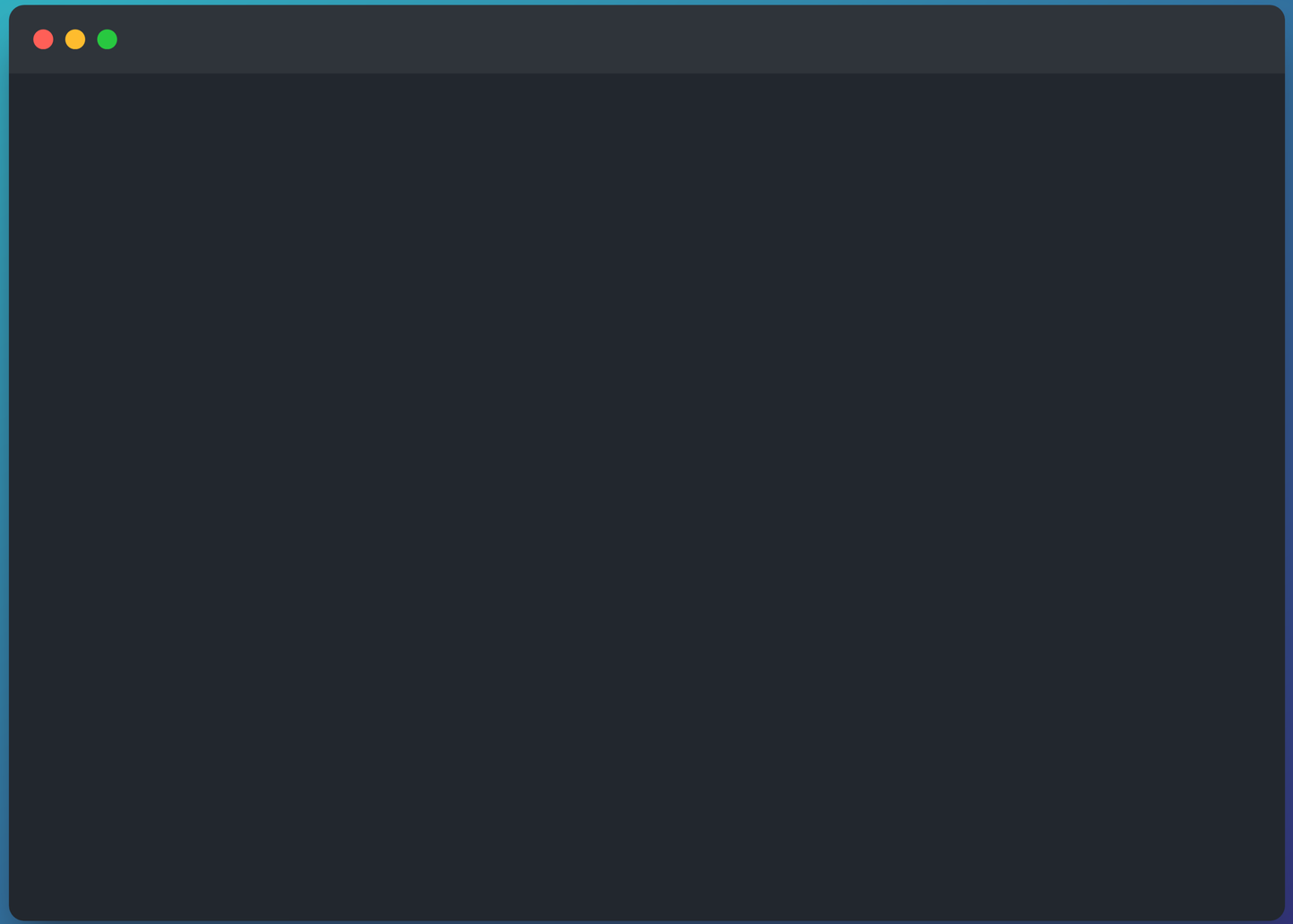
Ranges library: Projections

```cpp
//Points: compared based on distance from (0,0)
bool Point::operator==(const Point& other) const{
    return (this->length() == other.length());
}

std::partial_ordering Point::operator<=>(const Point& right) const{
    if(length() > right.length())
        return std::partial_ordering::greater;
    else if(length() == right.length())
        return std::partial_ordering::equivalent;
    else if(length() < right.length())
        return std::partial_ordering::less;
    else
        return std::partial_ordering::unordered;
}
```

Ranges library: Projections

```cpp
std::vector<ranges_library_03::Point> points { {10,90} ,{30,70}, {20,80} };

//Sorting with the default comparator
std::ranges::sort(points,std::less<>{}); // Default sort based on distance


//Sorting with a projection : The data is passed into the projection before
//it's passed into the comparator. std::less<> is going to compare two doubles
//instead of comparing two Points.
std::ranges::sort(points,std::less<>{},[](auto const & p){
    return p.m_x;
});

//Projecting with direct member variable
std::ranges::sort(points,std::less<>{},&ranges_library_03::Point::m_y);

//Projections with for_each
auto print = [](const auto& n) { std::cout << " " << n; };
using pair = std::pair<int, std::string>;
std::vector<pair> pairs{{1,"one"}, {2,"two"}, {3,"tree"}};

std::ranges::for_each(pairs, print, [](const pair& p) { return p.first; });
std::ranges::for_each(pairs, print, &pair::first);
```

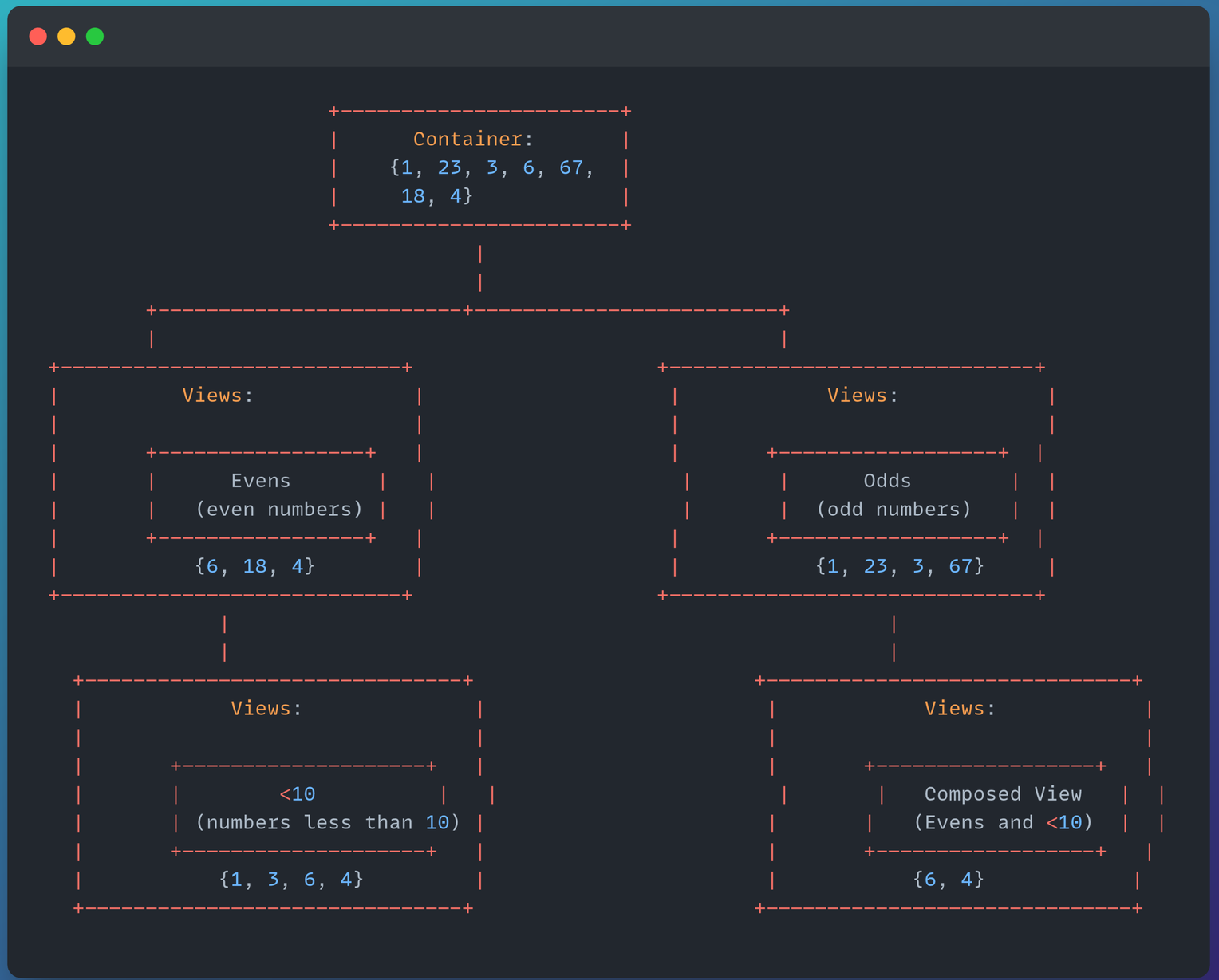
Ranges library: Projections - Demo time!



Ranges library: Views and view adaptors

```
/*
        . A view is an non-owning range
        . It's like a window we can set up to view some real data without
            setting up the infrastructure to store data
        . Views are cheap to copy and pass around as function parameters (by value),
            by design
        . Views set up lazy computations
            . Computation only happens when we iterate over the view
        . The difference between view and view adaptor:

            . Namespace Location:
                . std::ranges::take_while_view: This is a view type, which represents
                    the actual object that holds the data.
                . std::views::take_while: This is a view adaptor, which is used to create
                    a view by transforming an underlying range (like a container or another view).

                . Another way to look at this:
                    . std::ranges::take_while_view is the view object type.
                    . std::views::take_while is the function you use to adapt or create a view from an existing range.

            . View Types vs. View Adaptors:

                . A view type (e.g., std::ranges::take_while_view) represents the type of a view object and can be
                    instantiated directly (though typically not done manually).
                . A view adaptor (e.g., std::views::take_while) is a more convenient tool to transform or adapt a range,
                     making it easier to chain together operations.
*/
```

Ranges library: Views and view adaptors



Filtering view

```cpp
// std::ranges::filter_view: This is a view
std::vector<int> vi{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto evens = [](int i) { return (i % 2) == 0; };

std::ranges::filter_view v_evens = std::ranges::filter_view(vi, evens);//View created. No computation yet!

ranges_library_04::print(v_evens);// Computation happens in the print function
```

Filtering view adaptor

```cpp
//std::views::filter: This is a view adaptor
auto evens = [](int i) { return (i % 2) == 0; };
auto v_a = std::views::filter(vi, evens);
ranges_library_04::print(v_a);

//The names are different too:
  . std::ranges::filter_view      : A view
  . std::views::filter            : A view adaptor. [This is what you should prefer].
```

Transform view

```cpp
// std::ranges::transform_view
fmt::println("");
fmt::println("std::ranges::transform_view : ");
std::ranges::transform_view v_transformed = std::ranges::transform_view(vi, [](int i) { return i * 10; });
fmt::print("vi : ");
ranges_library_04::print(vi);
fmt::println("vi transformed : ");
ranges_library_04::print(v_transformed);
fmt::print("vi : ");
ranges_library_04::print(vi);
```

Take view

```
// std::ranges::take_view
fmt::println("");
fmt::println("std::ranges::take_view : ");
std::ranges::take_view v_taken = std::ranges::take_view(vi, 5);
fmt::print("vi : ");
ranges_library_04::print(vi);
fmt::println("vi taken : ");
ranges_library_04::print(v_taken);
```

Take_while view

```cpp
// std::ranges::take_while_view : takes elements as long as the predicate is met
fmt::println("");
fmt::println("std::views::take_while : ");
vi = { 1, 11, 23, 131, 2, 3, 4, 5, 6, 7, 8, 9 };
std::ranges::take_while_view v_taken_while = std::ranges::take_while_view(vi, [](int i) { return (i % 2) == 0; });
fmt::print("vi : ");
ranges_library_04::print(vi);
fmt::println("vi taken_while : ");
ranges_library_04::print(v_taken_while);
```

Drop view

```
// std::ranges::drop_view : drop n first elements
fmt::println("");
fmt::println("std::ranges::drop_view : ");
vi = { 1, 11, 23, 131, 2, 3, 4, 5, 6, 7, 8, 9 };
std::ranges::drop_view v_drop = std::ranges::drop_view(vi, 5);
fmt::print("vi : ");
ranges_library_04::print(vi);
fmt::println("vi_drop : ");
ranges_library_04::print(v_drop);
```

Drop_while view

```cpp
// std::views::drop_while_view : drops elements as long as the predicate is met
fmt::println("");
fmt::println("std::ranges::drop_while_view : ");
vi = { 1, 11, 23, 4, 2, 3, 4, 5, 6, 7, 8, 9 };
std::ranges::drop_while_view v_drop_while = std::ranges::drop_while_view(vi, [](int i) { return (i % 2) == 0; });
fmt::print("vi : ");
ranges_library_04::print(vi);
fmt::println("v_drop_while : ");
ranges_library_04::print(v_drop_while);
```

Keys_view and Values_view

```cpp
using pair = std::pair<int, std::string>;
std::vector<pair> numbers{ { 1, "one" }, { 2, "two" }, { 3, "tree" } };

// Compiler error when you build views explicitly. Don't understand why yet
//auto k_view = std::ranges::keys_view(numbers);
//auto v_view = std::ranges::values_view(numbers);

// But we can use view adaptors
auto k_view = std::views::keys(numbers);
auto v_view = std::views::values(numbers);
ranges_library_04::print(k_view);
ranges_library_04::print(v_view);
```

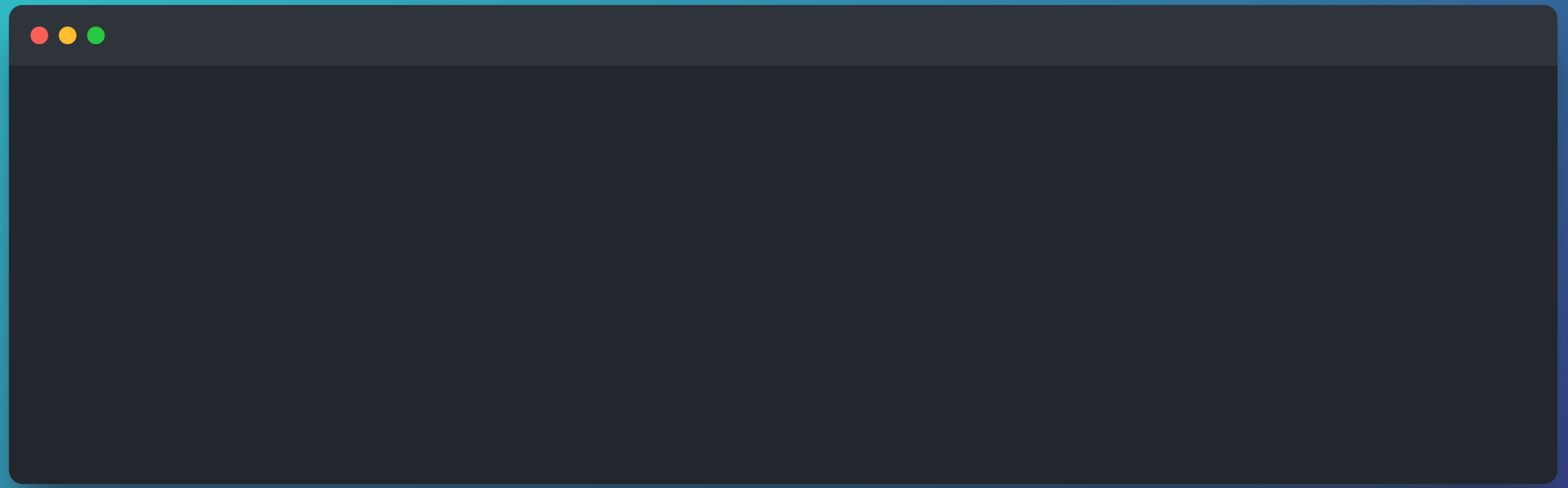
Student example

```cpp
export struct Student{
    friend std::ostream& operator<<(std::ostream& out, const Student& s){
        out << "Student [ name : " << s.m_name << ", age : " << s.m_age << "]";
        return out;
    }
    auto operator <=>(const Student& s) const= default;
    std::string m_name;
    unsigned int m_age;
};
```

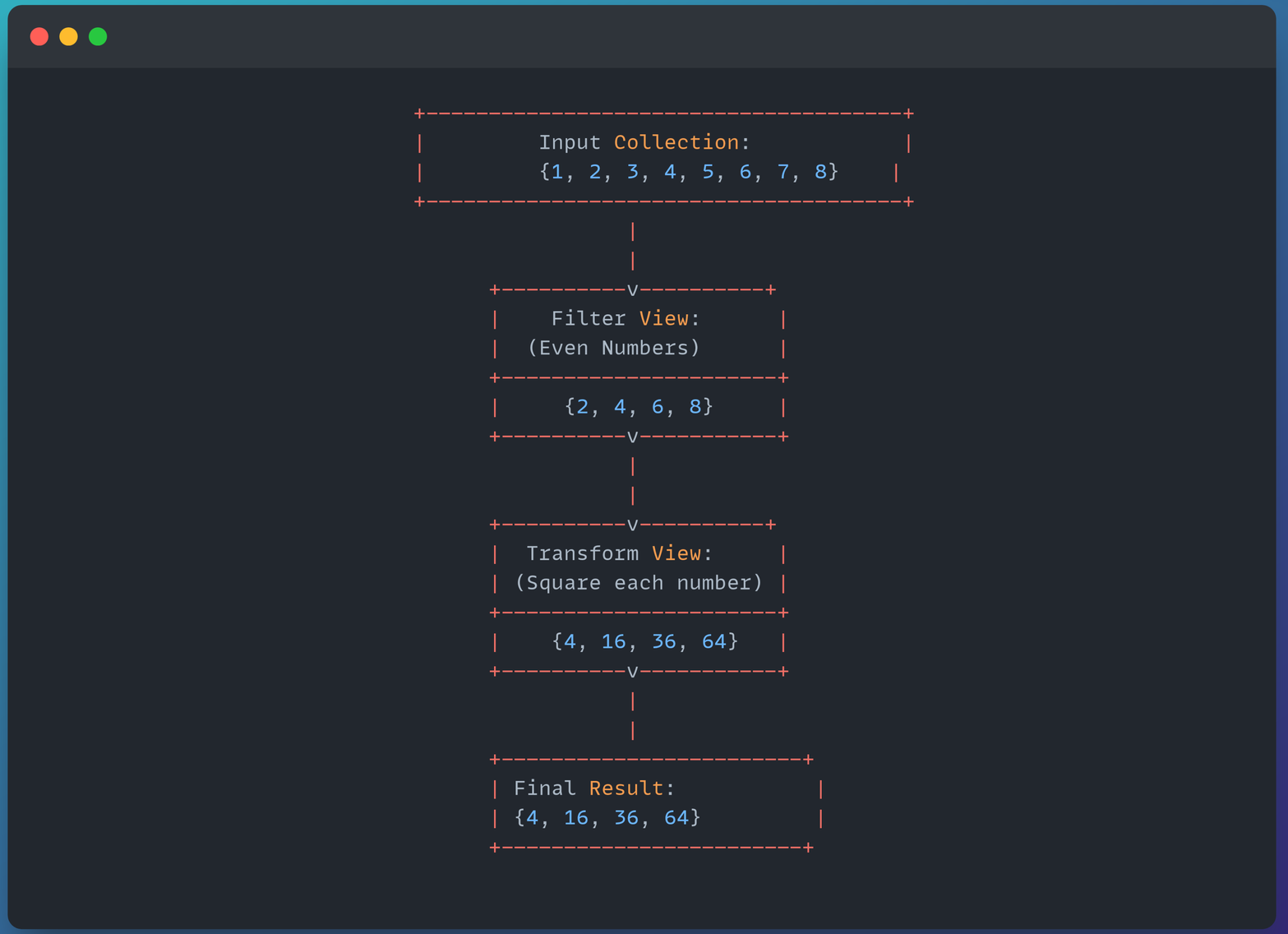
Student example

```
//Set up a classroom of students
std::vector<ranges_library_04::Student> class_room{ { "Mike", 12 }, { "John", 17 }, { "Drake", 14 }, { "Mary", 16 } };

// Sort the students by age in ascending order
std::ranges::sort(class_room, std::less<>{}, &ranges_library_04::Student::m_age);

//Only take those whose age is less than 15: Notice that we're using a view adaptor, like we should.
fmt::println("students under 15 : ");
ranges_library_04::print(std::views::take_while(class_room, [](const ranges_library_04::Student &s) {
    return (s.m_age < 15);
}));
```

Views and view adaptors: Demo time!



View composition and the pipe operator



View composition and the pipe operator

```cpp
std::vector<int> vi{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Filter out evens and square them out.
fmt::print("vi : ");
ranges_library_05::print(vi);
// V1 : Raw function composition
auto even = [](int n) { return n % 2 == 0; };
auto my_view = std::views::transform(std::views::filter(vi, even), [](auto n) { return n *= n; });
fmt::print("vi transformed : ");
ranges_library_05::print(my_view);

fmt::println("");
fmt::println("Pipe operator : ");
auto my_view1 = vi | std::views::filter(even) | std::views::transform([](auto n) { return n *= n; });
fmt::print("vi transformed : ");
ranges_library_05::print(my_view1);
```

View composition and the pipe operator: Classroom example

```cpp
// classroom done as map : Keys are sorted by default
// std::unordered_map<std::string,unsigned int> classroom    {
std::map<std::string, unsigned int> classroom{
  { "John", 11 }, { "Mary", 17 }, { "Steve", 15 }, { "Lucy", 14 }, { "Ariel", 12 }
};

// Print out the names
// auto names_view = std::views::keys(classroom);
auto names_view = classroom | std::views::keys;
fmt::println("names : ");
std::ranges::copy(names_view, std::ostream_iterator<std::string>(std::cout, " "));

// Print out the ages :
auto ages_view = std::views::values(classroom);
std::ranges::copy(ages_view,std::ostream_iterator<unsigned int>(std::cout," "));


// Print names in reverse : this doesn't work if you store the data in anunordered_map. The reason is that unoredered_map
// doesn't have reverse iterators that are needed to set up a reverse view.
std::ranges::copy(std::views::keys(classroom) | std::views::reverse ,
             std::ostream_iterator<std::string>(std::cout, " "));

// Pick names that come before the letter "M" in the alphabet
auto before_M = [](const std::string &name) {
  return (static_cast<unsigned char>(name[0]) < static_cast<unsigned char>('M'));
};

std::ranges::copy(classroom | std::views::keys | std::views::filter(before_M),
      std::ostream_iterator<std::string>(std::cout, " "));
```

Range factories

```
/*
  . A range factory creates views from scratch, rather than adapting an existing range.
    These factories generate new ranges rather than modifying existing ones.
  . Example of range factories include std::views::iota.

  . View vs View adaptor vs Range factory:
    . View:
      . A view is an object that represents a non-owning, lazily evaluated transformation of a range.
      . Example: std::ranges::take_while_view.
      . Can be directly created but typically created using adaptors or range factories.
    . View Adaptor:
      . A view adaptor is a function-like entity used to create or adapt a view from an existing range.
      . Example: std::views::take_while.
      . Cleaner and more convenient to use, supports chaining operations on ranges.
    . Range Factory:
      . A range factory generates a range (or view) from scratch, without needing an underlying range.
      . Example: std::views::iota.
      . Used to produce ranges that do not directly rely on another range (e.g., infinite ranges, ranges of numbers, etc.).

*/
```

Range factories

```cpp
//Range factories
// Generate an infinite sequence of numbers
// auto infinite_view = std::views::iota(1) | std::views::take(20); // Stores the compuation
// auto data_view = std::views::take( std::views::iota(1) , 20);

// Numbers are generated lazily, on the fly, as we need them in each iteration
for (auto i : std::views::iota(1) | std::views::take(20)) {
  fmt::println("{}", i);
}
```

Some other views adaptors introduced in C++23

```cpp
//std::views::zip (C++23)
std::vector<int> v1 = {1, 2, 3};
std::vector<std::string> v2 = {"one", "two", "three"};

auto zipped = std::views::zip(v1, v2); // The result is a view where each element is a
                                       //tuple of the corresponding elements from the input ranges


fmt::print("std::views::zip:\n");
for (const auto& [i, s] : zipped) {
    fmt::print("({}, {}) ", i, s);
}
```

Some other views adaptors introduced in C++23

```cpp
//std::views::adjacent (C++23)
auto adjacent = std::views::adjacent<2>(v1); // The result is a view where each element is a pair
                                             // of adjacent elements from the input range

fmt::print("std::views::adjacent:\n");
for (const auto& [a, b] : adjacent) {
    fmt::print("({}, {}) ", a, b);
}
fmt::print("\n\n");
```

Some other views adaptors introduced in C++23

```cpp
// std::views::chunk (C++23)
std::vector<int> v1 {1, 2, 3, 4, 5, 6, 7, 8};
auto chunked = std::views::chunk(v1, 2); // The result is a view where each element is a chunk of
                                         // the input range with the specified size
// Iterate over the chunked view and print each subrange
for (const auto& chunk : chunked) {
    fmt::print("Chunk: ");
    for (int n : chunk) {
        fmt::print("{} ", n);
    }
    fmt::println(""); // Prints a new line after each chunk
}
fmt::print("\n\n");
```

Some other views adaptors introduced in C++23

```cpp
// std::views::stride (C++23)
std::vector<int> v1 {1, 2, 3, 4, 5, 6, 7, 8};
auto strided = std::views::stride(v1, 2);    // The result is a view where each element
                                             // is every nth element from the input range

fmt::print("std::views::stride:\n");
for (int n : strided) {
    fmt::print("{} ", n);
}
fmt::print("\n\n");
```

Some other views adaptors introduced in C++23

```cpp
// std::views::cartesian_product (C++23)
std::vector<int> v1 = {1, 2, 3};
std::vector<char> v3 = {'a', 'b', 'c'};
auto cartesian_product = std::views::cartesian_product(v1, v3); // The result is a view where each element is a pair
        // of elements from the input ranges: [(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3, 'a'), (3, 'b'), (3, 'c')]

fmt::print("std::views::cartesian_product:\n");
for (const auto& [i, c] : cartesian_product) {
    fmt::print("({}, {}) ", i, c);
}
fmt::print("\n");
```