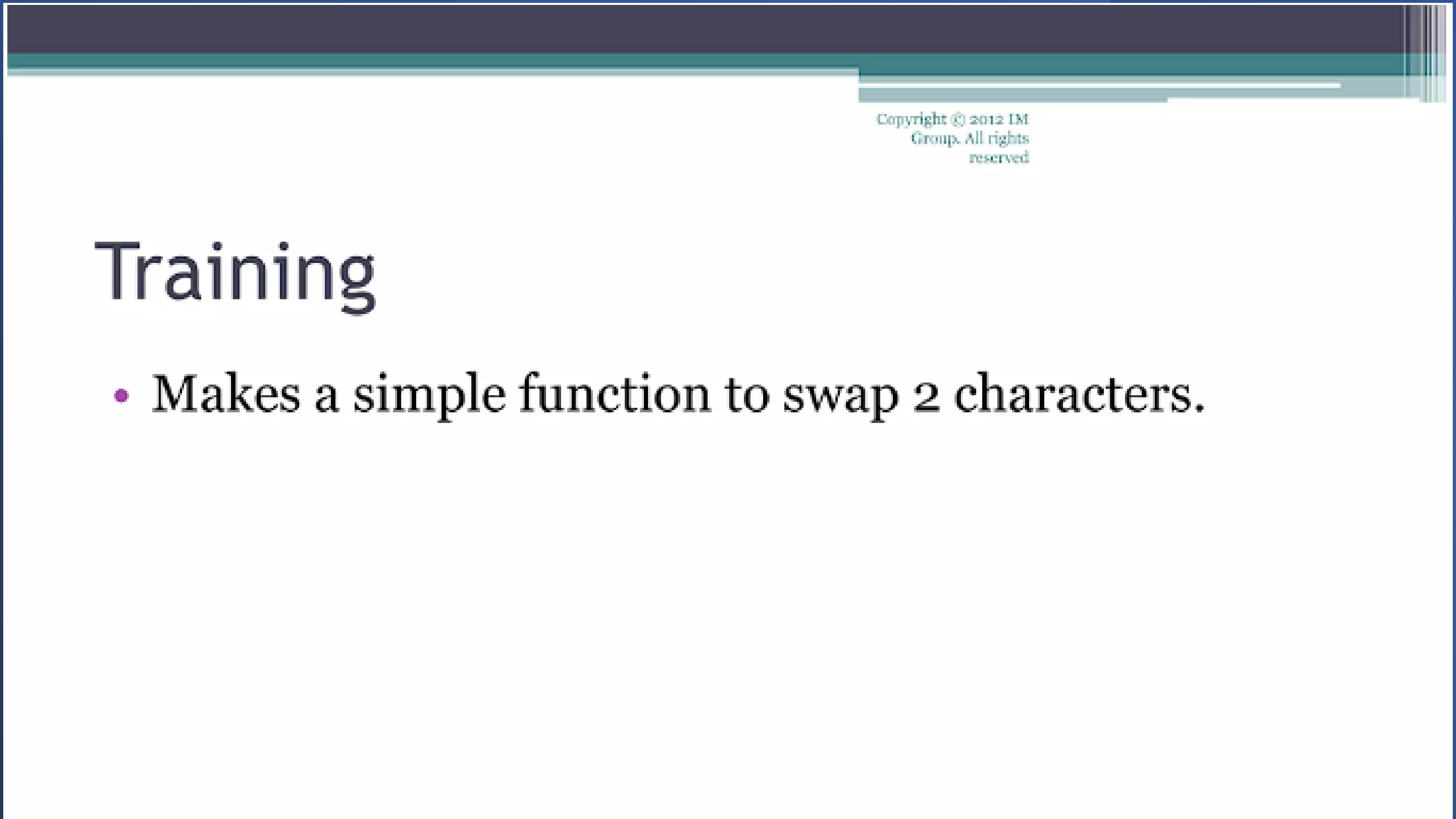
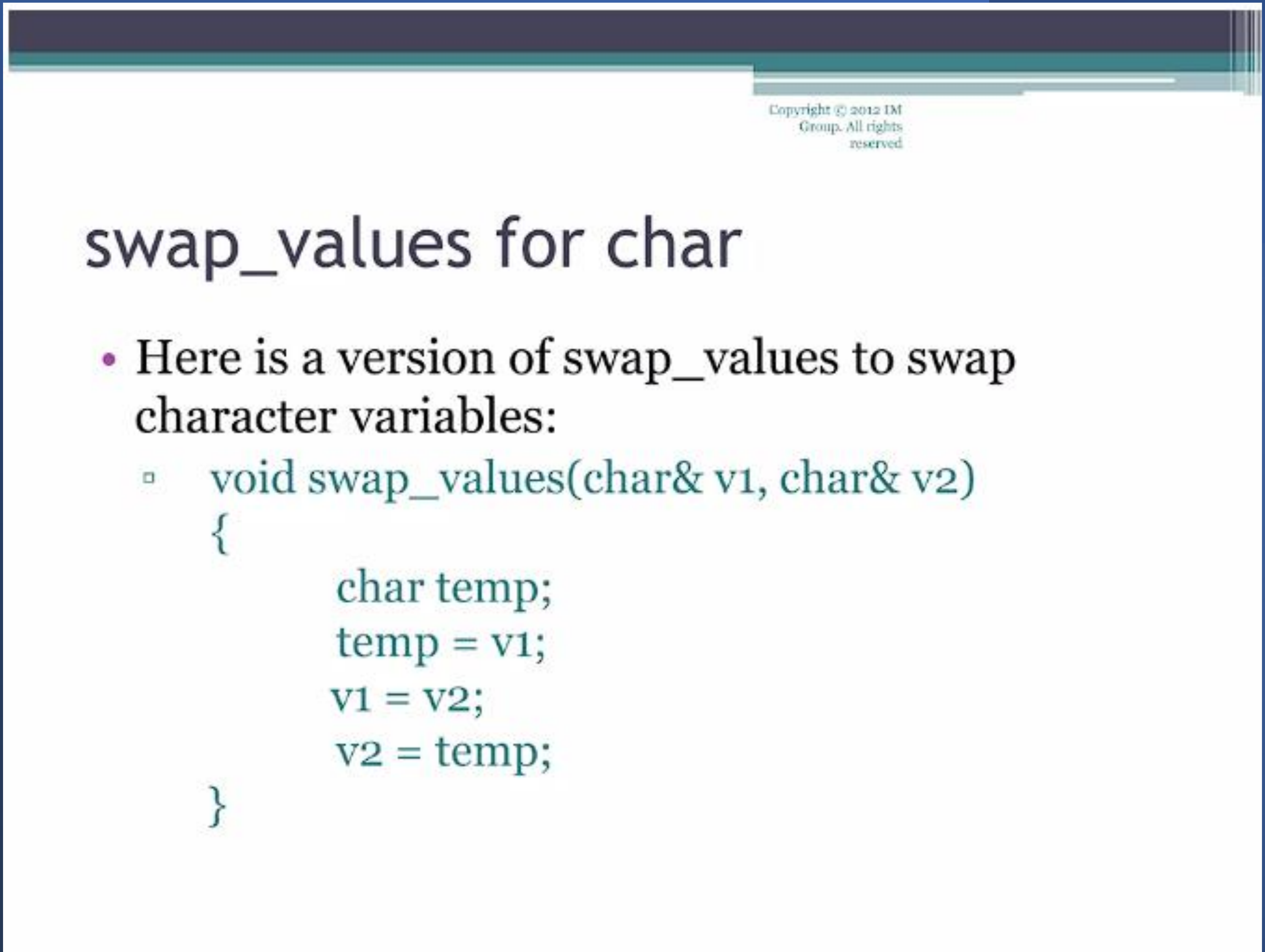
Templates

Object Oriented Programming



Training

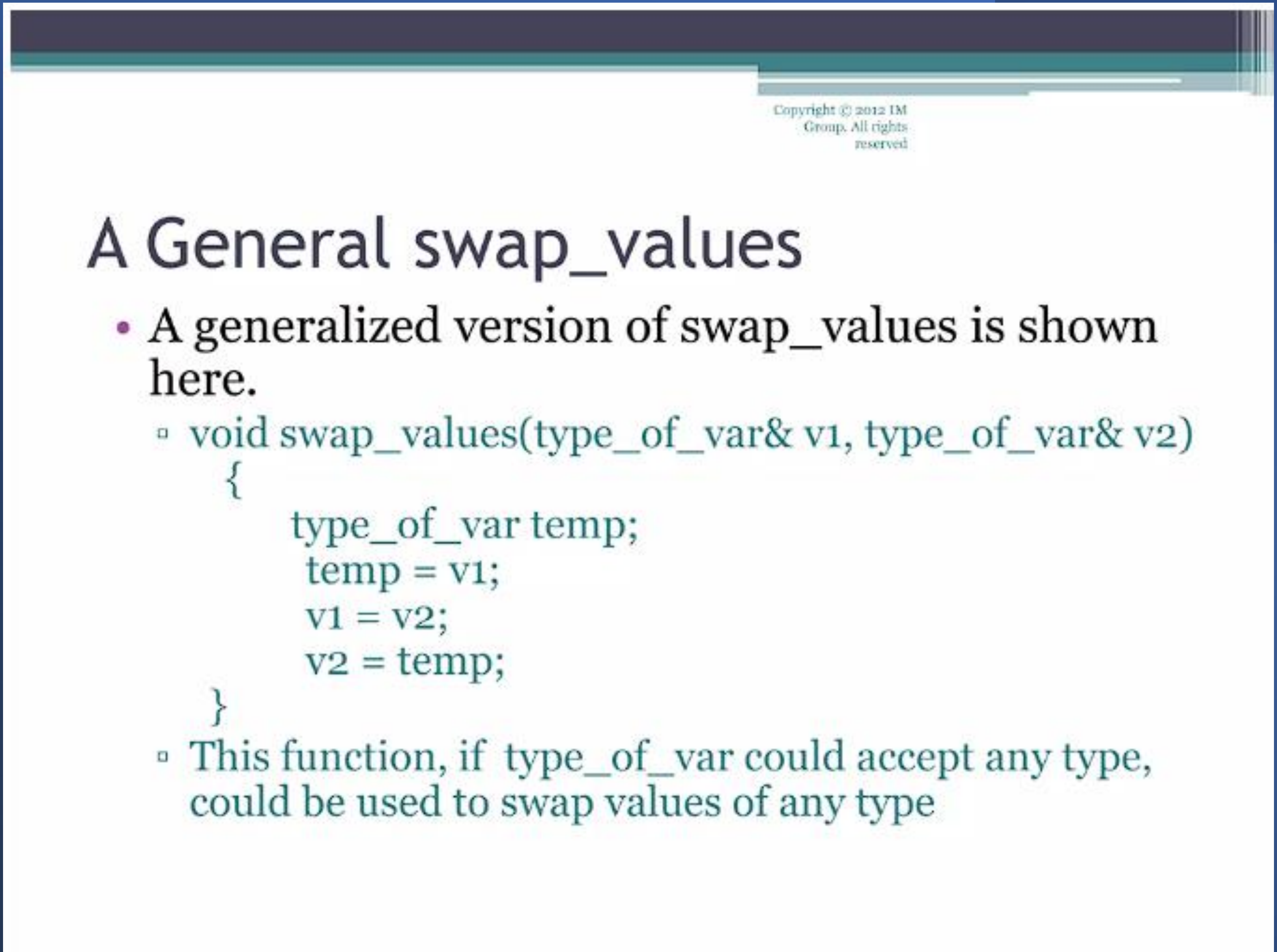
- Makes a simple function to swap 2 characters.



swap_values for char

- Here is a version of swap_values to swap character variables:

```
void swap_values(char& v1, char& v2)
{
        char temp;
        temp = v1;
        v1 = v2;
        v2 = temp;
}
```



A General swap_values

- A generalized version of swap_values is shown here.

  -
    ```
    void swap_values(type_of_var& v1, type_of_var& v2)
      {
           type_of_var temp;
            temp = v1;
            v1 = v2;
            v2 = temp;
     }
    ```
  - This function, if type_of_var could accept any type, could be used to swap values of any type

Templates for Functions

- A C++ function template will allow swap_values to swap values of two variables of the same type
  - Example:

Type parameter

Template prefix →

```
template<class T>
  void swap_values(T& v1, T& v2)
  {
    T temp;
     temp = v1;
     v1 = v2;
     v = temp;
   }
```

Template Details

- template<class T> is the template prefix
  - Tells compiler that the declaration or definition that follows is a template
  - Tells compiler that T is a type parameter
    - class means type in this context
      (typename could replace class but class is usually used)
    - T can be replaced by any type argument
      (whether the type is a class or not)
- A template overloads the function name by replacing T with the type used in a function call

Calling a Template Function

- Calling a function defined with a template is identical to calling a normal function
  - Example:

    To call the template version of swap_values

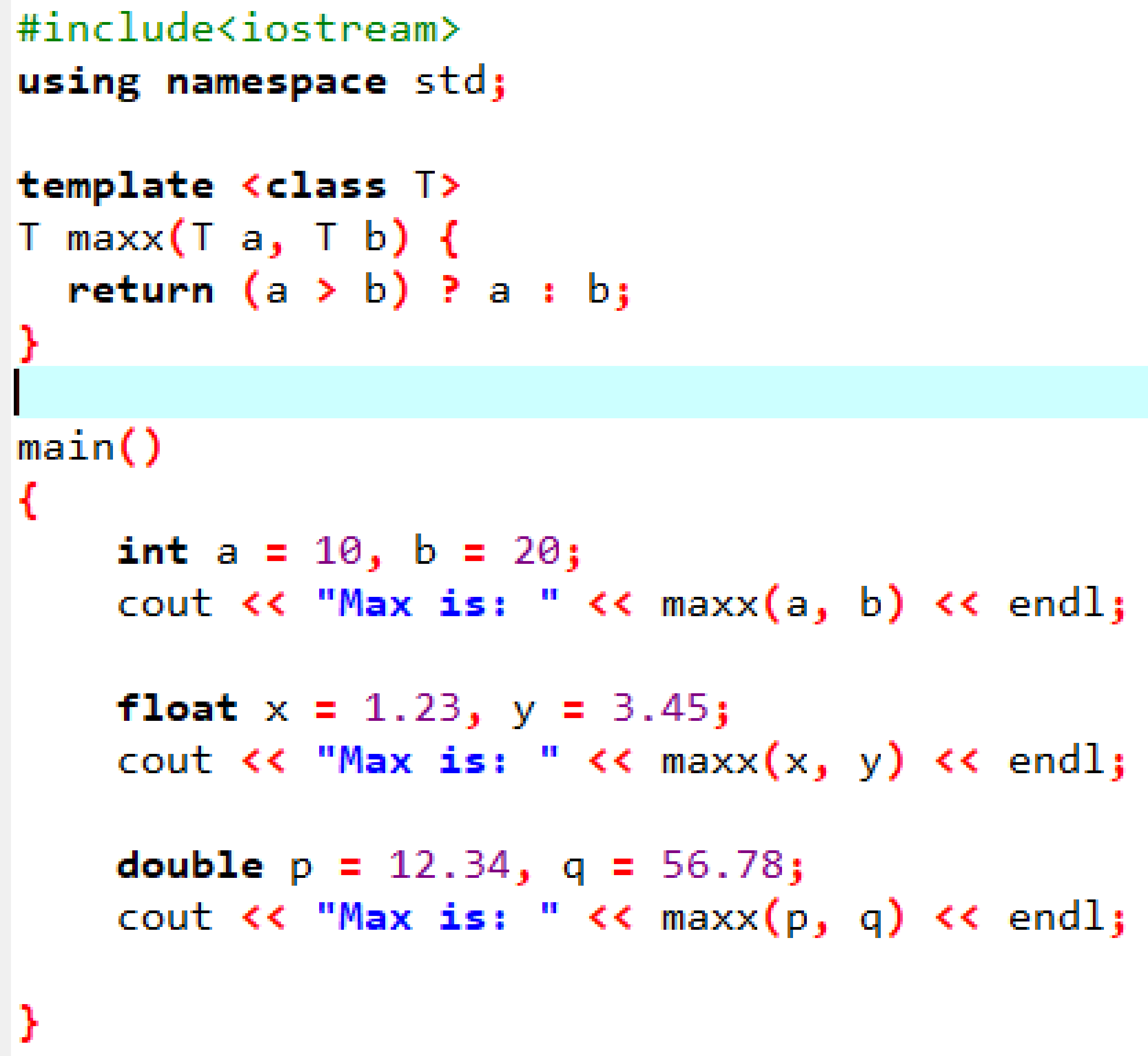
    ```
    char s1, s2;
     int i1, i2;
     ...
     swap_values(s1, s2);
     swap_values(i1, i2);
    ```

    - The compiler checks the argument types and generates an appropriate version of swap_values

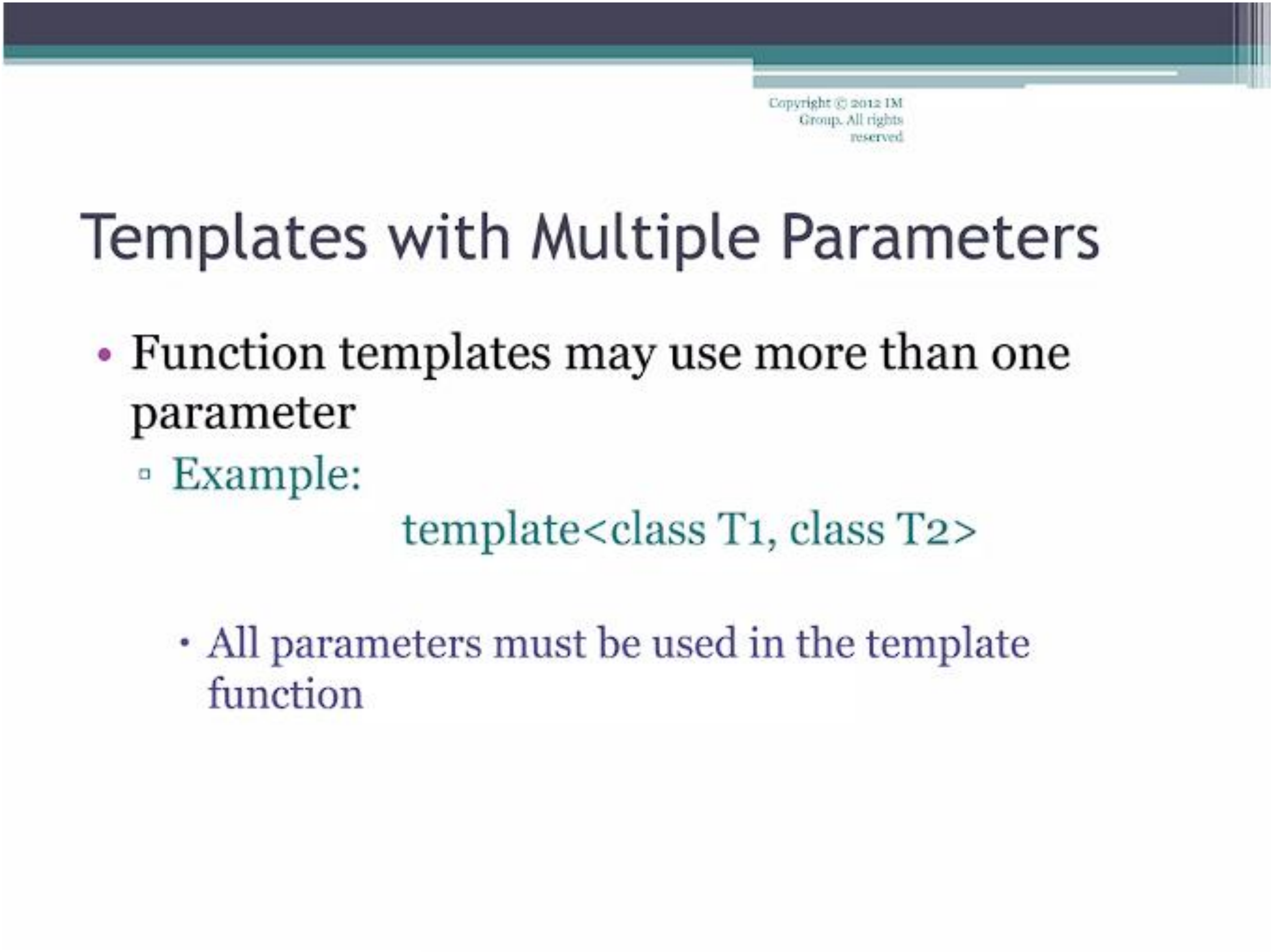
Templates and Declarations

- A function template may also have a separate declaration
  - The template prefix and type parameter are used
  - Depending on your compiler
    - You may, or may not, be able to separate declaration and definitions of template functions just as you do with regular functions
  - To be safe, place template function definitions in the same file where they are used…with no declaration
    - A file included with #include is, in most cases, equivalent to being "in the same file“
    - This means including the .cpp file or .h file with implementation code

Example



```
#include<iostream>
using namespace std;

template <class T>
T maxx(T a, T b) {
  return (a > b) ? a : b;
}

main()
{
    int a = 10, b = 20;
    cout << "Max is: " << maxx(a, b) << endl;

    float x = 1.23, y = 3.45;
    cout << "Max is: " << maxx(x, y) << endl;

    double p = 12.34, q = 56.78;
    cout << "Max is: " << maxx(p, q) << endl;

}
```



Templates with Multiple Parameters

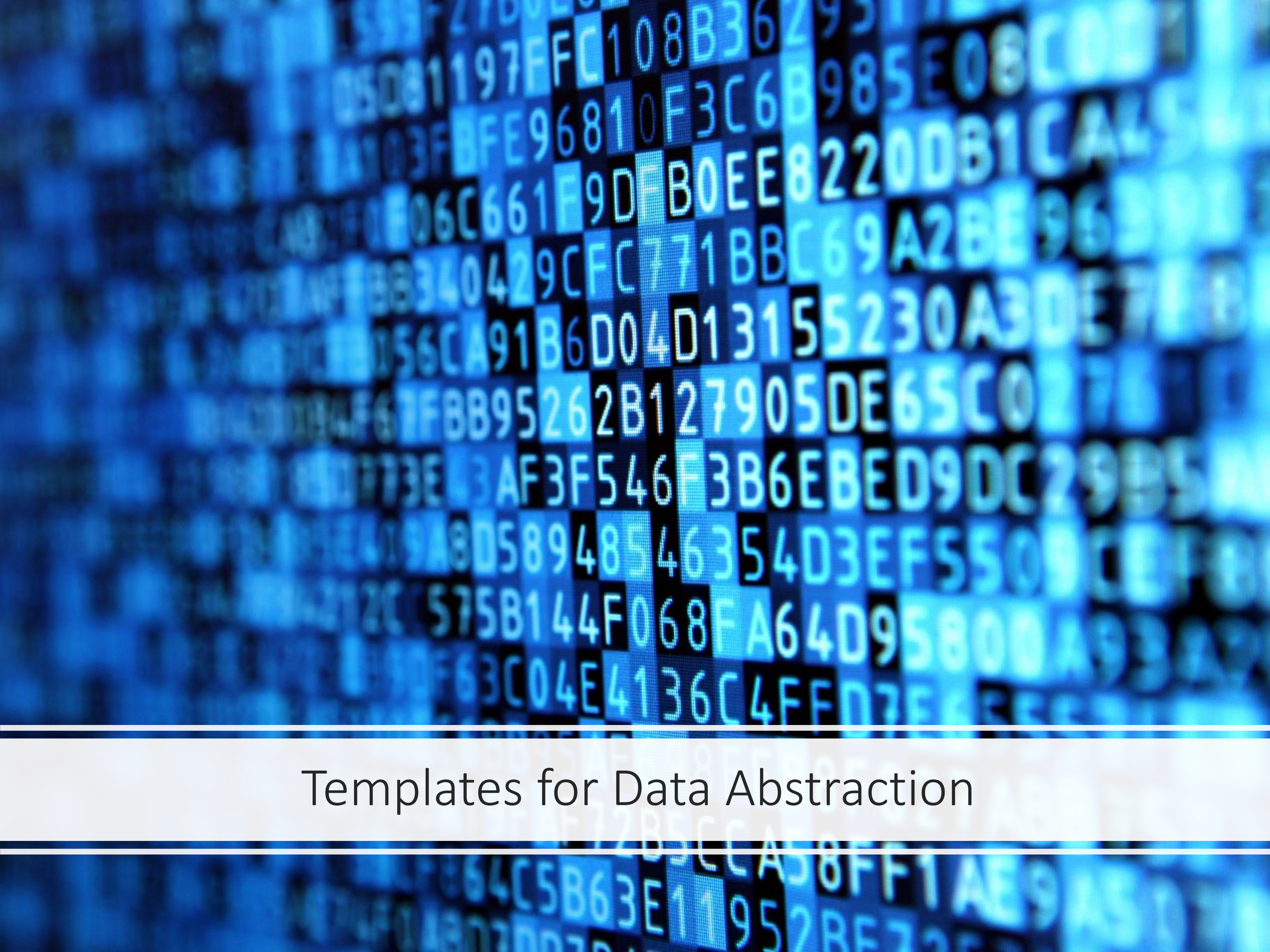
- Function templates may use more than one parameter
  - Example:

    ```
    template<class T1, class T2>
    ```

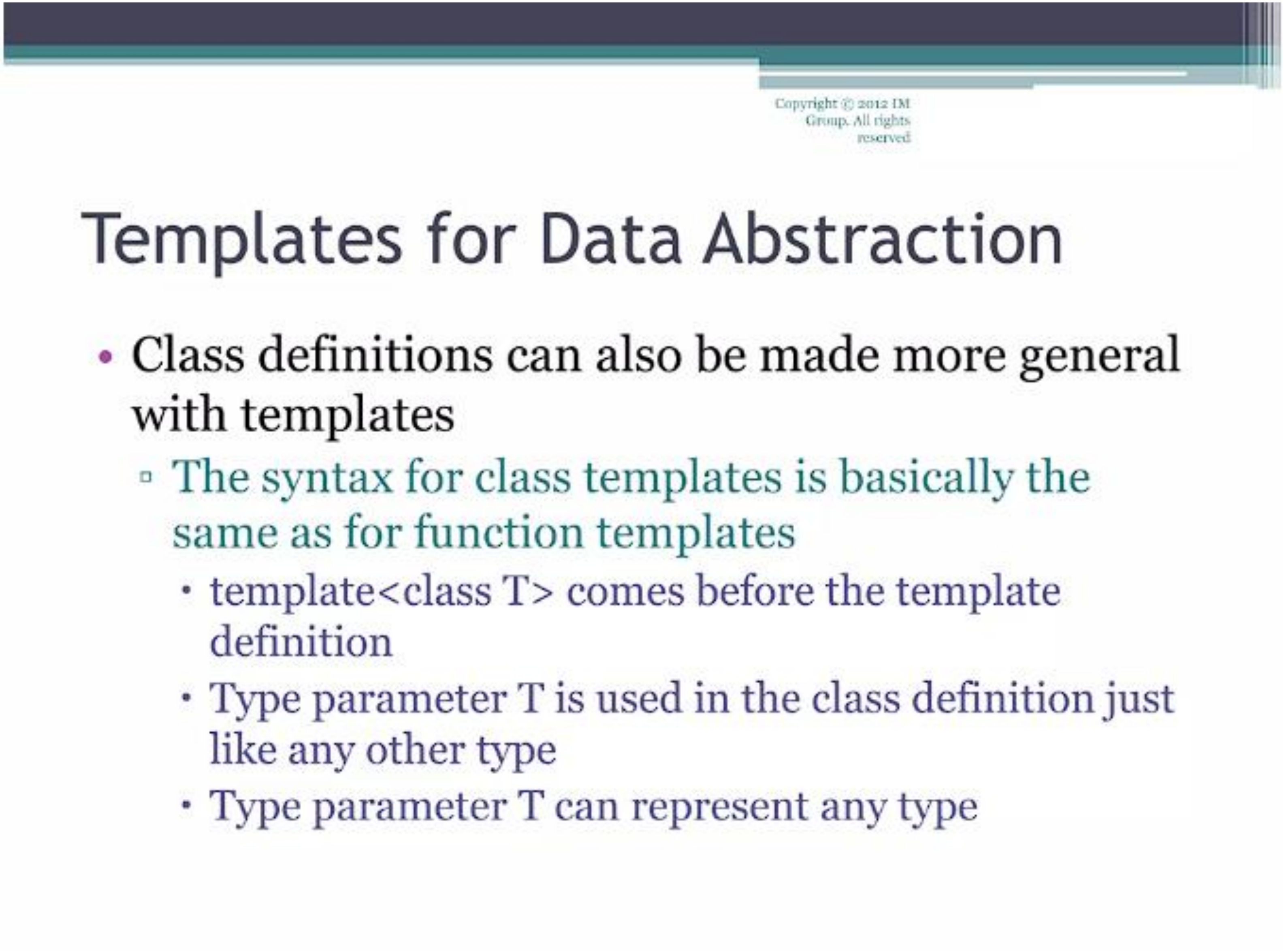
    - All parameters must be used in the template function

Defining Templates

- When defining a template it is a good idea…
  - To start with an ordinary function that accomplishes the task with one type
    - It is often easier to deal with a concrete case rather than the general case
  - Then debug the ordinary function
  - Next convert the function to a template by replacing type names with a type parameter



Templates for Data Abstraction



Templates for Data Abstraction

- Class definitions can also be made more general with templates
  - The syntax for class templates is basically the same as for function templates
    - template<class T> comes before the template definition
    - Type parameter T is used in the class definition just like any other type
    - Type parameter T can represent any type

A Class Template

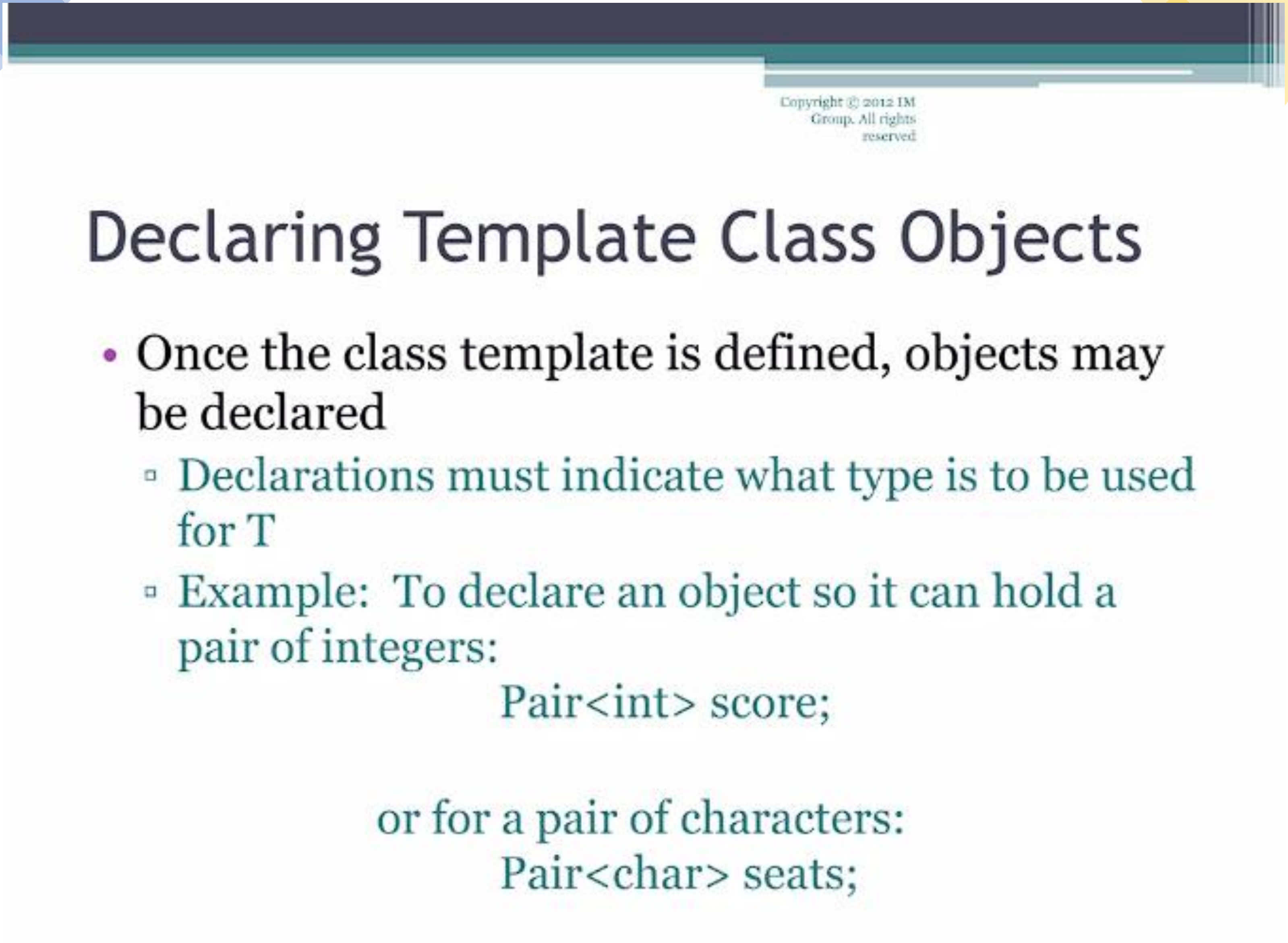
- The following is a class template
  - An object of this class contains a pair of values of type T
  - template <class T>

    ```
    class Pair
    {
            public:
                    Pair( );
                    Pair( T first_value, T second_value);

                    ...
    ```

continued on next slide

Template Class Pair (cont.)

```
▫   void set_element(int position, T value);
   //Precondition: position is 1 or 2
   //Postcondition:  position indicated is set to value

   T get_element(int position) const;
   // Precondition:  position is 1 or 2
   // Returns value in position indicated

  private:
     T first;
     T second;
  };
```



Declaring Template Class Objects

- Once the class template is defined, objects may be declared
  - Declarations must indicate what type is to be used for T
  - Example: To declare an object so it can hold a pair of integers:

```
Pair<int> score;
```

or for a pair of characters:

```
Pair<char> seats;
```

Using the Objects

- After declaration, objects based on a template class are used just like any other objects
  - Continuing the previous example:

```
score.set_element(1,3);
score.set_element(2,0);
seats.set_element(1, 'A');
```

Defining the Member Functions

- Member functions of a template class are defined the same way as member functions of ordinary classes
  - The only difference is that the member function definitions are themselves templates

Defining a Pair Constructor

- This is a definition of the constructor for class Pair that takes two arguments

```
template<class T>
 Pair<T>::Pair(T first_value, T second_value)
          : first(first_value), second(second_value)
  {
    //No body needed due to initialization above
  }
```

  - The class name includes <T>

Defining set_element

- Here is a definition for set_element in the template class Pair

```
void Pair<T>::set_element(int position, T value)
{
    if (position = = 1)
      first = value;
    else if (position = = 2)
      second = value;
    else
     ...
}
```

Template Class Names as Parameters

- The name of a template class may be used as the type of a function parameter
  - Example: To create a parameter of type Pair<int>:

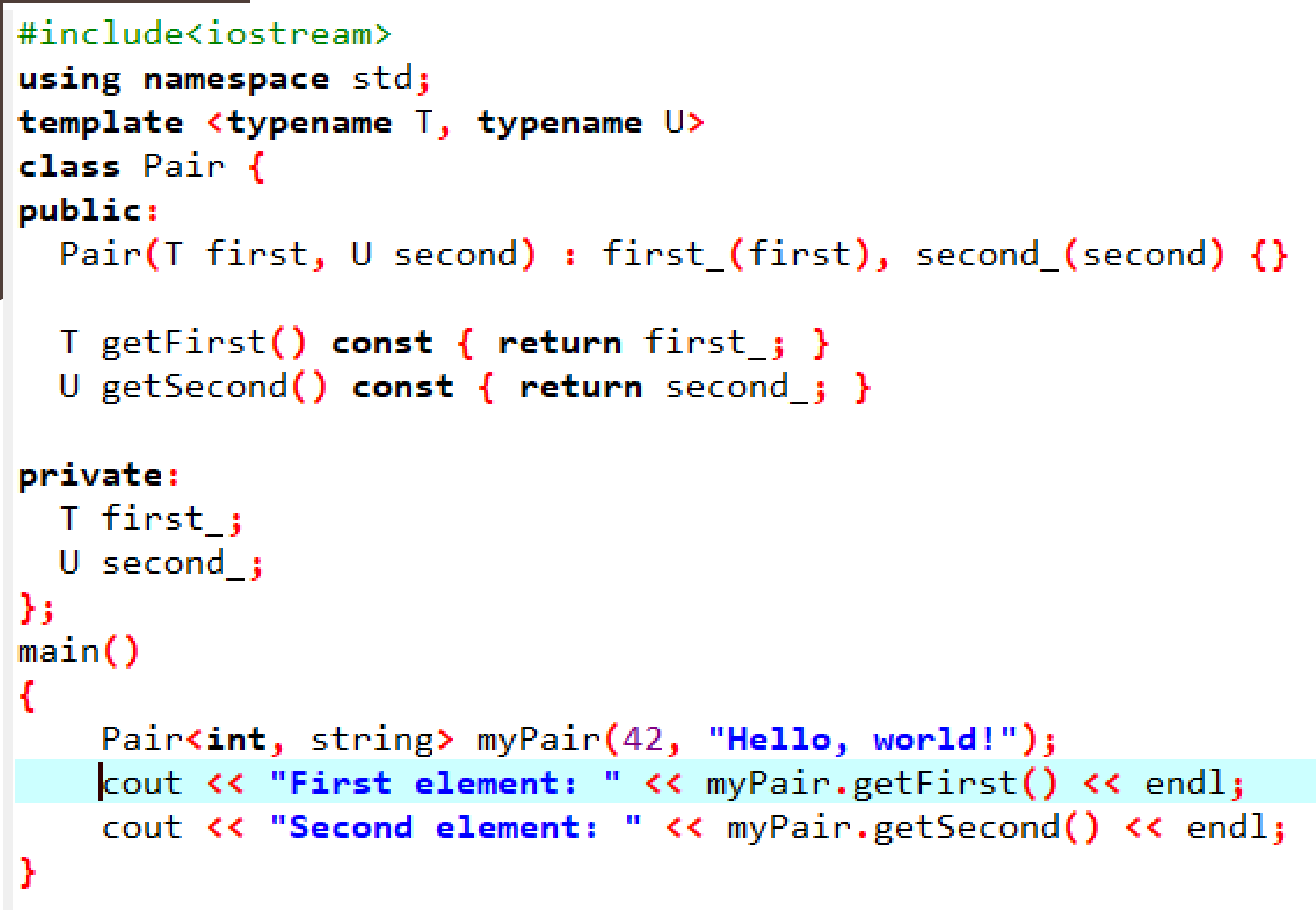
```
int add_up(const Pair<int>& the_pair);
//Returns the sum of two integers in the_pair
```

Template Functions with Template Class Parameters

- Function add_up from a previous example can be made more general as a template function:

```
        template<class T>
        T add_up(const Pair<T>& the_pair)
         //Precondition:  operator + is defined for T
         //Returns sum of the two values in
the_pair
```

Example



```
#include<iostream>
using namespace std;
template <typename T, typename U>
class Pair {
public:
  Pair(T first, U second) : first_(first), second_(second) {}

  T getFirst() const { return first_; }
  U getSecond() const { return second_; }

private:
  T first_;
  U second_;
};
main()
{
    Pair<int, string> myPair(42, "Hello, world!");
    cout << "First element: " << myPair.getFirst() << endl;
    cout << "Second element: " << myPair.getSecond() << endl;
}
```

You can also use typename to define template

- In C++, both the "typename" and "class" keywords are used to declare template parameters in class templates. They are interchangeable and can be used interchangeably in most cases. However, there is a subtle difference between the two keywords that can affect the code in certain situations.
- Class might cause ambiguity sometimes if not used properly

```
template <typename T>
class MyTemplate {
public:
  void doSomething() {
    T::myFunction();
  }
};
```

Thank you