Generic programming with templates

```
/*
    . Template basics in C++
        .#1: Basic function template
        .#2: Class template basics
        .#3: Template member functions
        .#4: Template parameter types:
            . type template parameters
            . non-type template parameters
        .#5: Template template parameters
*/
```

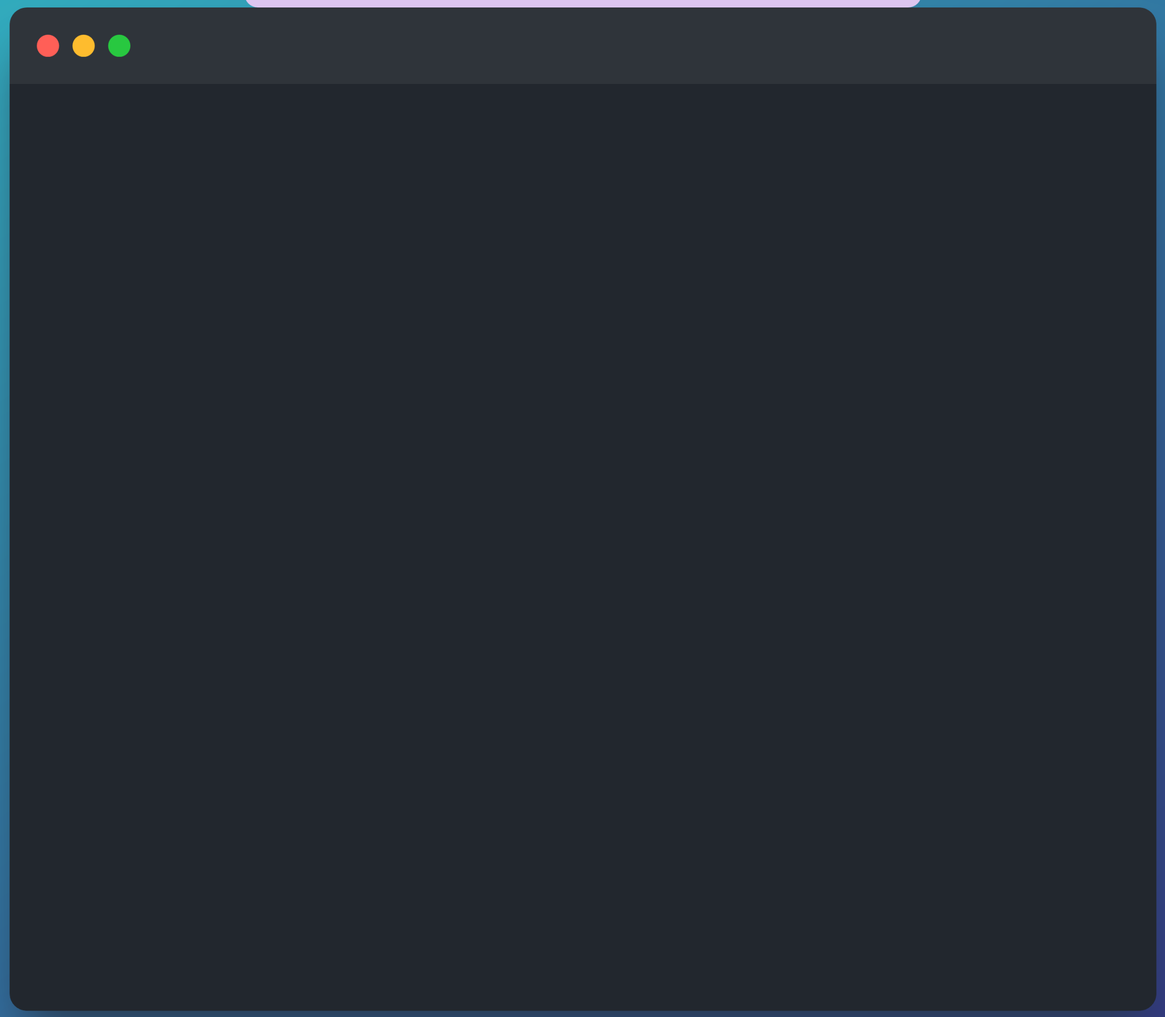
Function templates

```
//A function that can add up any similar types
export template <typename T>
T add(T a, T b) {
    return a + b;
}
//The compiler generates any instance you need.
```

Function templates

```
//A function that can add up any similar types
auto value =add(1,2);
fmt::print("The value is: {}\n", value);

Integer a(1);
Integer b(2);
auto sum = a + b;
fmt::print("The sum is: {}\n", sum.get());

auto result =add(11.1, 22.2);
fmt::print("The result is: {}\n", result);

std::string lastname = "Doe";
std::string firstname = "John";
auto fullname =add(firstname, lastname);
fmt::print("The full name is: {}\n", fullname);


Point p1(1.1, 2.2);
Point p2(3.3, 4.4);
auto p3 = p1 + p2;
fmt::print("The point is: ({}, {})\n", p3.get_x(), p3.get_y());
```

Demo time!



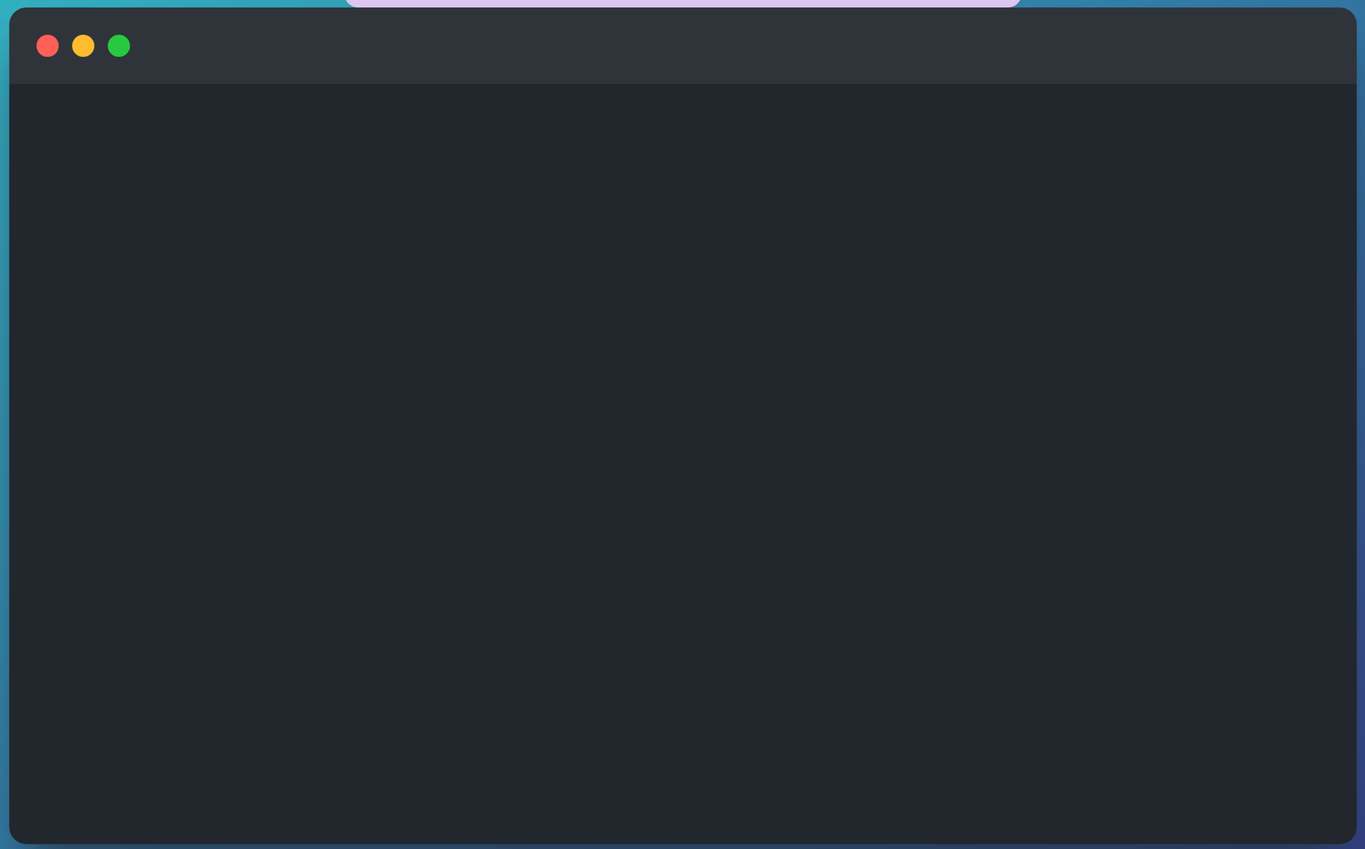
Class templates

```
//The requirement is that the type T must support the + operator.
//The types of the coordinates also must be the same.
//The hard work happens at compile time
export template <typename T>
class Point{
    friend Point operator+ (const Point a,const Point b){
        return Point(a.x + b.x, a.y + b.y);
    }
public:
    Point(T x, T y): x(x), y(y) {}
    T get_x() const { return x;}
    T get_y() const { return y;}
private:
    T x;
    T y;
};
```

Class templates

```cpp
//Points made up of int
Point<int> p1(1, 2);
Point<int> p2(3, 4);
auto p3 = p1 + p2;
fmt::print("The point is: ({}, {})\n", p3.get_x(), p3.get_y());


//Points made up of double
Point<double> p4(1.1, 2.2);
Point<double> p5(3.3, 4.4);
auto p6 = p4 + p5;
fmt::print("The point is: ({}, {})\n", p6.get_x(), p6.get_y());

//Custom integers
Point<Integer> p7(Integer(10), Integer(20));
Point<Integer> p8(Integer(30), Integer(40));
auto p9 = p7 + p8;
fmt::print("The point is: ({}, {})\n", p9.get_x().get(), p9.get_y().get());
```

Demo time!



Template parameter types:

```cpp
//Type template parameters
export template <typename T>
class Point {
    T x, y;
public:
    Point(T x, T y) : x(x), y(y) {}
    T get_x() const { return x; }
    T get_y() const { return y; }
    Point operator+(const Point& other) {
        return Point(x + other.x, y + other.y);
    }
};
```

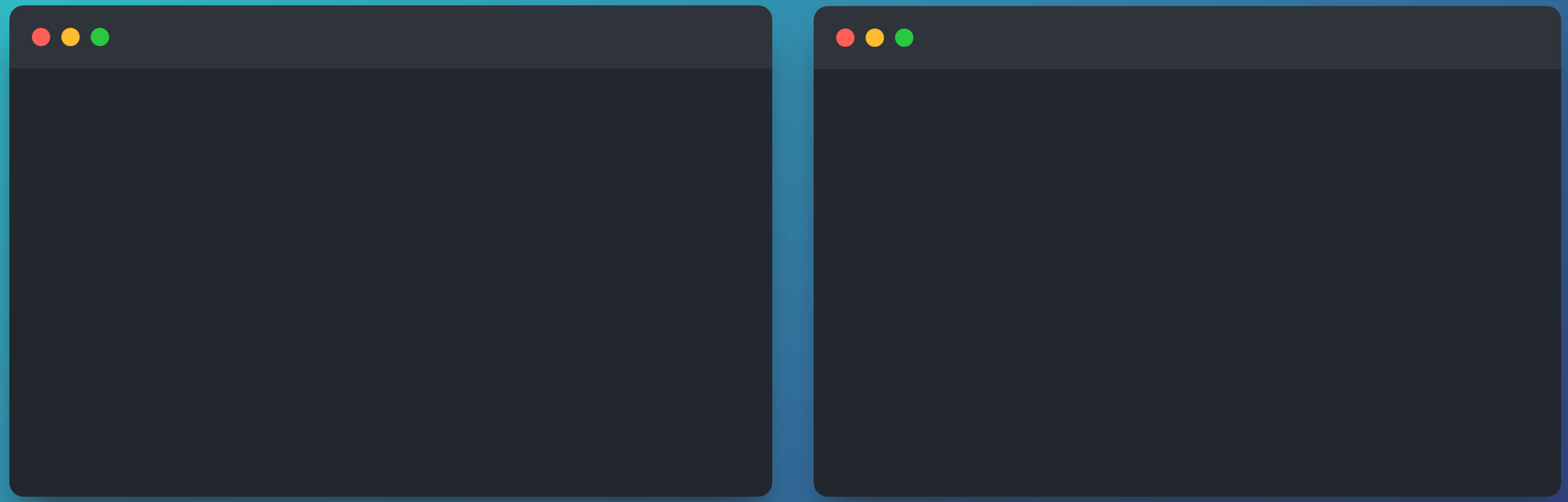
```cpp
//Non-type template parameters
export template <int N>
class Array{
private:
    int m_array[N];
public:
    int get_size() const{
        return N;
    }
};
```

Template parameter types:

```cpp
//Type template parameters
export template <typename T>
class Point {
    T x, y;
public:
    Point(T x, T y) : x(x), y(y) {}
    T get_x() const { return x; }
    T get_y() const { return y; }
    Point operator+(const Point& other) {
        return Point(x + other.x, y + other.y);
    }
};


//Usage
//class template
Point<int> p1(1, 2);
Point<int> p2(3, 4);
auto p3 = p1 + p2;
fmt::print("The point is: ({}, {})\n",
                  p3.get_x(), p3.get_y());

//function template
auto result1 = maximum(11, 2);
fmt::print("The result is: {}\n", result1);

auto result2 = add(11, 2);
fmt::print("The result is: {}\n", result2);
```

```cpp
//Non-type template parameters
export template <int N>
class Array{
private:
    int m_array[N];
public:
    int get_size() const{
        return N;
    }
};


//Usage
    //Non-type template parameters
Array<10> arr1;
fmt::print("The size is: {}\n", arr1.get_size());
```

Demo time!



Template template parameters

```
//The container template.
export template <typename T>
class Container {
public:
    void add(const T& element) {
        data.push_back(element);
    }

    void print() const {
        for (const auto& element : data) {
            fmt::print("{} ", element);
        }
        fmt::print("\n");
    }

private:
    std::vector<T> data;
};
```

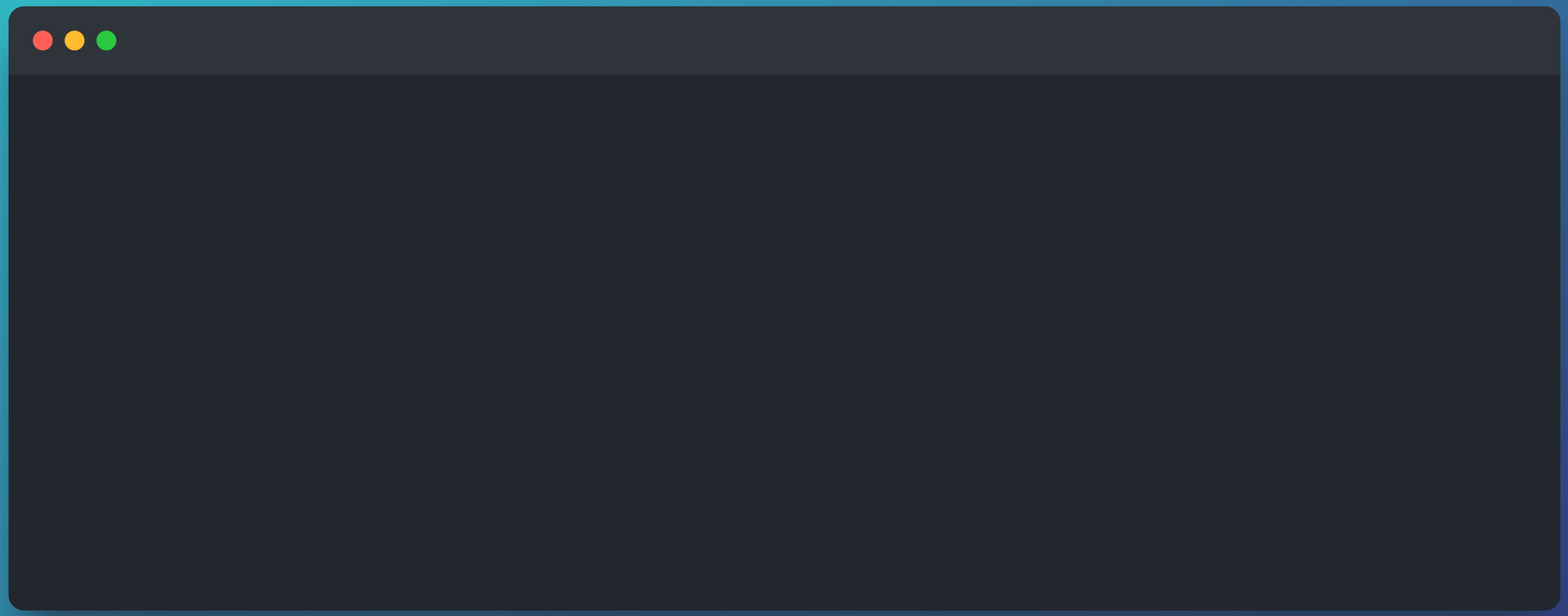
Template template parameters

```
//The class accepts another template class as a parameter, which in this case is ContainerType.
export template <template <typename> class ContainerType, typename T>
class Processor {
public:
    void process(ContainerType<T>& container, const T& element) {
        container.add(element);
        container.print();
    }
};
```

Template template parameters

```
  //Usage
Container<int> container1;
Processor<Container, int> processor1;
processor1.process(container1, 1);
processor1.process(container1, 2);
processor1.process(container1, 3);

//We can easily change the container that Processor works with, without having to change the class.
//This is powered by template template parameters.
Container<std::string> container2;
Processor<Container, std::string> processor2; // A class template instance passed as a template argument.
processor2.process(container2, "Hello");
processor2.process(container2, "World");
```

Demo time!



Template member functions: The options

```
/*
    . Exploring different arrangements for classes and member functions when templates are involved:
        . The class is a template, member uses the template parameter
        . The class is not a template, but the member functions are templates.
        . Both the class and the member functions are templates.

*/
```

Template member functions: The options

```
// Class is a template, member uses the template parameter
export template <typename T>
class Calculator_1 {
public:
    Calculator_1() = default;

    T add(T a, T b) { return a + b; }
    T subtract(T a, T b) { return a - b; }
    T multiply(T a, T b) { return a * b; }
    T divide(T a, T b) { return a / b; }
};
```

Template member functions: The options

```
// Class is not a template, member is a function template
export class Calculator_2 {
public:
    Calculator_2() = default;

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

    template <typename T>
    T subtract(T a, T b) { return a - b; }

    template <typename T>
    T multiply(T a, T b) { return a * b; }

    template <typename T>
    T divide(T a, T b) { return a / b; }
};
```

Template member functions: The options

```
// Class is not a template, member is a function template
export class Calculator_2 {
public:
    Calculator_2() = default;

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

    template <typename T>
    T subtract(T a, T b) { return a - b; }

    template <typename T>
    T multiply(T a, T b) { return a * b; }

    template <typename T>
    T divide(T a, T b) { return a / b; }
};
```

Template member functions: The options

```
//Both class and member are templates
// Template class Box that can hold any type of item
export template <typename T>
class Box {
private:
    T item;
public:
    // Constructor to initialize the item
    Box(T item) : item(item) {}

    // Method to get the item
    T getItem() const {
        return item;
    }

    // Template member function to compare the item with another item of a different type
    template <typename U>
    bool compare(const U& other) const {
        return item == other; // There may be assumptions made on the U type here, but we'll just use ==.
    }
};
```

Demo time!