C++20 Concepts

```
/*
  . Concepts:
      .#1: Basic concepts

      .#2: Build your own concepts

      .#3: Zooming in on the requires clause

      .#4: Combining concepts

      .#5: Concepts and auto

*/
```

C++20 Concepts: Basics and syntax

```cpp
// Syntax1
export template <typename T>
requires std::integral<T>
T add(T a, T b) {
    return a + b;
}

// Syntax2
export template <std::integral T>
T add(T a, T b) {
    return a + b;
}

// Syntax3
export auto add(std::integral auto a, std::integral auto b) {
    return a + b;
}

// Syntax4
export template <typename T>
T add(T a, T b) requires std::integral<T> {
    return a + b;
}
```

C++20 Concepts: Basics and syntax

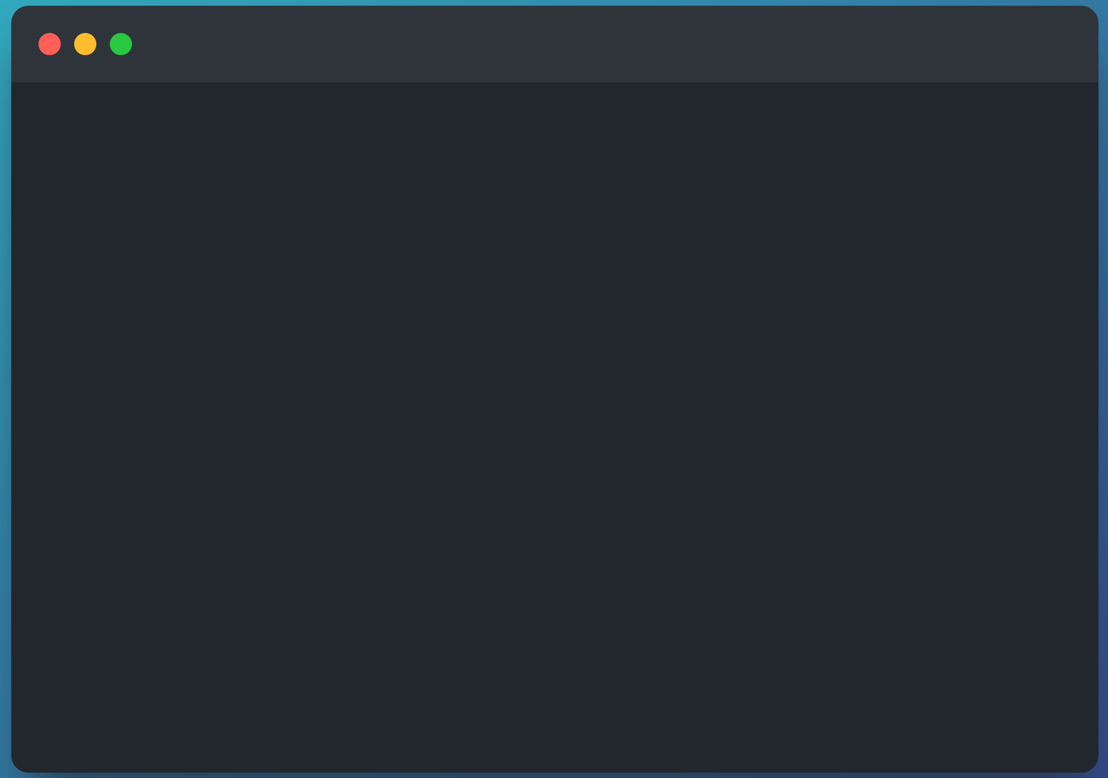
```cpp
char a_0{ 10 };
char a_1{ 20 };

auto result_a = concepts_01::add(a_0, a_1);
fmt::println("result_a : {}", static_cast<int>(result_a));

int b_0{ 11 };
int b_1{ 5 };
auto result_b = concepts_01::add(b_0, b_1);
fmt::println("result_b : {}", result_b);


double c_0{ 11.1 };
double c_1{ 1.9 };
auto result_c = concepts_01::add(c_0, c_1);
fmt::println("result_c : {}", result_c);
```

C++20 Concepts: Basics and syntax - Demo time!



C++20 Concepts: Build your own.

```cpp
// Syntax1
/*
export template <typename T>
concept MyIntegral = std::is_integral_v<T>;

MyIntegral auto add(MyIntegral auto a, MyIntegral auto b) {
    return a + b;
}
*/

export template<typename T>
concept Multipliable = requires(T a, T b) {
    a * b; // Just makes sure the syntax is valid
};

export template<typename T>
concept Incrementable = requires(T a) {
    a += 1;
    ++a;
    a++;
};

export template<typename T>
requires Incrementable<T>
T add(T a, T b) {
    return a + b;
}
```

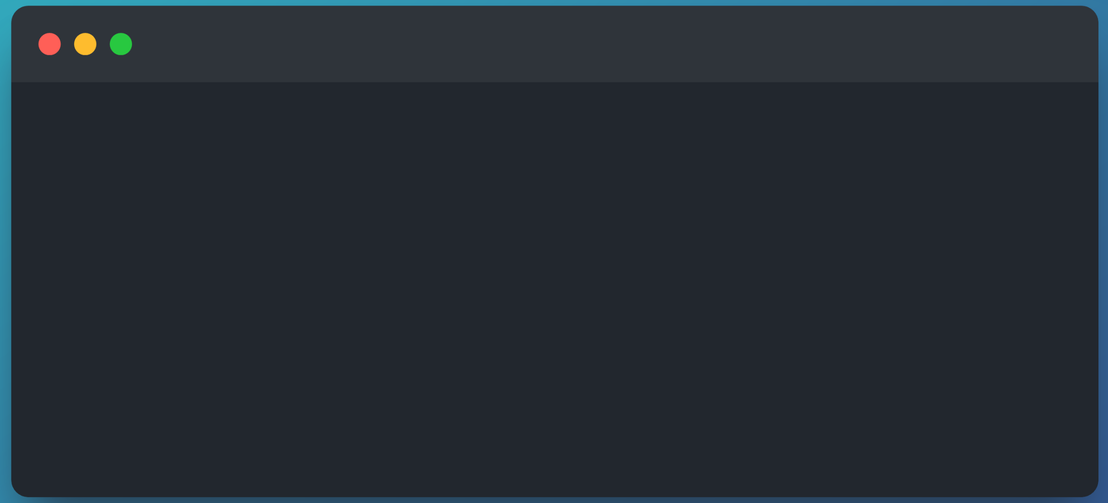
C++20 Concepts: Build your own.

```
double x{ 6 };
double y{ 7 };

//std::string x{"Hello"};
//std::string y{"World"};

concepts_02::add(x, y);
```

C++20 Concepts: Build your own - Demo time!



C++20 Concepts: The requires clause

```cpp
export template<typename T>
concept TinyType = requires(T t) {
    sizeof(T) ⩽ 4; // Simple requirement : Only enforces syntax
    requires sizeof(T) ⩽ 4; // Nested requirements
};

// Compound requirement
export template<typename T>
concept Addable = requires(T a, T b) {
    // noexcept is optional
    {
        a + b
    } → std::convertible_to<int>; // Compound requirement
    // Checks if a + b is valid syntax, doesn't throw exceptions(optional), and the result
    // is convertible to int(optional)
};

export Addable auto add(Addable auto a, Addable auto b) {
    return a + b;
}
```

C++20 Concepts: The requires clause

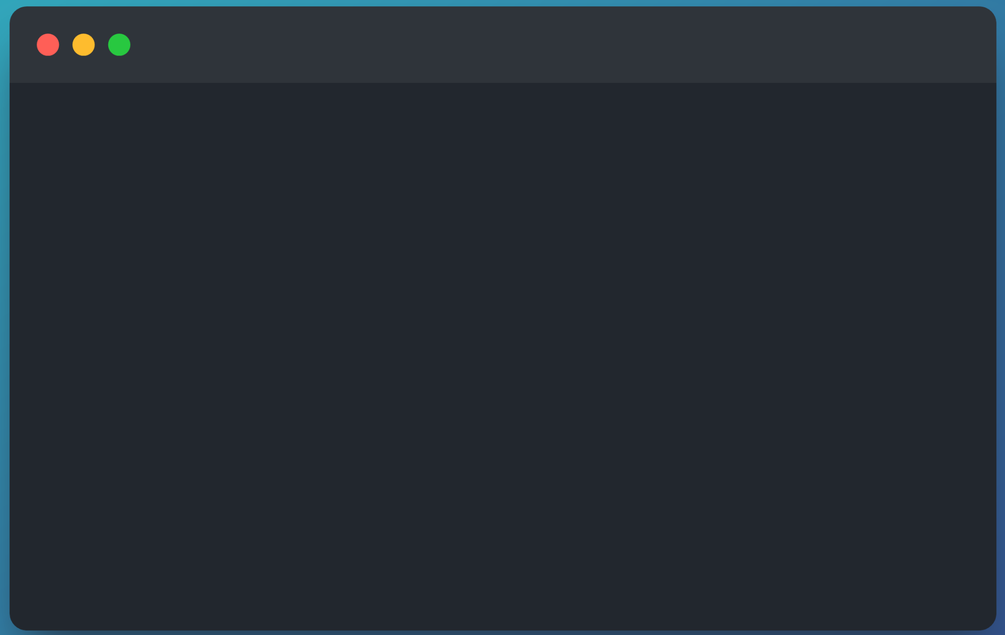
```
double x{ 67 };
double y{ 56 };

// std::string x{"Hello"};
// std::string y{"World"};

// auto s = x + y;

auto result = concepts_03::add(x, y);
fmt::println("result : {}", result);
fmt::println("sizeof(result) : {}", sizeof(result));
```

C++20 Concepts: The requires clause - Demo time!

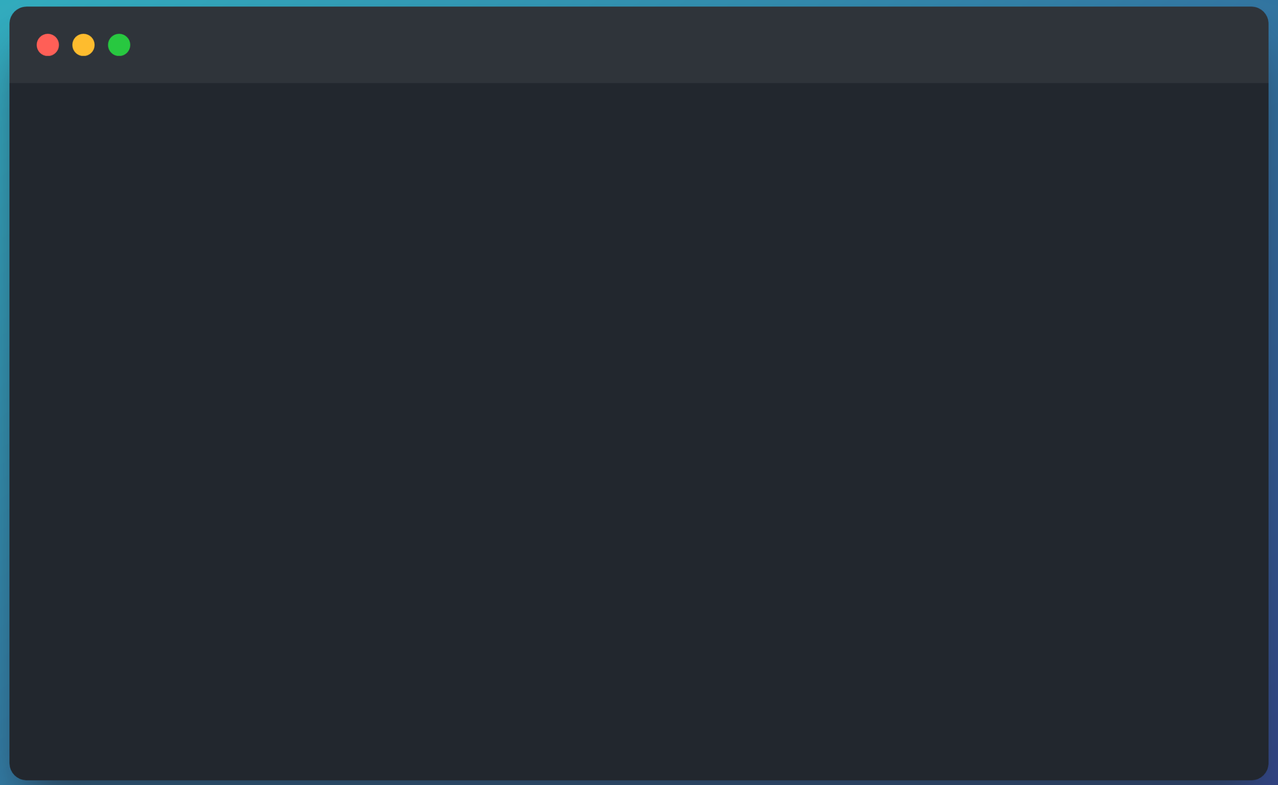


C++20 Concepts: Combining concepts

```cpp
export template<typename T>
concept TinyType = requires(T t) {
    sizeof(T) ⩽ 4; // Simple requirement
    requires sizeof(T) ⩽ 4; // Nested requirement
};

export template<typename T>
//requires std::integral<T> || std::floating_point<T> // OR operator
//requires std::integral<T> && TinyType<T>
requires std::integral<T> && requires ( T t){
    sizeof(T) ⩽4; // Simple requirement
    requires sizeof(T) ⩽ 4; // Nested requirement
}
T add(T a, T b){
    return a + b;
}
```

C++20 Concepts: Combining concepts - Demo time!



C++20 Concepts: Concepts and auto.

```cpp
// This syntax constrains the auto parameters you pass in
// to comply with the std::integral concept
export std::integral auto add(std::integral auto a, std::integral auto b) {
    return a + b;
}
```