# Templates and friendship

```
/*
    . Templates and friendship:

      .#1: Friendship without templates
          . Exploring the basics of friendship
          . This has been covered earlier in the course.

      .#2: Class is not template, friend function and friend class are templates.

      .#3:  Class is not a template, but the friend function and class are specific specializations of the template.
          . Only a few friend functions and templates have access to the private members of the class.

      .#4: Class is a template, and the friend function is not a template.

      .#5: The class is a template, friend functions and classes are templates, but
            but we want a few instances of the friends to have access to the private members of the class.
          . Only int instances of the friends have access to the private members of the class.
          . Others can't. They will trigger a compiler error.

      .#6: Class is template, friends are template, and we want any instance of the friend templates to have
              access to private members of the class.

      .#7: Class is a template, friends are templates, and we only want friends whose template paramter matches that
          of the class to have access to the private members of the class.

*/
```

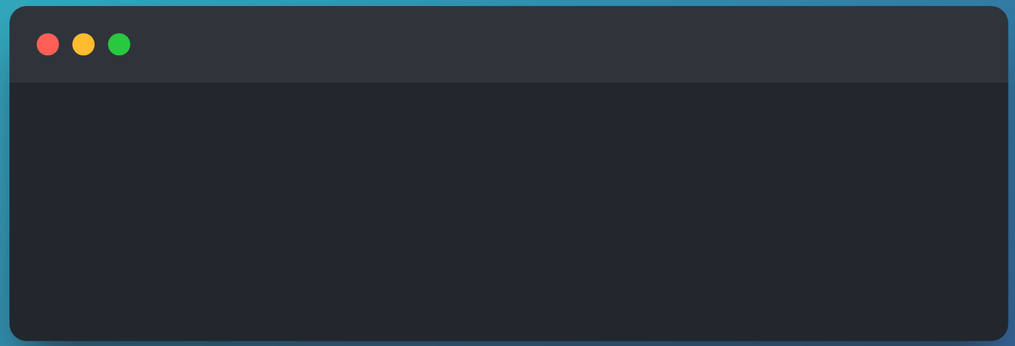
# Friendship without templates

```
//Friendship without templates
export class Point {
    int x;
    int y;
public:
    Point(int x, int y) : x(x), y(y) {}
    friend void print_point(const Point& p);
};

export void print_point(const Point& p) {
    fmt::print("Point: ({}, {})\n", p.x, p.y);
}
```

# Friendship without templates

```
//Friendship without templates
templates_1::Point p(1, 2);
print_point(p);
```

# Friendship without templates: Demo time!



# Class is not a template, but friends are templates

```
/*

    #2: Class is not template, friend function and friend class are templates.
        . Any template instantiations of print_point, regardless of the
          template argument, can access the private members of the Point class.
        . Notice the syntax used to declare it as a friend of the
          Point class: it uses the template syntax.
        . The same applies to the class template Canvas. It's declared as a friend
           of the Point class and any template instantiation, regardless of the
           template argument, can access its private members.
*/
```

# Class is not a template, but friends are templates

```cpp
//The class is not a template, but the friend function and class are templates.
export class Point {
    template<typename T>
    friend class Canvas;

    template <typename T>
    friend void print_point(const Point& p);

    int x;
    int y;

public:
    Point(int x, int y) : x(x), y(y) {}
};

export template<typename T>
class Canvas {
public:
    void draw(const Point& p) {
        if constexpr (std::is_same_v<T, int>) {
            fmt::print("Drawing integer point: ({}, {})\n", p.x, p.y);
        } else if constexpr (std::is_same_v<T, float>) {
            fmt::print("Drawing float point: ({:.2f}, {:.2f})\n"
                    , static_cast<float>(p.x)
                    , static_cast<float>(p.y));
        } else {
            fmt::print("Drawing generic point: ({}, {})\n", p.x, p.y);
        }
    }
};
```

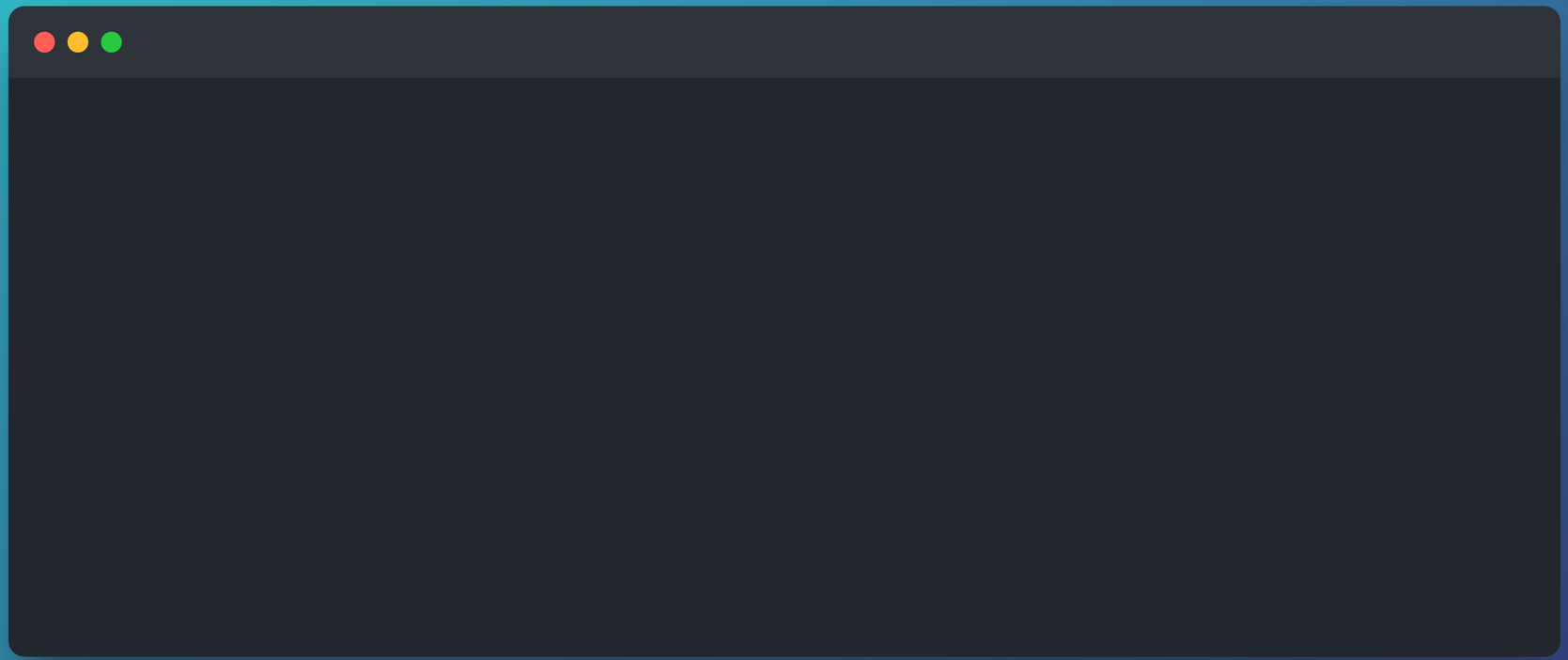
# Class is not a template, but friends are templates

```cpp
//The class is not a template, but the friend function and class are templates.
export template <typename T>
void print_point(const Point& p) {
    if constexpr (std::is_same_v<T, int>) {
        fmt::print("Point with int format: ({}, {})\n", p.x, p.y);
    } else if constexpr (std::is_same_v<T, float>) {
        fmt::print("Point with float format: ({:.2f}, {:.2f})\n"
                , static_cast<float>(p.x)
                , static_cast<float>(p.y));
    } else {
        fmt::print("Generic point: ({}, {})\n", p.x, p.y);
    }
}
```

# Class is not a template, but friends are templates

```
Point p1(3, 4);
Canvas<int> canvas1;
canvas1.draw(p1);
print_point<int>(p1);

Point p2(5, 7);
Canvas<float> canvas2;
canvas2.draw(p2); // Canvas<float> can access the private members of Point.
print_point<float>(p2);
print_point<float>(p1); // A float instance of print_point can access the private members of Point.
```

# Class is not a template, but friends are templates: Demo time!



# Class is not a template, but friends int instances or specializations

```
/*
        .#3: Class is not a template, but the friend function and class are
         specific specializations (or instances) of the template.
            . Specifically, friendship is only granted to int instances or specializations
                of the friend classes or functions.
            . print_point<double> and Canvas<double> cannot access the private
                members of Point for example.
*/
```

# Class is not a template, but friends int instances or specializations

```
class Point;    //Forward declaration

template  <typename T>
void print_point(const Point& p);   //Forward declaration

template <typename T>
class Canvas;   //Forward declaration

export class Point {
    //Only int instances of the friends have access to the private
    //members of any instance of the Point class.
    friend void print_point<int>(const Point& p);
    friend class Canvas<int>;
public:
    Point(int x, int y) : x(x), y(y) {}
private:
    int x;
    int y;
};
```

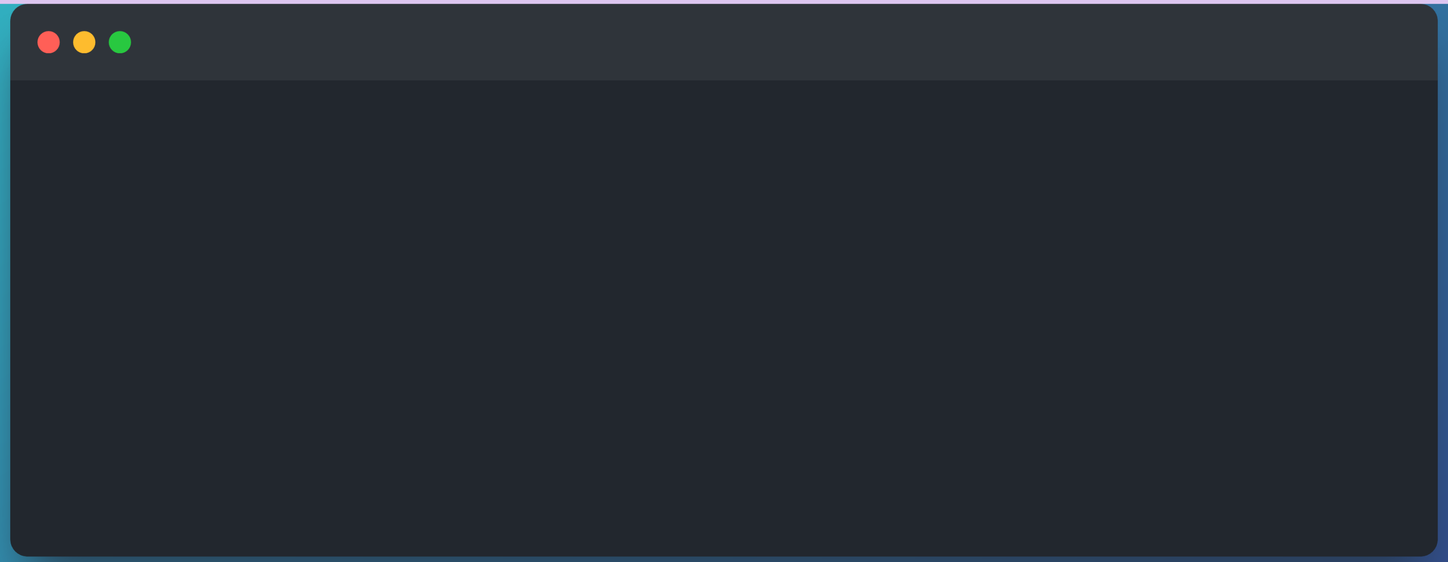
# Class is not a template, but friends int instances or specializations

```
//Definition if print_point
export template <typename T>
void print_point(const Point& p) {
    fmt::print("Point: ({}, {})\n", p.x, p.y);
}

//Definition of Canvas
export template <typename T>
class Canvas {
public:
    void draw(const Point& p) {
        fmt::print("Drawing point: ({}, {})\n", p.x, p.y);
    }
};

//Specializations
/*
template <>
void print_point<int>(const Point& p) {
    fmt::println("Point - function specialization : ({}, {})\n", p.x, p.y);
}


template <>
class Canvas<int> {
public:
    void draw(const Point& p) {
        fmt::println("Drawing point - specialization: ({}, {})\n", p.x, p.y)
    }
};
*/
```

# Class is not a template, but friends int instances or specializations

```
//Int instances of the friends.
templates_3::Point p(5, 6);

//print_point<int> can access the private members of Point<int>.
//If it was double, you'd get a compiler error.Canvas<int> can access the private
templates_3::print_point<int>(p);

//The same goes for the Canvas class template.
templates_3::Canvas<int> canvas;
canvas.draw(p);
```

# Class is not a template, but friends int instances or specializations: Demo time!



# Class is a template but the friend function is not a template

```
/*
    . Class is a template, and the friend function is not a template.
        . Granting access to a limited set of function overloads.
*/
```

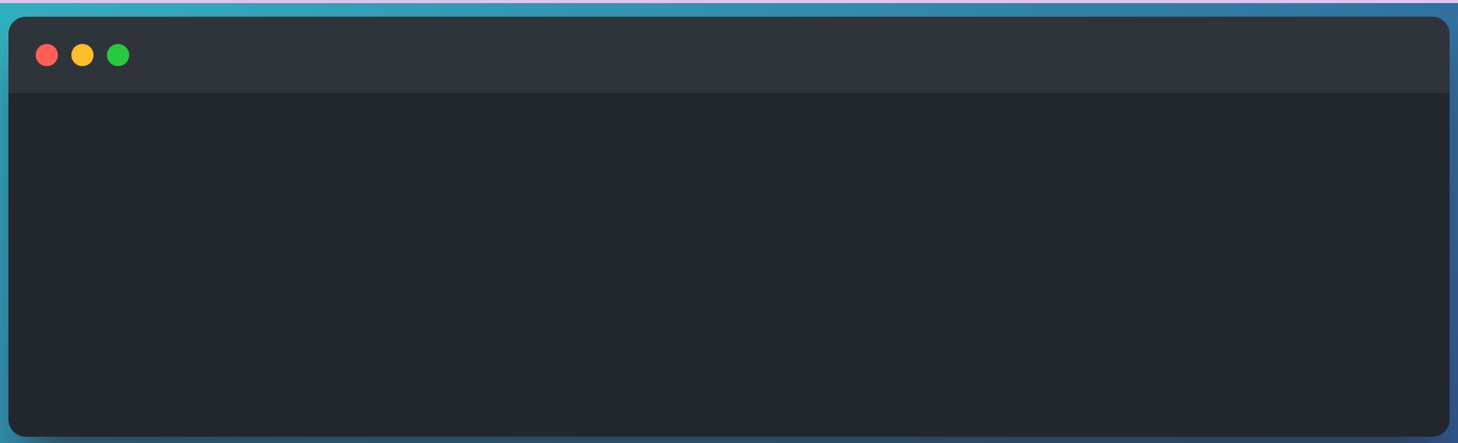
# Class is a template but the friend function is not a template

```
export template <typename T>
class Point {
    //The overload of print_point that takes a Point<int> as an argument can
    //access the private members of any instance of Point.
    friend void print_point(const Point<int>& p);
    int x;
    int y;
public:
    Point(T x, T y) : x(x), y(y) {}
};

export void print_point(const Point<int>& p) {
    fmt::print("Point: ({}, {})\n", p.x, p.y);
}

//This overload can't access the private members of any instance of Point.
//Error!
/*
export void print_point(const Point<double>& p) {
    fmt::print("Point: ({}, {})\n", p.x, p.y);
}
*/
```

# Class is a template but the friend function is not a template

```cpp
templates_4::Point<int> p(7, 8);
print_point(p);    // print_point can access the private members of Point<int>

templates_4::Point<double> p2(9.0, 10.0);
//print_point(p2);    // Error. Why?
```

# Class is a template but the friend function is not a template: Demo time!



# Class is a template and friends are templates

```
/*
            .#5: The class is a template, friend functions and classes are templates, but
              but we want a few instances of the friends to have access to the private members of the class.
            . Only int instances (or specializations) of the friends have access to the private members
                of Point.
            . Others can't. They will trigger a compiler error.
*/
```

# Class is a template and friends are templates

```
template <typename T>
class Canvas;

export template <typename T>
class Point {
    //Only int instances or specializations of the friends have access to private members.
    friend class Canvas<int>;
    friend void print_point<int>(const Point<int>& p);
public:
    Point (T x, T y) : x(x), y(y) {}
private:
    T x;
    T y;
};

export template <typename T>
void print_point(const Point<T>& p) {
    fmt::print("Point: ({}, {})\n", p.x, p.y); // Instances other than int will cause an error
}

template <>
void print_point<int>(const Point<int>& p) {
    fmt::print("Point - function specialization : ({}, {})\n", p.x, p.y);
}
```

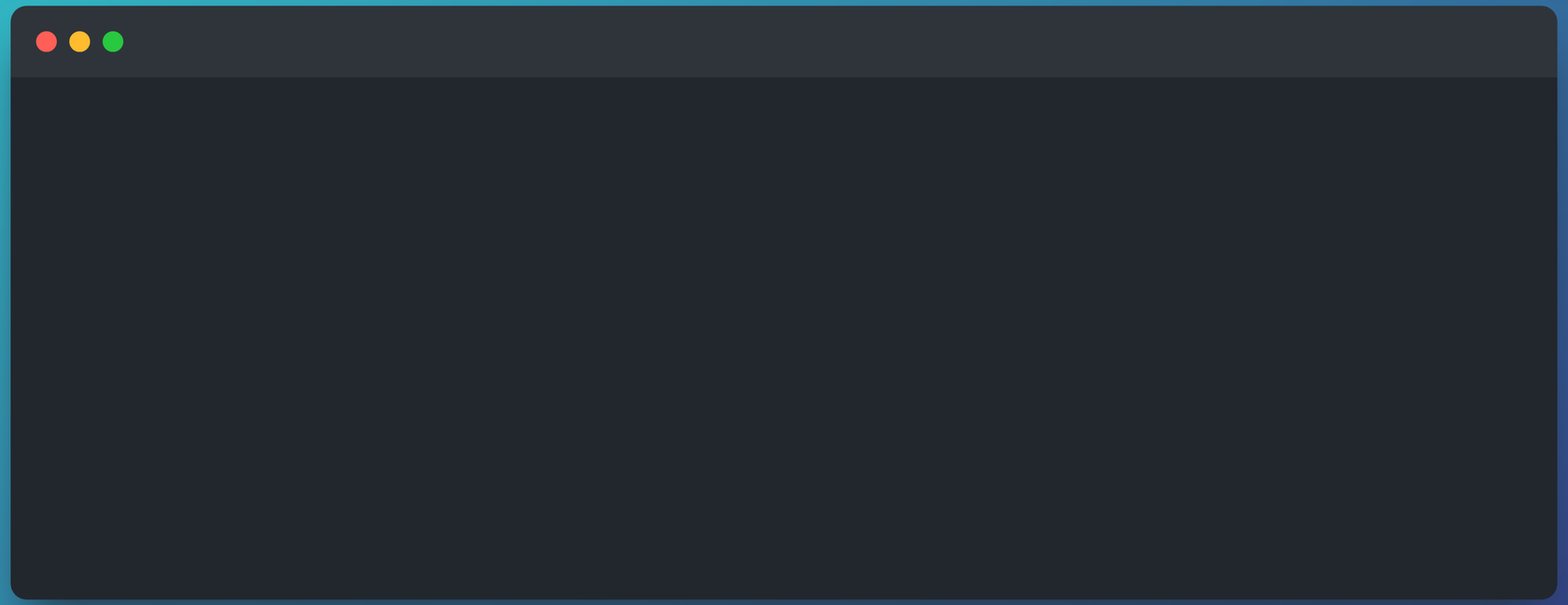
# Class is a template and friends are templates

```
template <typename T>
class Canvas;

export template <typename T>
class Point {
    //Only int instances or specializations of the friends have access to private members.
    friend class Canvas<int>;
    friend void print_point<int>(const Point<int>& p);
public:
    Point (T x, T y) : x(x), y(y) {}
private:
    T x;
    T y;
};

export template <typename T>
class Canvas {
public:
    void draw(const Point<T>& p) {
        fmt::print("Drawing point: ({}, {})\n", p.x, p.y); // Error: Instances othe than int will cause an error.
    }
};

template <>
class Canvas<int> {
public:
    void draw(const Point<int>& p) {
        fmt::print("Drawing point - specialization: ({}, {})\n", p.x, p.y);
    }
};
```

# Class is a template and friends are templates.

```cpp
templates_5::Point<int> p(11, 12);
print_point(p);    // print_point can access the private members of Point<int>

templates_5::Canvas<int> canvas;
canvas.draw(p);    // Canvas<int> can access the private members of Point<int>

// Creating a Point<double> instance
templates_5::Point<double> p2(13.0, 14.0);
//print_point(p2);    // Error: print_point can't access the private members of // Point<double>
```

# Class is a template and friends are templates: Demo time!



# Class is a template and friends are templates.

```
/*
  .#6: Class is template, friends are template, and we want any instance
       of the friend templates to have access to private members of the class.
*/
```

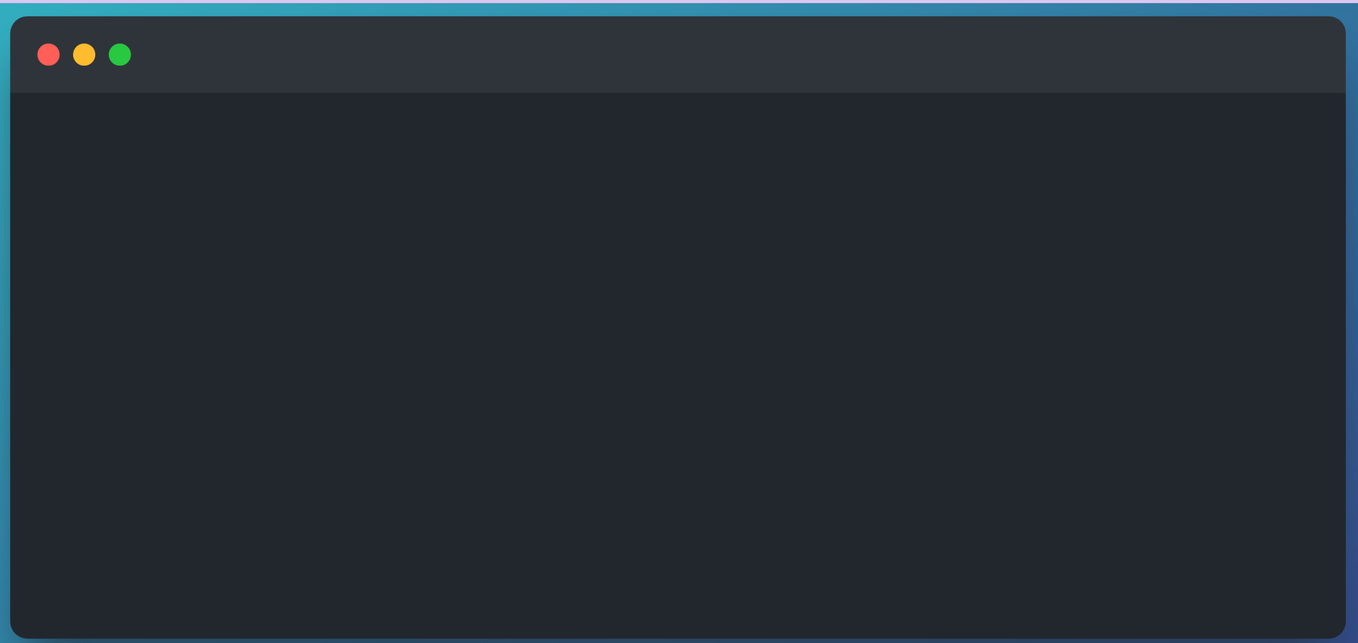
# Class is a template and friends are templates.

```cpp
//template <typename T>
//class Canvas;

export template <typename T>
class Point {
  template <typename U>
  friend class Canvas;

  template <typename U>
  friend void print_point(const Point<U>& p);

  T x;
  T y;

public:
  Point(T x, T y) : x(x), y(y) {}
};

export template <typename T>
void print_point(const Point<T>& p) {
  fmt::print("Point: ({}, {})\n", p.x, p.y);
}

export template <typename T>
class Canvas {
public:
  void draw(const Point<T>& p) {
    fmt::print("Drawing point: ({}, {})\n", p.x, p.y);
  }
};
```

# Class is a template and friends are templates.

```cpp
templates_6::Point<int> p(15, 16);
print_point(p);  // print_point can access the private members of Point<int>

templates_6::Canvas<int> canvas;
canvas.draw(p);  // Canvas<int> can access the private members of Point<int>

// Creating a Point<double> instance
templates_6::Point<double> p2(17.0, 18.0);
print_point(p2);

templates_6::Canvas<double> canvas2;
canvas2.draw(p2);
```

# Class is a template and friends are templates: Demo time!



# Class is a template and friends are templates.

```
/*
    .#7: Class is a template, friends are templates, and we only want friends
        whose template parameter matches that  of the class to have access to
        the private members of the class.
*/
```

# Class is a template and friends are templates.

```
template <typename T>
class Canvas;

export template <typename T>
class Point {
    //We only grant access to friends whose template paramters match that of the class.
    friend class Canvas<T>;
    friend void print_point<T>(const Point<T>& p);
public:
    Point(T x, T y) : x(x), y(y) {}
private:
    T x;
    T y;
};

export template <typename T>
void print_point(const Point<T>& p) {
    fmt::print("Point: ({}, {})\n", p.x, p.y);
}

export template <typename T>
class Canvas {
public:
    void draw(const Point<T>& p) {
        fmt::print("Drawing point: ({}, {})\n", p.x, p.y);
    }
};
```

# Class is a template and friends are templates.

```
templates_7::Point<int> p(19, 20);
print_point(p);    // print_point can access the private members of Point<int>

templates_7::Canvas<int> canvas;
canvas.draw(p);    // Canvas<int> can access the private members of Point<int>

//Creating a Point<double> instance
templates_7::Point<double> p2(21.0, 22.0);
print_point<double>(p2);
//print_point<int>(p2);    // Error

templates_7::Canvas<double> canvas2;
canvas2.draw(p2);
//canvas2.draw(p);    // Error
```

# Class is a template and friends are templates.

```
templates_7::Point<int> p(19, 20);
print_point(p);    // print_point can access the private members of Point<int>

templates_7::Canvas<int> canvas;
canvas.draw(p);    // Canvas<int> can access the private members of Point<int>

//Creating a Point<double> instance
templates_7::Point<double> p2(21.0, 22.0);
print_point<double>(p2);
//print_point<int>(p2);    // Error

templates_7::Canvas<double> canvas2;
canvas2.draw(p2);
//canvas2.draw(p);    // Error
```