# C++ Notes

# Part 1-2

**Slides were created from CodewithMosh.com**

# Formatting Output

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    cout << left;
    cout << setw( n: 10) << "Spring" << setw( n: 10) << "Nice"
         << setw( n: 10) << "Summer" << setw( n: 10) << "Hot";

    return 0;
}
```

**left :** Left alignment

**right :** Right alignment

**setw :** String formatting

# Formatting Output


**fixed :** To force fixed point notation.

- 6 digits after the decimal point.

- Stays in effect until changed.

**setprecision :** To use certain number of digits after the decimal point.

# Data Types Size and Limits

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << numeric_limits<int>::min() << endl
         << numeric_limits<int>::max();
}
```

**numeric_limits :** Generic class

# Data Types: Overflowing

```cpp
#include <iostream>


using namespace std;

int main() {
    int number = numeric_limits<int>::max();
    number++;
    cout << number;
}
```

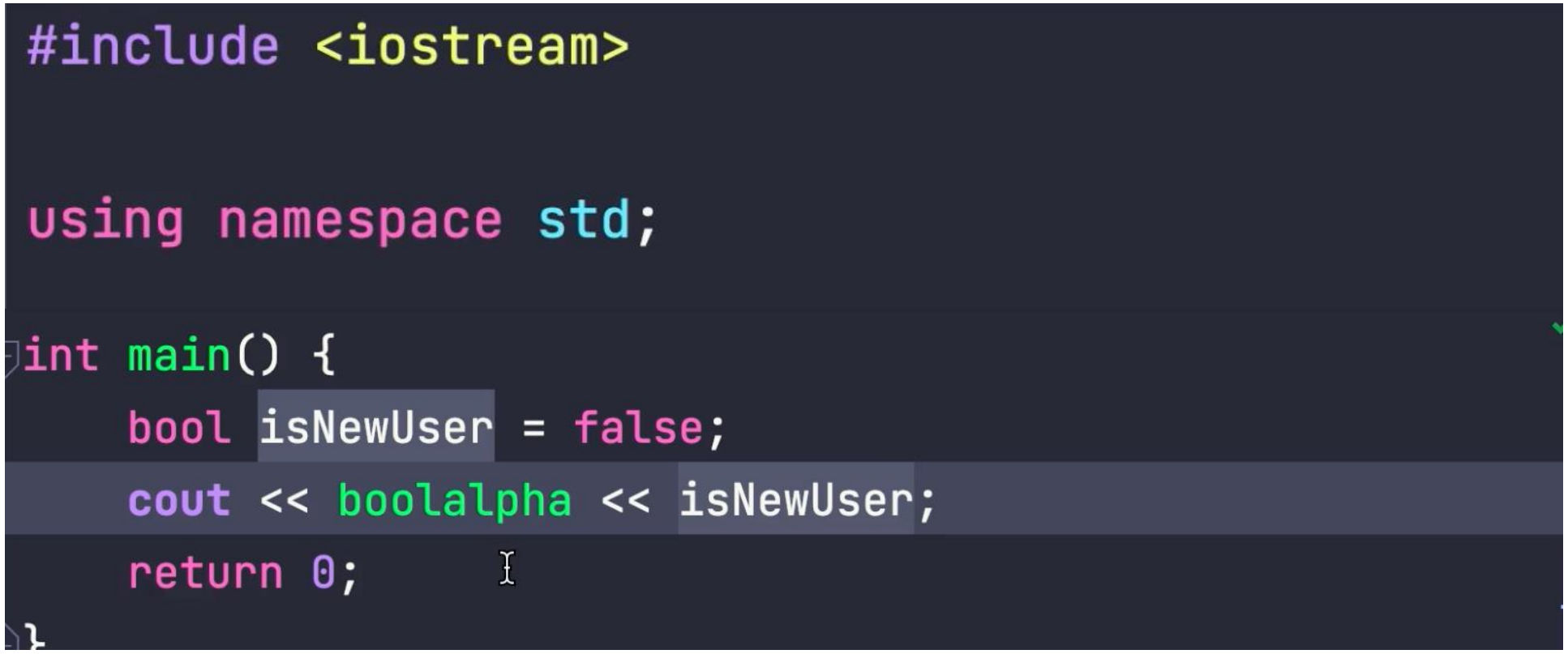
**numeric_limits :** Generic class

# Data Types: Underflowing

```cpp
#include <iostream>


using namespace std;

int main() {
    int number = numeric_limits<int>::min();
    number--;
    cout << number;
}
```

**numeric_limits :** Generic class

# Booleans



```cpp
#include <iostream>


using namespace std;

int main() {
    bool isNewUser = false;
    cout << boolalpha << isNewUser;
    return 0;
}
```

**boolalpha:** Display the value of a boolean variable as true/false

**noboolalpha:** Display the value of a boolean variable as 1/0 (Default)

# Characters

```cpp
#include <iostream>

using namespace std;

int main() {
    char ch = 'a';
    cout << ch;
    return 0;
}
```

```cpp
int main() {
    char ch = 98;
    cout << ch;
    return 0;
}
```

Even this is allowed, it is not a good practice.

```cpp
int main() {
    char ch = 'a';
    cout << +ch;
    return 0;
```

+ before the variable name tells the compiler to treat it as a number.

# Strings

```cpp
using namespace std;

int main() {
    string name;
    cout << "Enter your name: ";
    cin >> name;
    getline( &: cin,  &: name);
    cout << "Hi " << name;
    return 0;
}
```

- Reading a string with space using “cin”function  is not  possible.
- Instead we need to use “getline” function.

# Arrays

```cpp
#include <iostream>

using namespace std;

int main() {
    int numbers[5];
    numbers[1] = 10;
    cout << numbers[5]; // size - 1

    return 0;
}
```

- Array index starts from 0 (zero)
- Referencing an array index outside of the range can cause problem.
  - Program is still compiled but will produce a garbage value.

```cpp
int numbers[5] = {[0]: 10, [1]: 20};
```

# Type Conversion (Casting)

```cpp
int main() {
    int x = 1;
    double y = 2.0;
    int z = x + (int)y;
    cout << z;
    return 0;
}
```

- C-style casting: Casting problem cannot be caught until the run time.

```cpp
int main() {
    int x = 1;
    double y = 2.0;
    int z = x + static_cast<int>(y);
    cout << z;
    return 0;
}
```

- C++-style casting: Casting problem can be caught during the compile time.

# Type Conversion (Casting)

```cpp
#include <iostream>

using namespace std;

int main() {
    int x = 10;
    int y = 3;
    double z = static_cast<double>(x) / y;
    cout << z;
    return 0;
}
```

Example

# Decision Making

Comparison operators

Logical operators

If statements

Switch statements

Conditional operator

# Comparison Operators

```cpp
int main() {
    int x = 10;
    int y = 5;
    bool result = x == y;
    cout << boolalpha << result;
    return 0;
}
```

==
!=
>, >=
<, <=

```cpp
int main() {
    int x = 10;
    double y = 5;
    bool result = x == y;
    cout << boolalpha << result;
    return 0;
}
```

- The compiler automatically casts smaller value (less precise) to larger (more precise) types.
- Int 10 value is casted to double type.

# Comparison Operators

```cpp
int main() {
    char first = 'a';
    char second = 'A';
    bool result = first == second;
    cout << result;
    return 0;
}
```

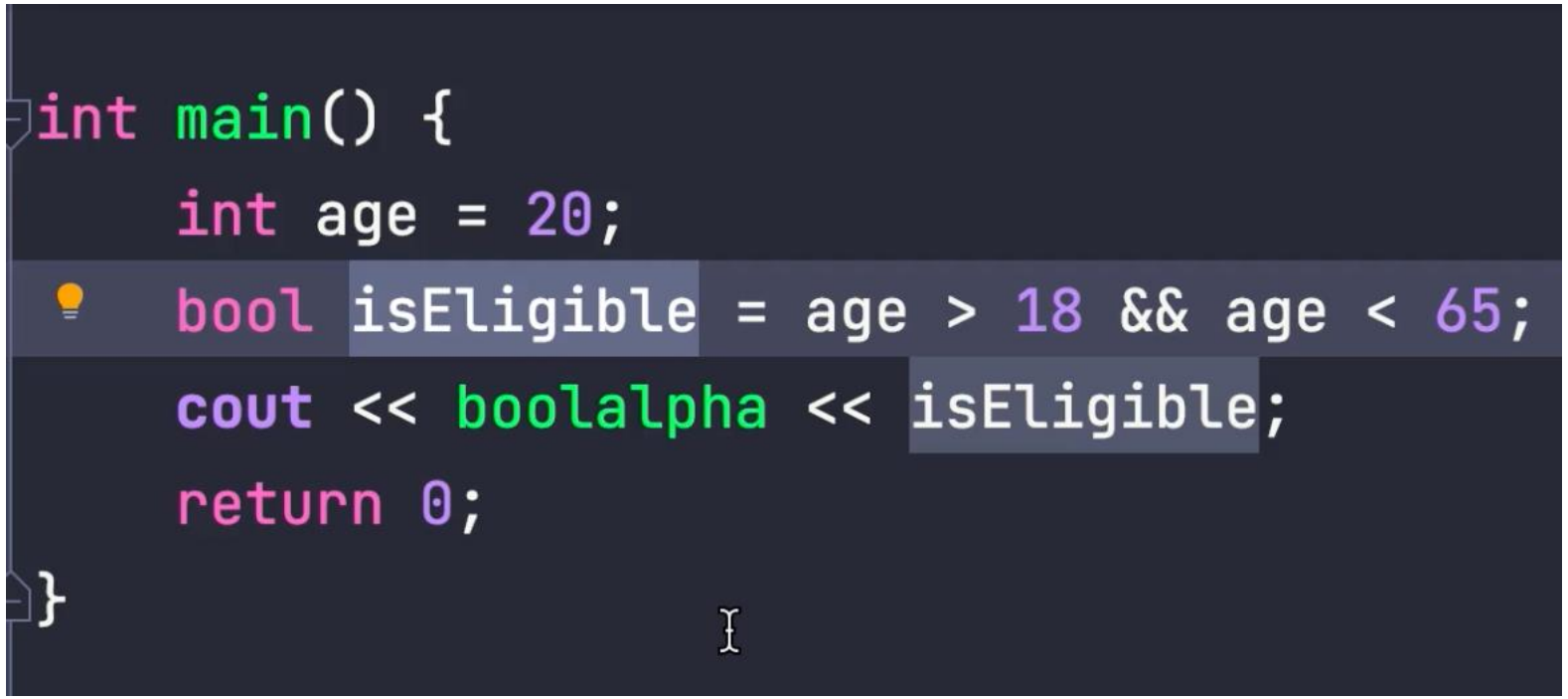
==

!=

>, >=

<, <=

# Logical Operators



```cpp
int main() {
    int age = 20;
    bool isEligible = age > 18 && age < 65;
    cout << boolalpha << isEligible;
    return 0;
}
```

**&& : AND**

**|| : OR**

**! : NOT**

- **Evaluation is from left-to-right.**

# Logical Operators

&& : AND

|| : OR

! : NOT

```
// ()
// !
// &&
// ||
bool a = true;
bool b = false;
bool c = false;
bool result = a || b && c;
cout << boolalpha << result;
return 0;
}
```

## Example

- Order is (), ! , &&, ||

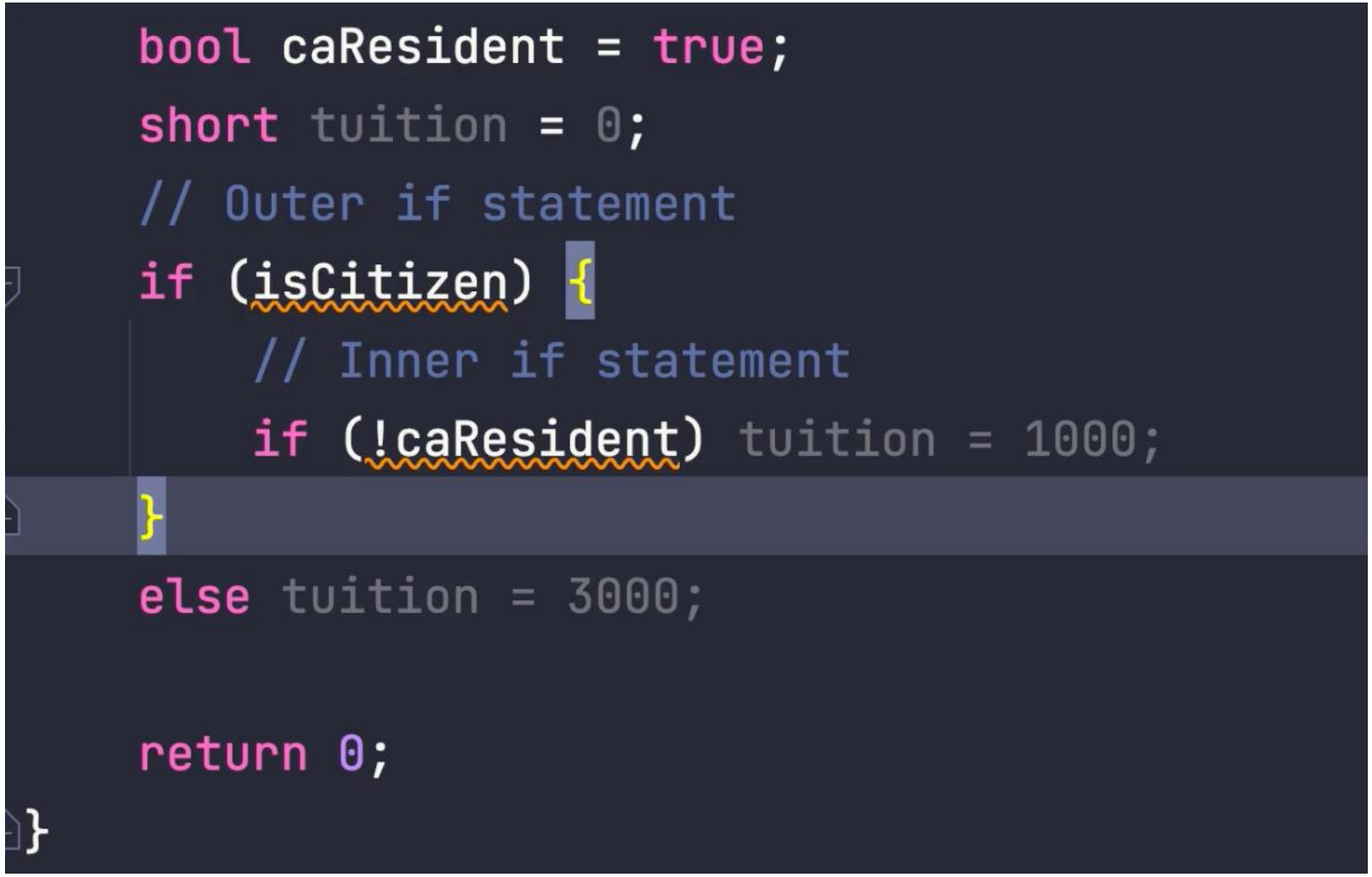
# IF Statement

```
int main() {
    int temperature = 70;
    if (temperature < 60)
        cout << "Cold";
    else if (temperature < 90)
        cout << "Nice";

    return 0;
}
```

- If (condition) {
  statement;
  }
  else if (condition) {
  statements;
  }
  else {
  statement;
  }

- Controls the logic of the program.

# Nested if Statements



# The Conditional Operator

```
int main() {
    int sales = 11'000;
    double commission;
    if (sales > 10'000)
        commission = .1;
    else
        commission = .05;

    return 0;
}
```

- Use conditional variable instead.
- If the condition is satisfied, then the value after ? is assigned, otherwise, the value after : is assigned to the conditional variable.

```
int main() {
    int sales = 11'000;
    double commission = (sales > 10'000) ? .1 : .05;
    cout << commission;

    return 0;
}
```

# The Switch Statement

## Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

# The Switch Statement

```
short input;
cin >> input;

if (input == 1)
    cout << "You selected 1";
else if (input == 2)
    cout << "You selected 2";
else
    cout << "Goodbye!";
```

```
switch (input) {
    case 1:
        cout << "You selected 1";
        break;
    case 2:
        cout << "You selected 2";
        break;
    default:
        cout << "Goodbye!";
}
```

# The Switch Statement

```cpp
int day = 4;
switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
    break;
  case 7:
    cout << "Sunday";
    break;
}
// Outputs "Thursday" (day 4)
```

## Example

From:https://www.w3schools.com/cpp/

# The Switch Statement

## Example

```cpp
using namespace std;

int main() {
    cout << "Enter two numbers: ";
    int first;
    int second;
    cin >> first >> second;

    cout << "Enter an operator: ";
    char op;
    cin >> op;

    switch (op) {
        case '+':
            cout << first + second;
            break;
        case '-':
            cout << first - second;
            break;
        default:
            cout << "Invalid operator!";

    }
```

# Loops in C++

For loops

Range-based for loops

While loops

Do-while loops

Break and continue statements

# The for Loop

```cpp
#include <iostream>

using namespace std;

int main() {
    for (int i = 0; i < 5; i++) {
        cout << i << endl;



    return 0;
```

## Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

# The for Loop

```cpp
int main() {
    for (int i = 5; i > 0; i--) {
        if (i % 2 != 0)
            cout << i << endl;
    }

    return 0;
}
```

Syntax

```cpp
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

# Example

- Odd numbers between 5 and 1
- %: Modulo operator. It calculates the remainder of a division between two integers.

# Compound Assignment Statement

C++

```
variable operator= expression;
```

This is equivalent to:

C++

```
variable = variable operator expression;
```

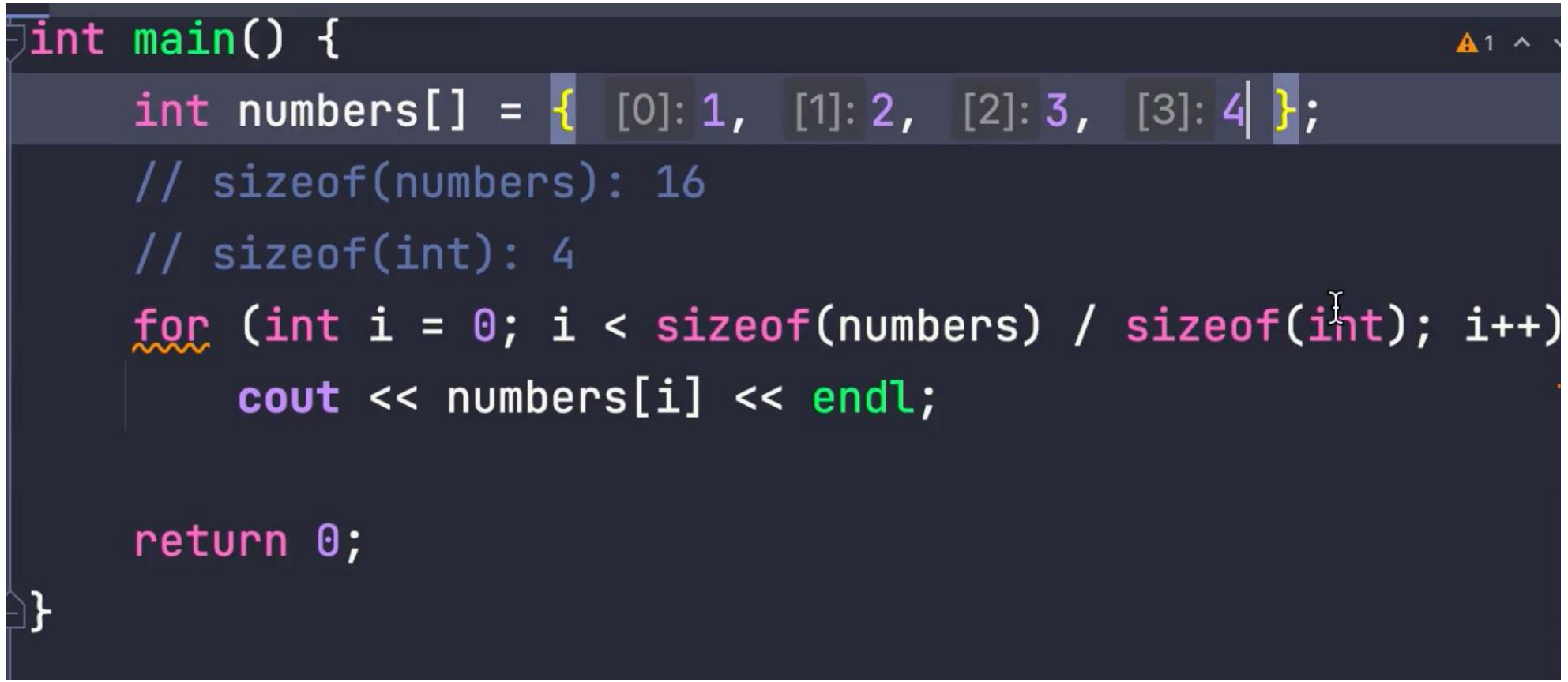
## Example

**x *= y**

**is equivalent to**

**x = x * y;**

# Range-based for Loops



# Range-based for Loops

```cpp
int main() {
    int numbers[] = { [0]: 1, [1]: 2, [2]: 3 };
    // sizeof(numbers): 16
    // sizeof(int): 4
    for (int i = 0; i < sizeof(numbers) / sizeof(int); i++)
        cout << numbers[i] << endl;

    for (int number: numbers)
        cout << number << endl;

    return 0;
```

# Range-based for Loops

```
int main() {
    int temperatures[] = { [0]: 60, [1]: 80, [2]: 90 };
    double sum = 0;
    for (int temperature: temperatures)
        sum += temperature;
    short count = sizeof(temperatures) / sizeof(int);
    double average = sum / count;
    cout << average;

    return 0;
}
```

## Example

Average of temperatures

# While Loop

```cpp
int main() {
    for (int i = 1; i <= 5; i++)
        cout << i << endl;

    int i = 1;
    while (i <= 5) {
        cout << i << endl;
        i++;
    }
```

## Syntax

```cpp
while (condition) {
  // code block to be executed
}
```

- We cannot declare a variable in the while loop unlike the for loop.
- Useful when we do not know how many times we need to loop ahead of time.

# While Loop

```
main() {
int number = 0;
while (number < 1 || number > 5) {
    cout << "Number: ";
    cin >> number;
    if (number < 1 || number > 5)
        cout << "Enter a number between 1 and 5!" << endl;
}
```

# Do-while Loop

```cpp
int main() {
    int number;
    do {
        cout << "Number: ";
        cin >> number;
    } while (number < 1 || number > 5);

    return 0;
}
```

## Syntax

```
do {
  // code block to be executed
}
while (condition);
```

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

# Break and Continue Statements

```cpp
#include <iostream>

using namespace std;

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i % 3 != 0)
            cout << i << endl;
    }

    return 0;
```

break: to break out of a loop

continue: to skip an iteration

```cpp
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i % 3 == 0)
            continue;
        cout << i << endl;
    }

    return 0;
}
```

# Break and Continue Statements

break: to break out of a loop

continue: to skip an iteration

```
while (true) {
    cout << "Number: ";
    int number;
    cin >> number;
    if (number >= 1 && number <= 5)
        break;
    cout << "Error! Enter a number between 1 and 5." <<
}

return 0;
```

# Nested Loops

```cpp
int main() {
    for (int x = 1; x <= 5; x++) {
        for (int y = 1; y <= 5; y++)
            cout << "(" << x << ", " << y << ")" << endl;
    }

    return 0;
}
```

# Nested Loops

```cpp
int main() {
    cout << "Rows: ";
    int rows;
    cin >> rows;

    for (int i = 1; i <= rows; i++) {
        for (int j = 0; j < i; j++)
            cout << "*";
        cout << endl;
    }
}
```

# Example

```
Rows: 5
*
**
***
****
*****
```

# Functions

Define and call functions

Assign parameters a default value

Overload functions

Pass arguments by value or reference

Local vs global variables

Organize functions in different files

```cpp
#include <iostream>

using namespace std;

void greet() {
    cout << "Hello World" << endl;
}

int main() {
    // Calling - invoking - executing
    greet();

    cout << "Done";

    return 0;
}
```

# Functions

- ❑ No parameters and no returns.
- ❑ Use **void** (No returns)

# Functions

- With parameters and return.

```cpp
#include <iostream>

using namespace std;

void greet(string firstName, string lastName) {
    cout << "Hello " << firstName << " " << lastName << endl;
}

string fullName(string firstName, string lastName) {
    // Concatenating (combining)
    return firstName + " " + lastName;
}

int main() {
    // Calling - invoking - executing
    string name = fullName(firstName: "Mosh", lastName: "Hameda
    greet(name);

    cout << "Done";

    return 0;
}
```

parameters

arguments

```cpp
void greet(string name) {
    cout << "Hello " << name << endl;
}
```

```cpp
int main() {
    // Calling - invoking - executing
    greet(name: fullName(firstName: "Mosh", lastName: "Hamedani
    cout << "Done";

    return 0;
}
```

# Functions

## Example

A function that receives two integers and returns the maximum of the two.

```cpp
#include <iostream>

using namespace std;

int max(int first, int second) {
    if (first > second)
        return first;
    else
        return second;
}
```

Version 1

```cpp
#include <iostream>

using namespace std;

int max(int first, int second) {
    if (first > second)
        return first;

    return second;
}
```

Version 2

```cpp
int max(int first, int second) {
    return (first > second) ? first : second;
}
```

Version 3

# Functions

```cpp
#include <iostream>

using namespace std;

int max(int first, int second) {
    return (first > second) ? first : second;
}

int main() {
    int larger = max( first: 1,  second: 2);
    cout << larger

    return 0;
}
```

## Example

A function that receives two integers and returns the maximum of the two.

# Parameters with a Default Value

```cpp
#include <iostream>

using namespace std;

double calculateTax(double income, double taxRate = .2) {
    return income * taxRate;
}

int main() {
    double tax = calculateTax( income: 10'000);
    cout << tax;

    return 0;
}
```

- If an argument is not provided for a parameter with a default value, then the default value is used.
- **NOTE:** Argument(s) with default value should be defined after the argument(s) without default value. Otherwise, a compiler error will occur.

# Overloading Functions


- ❑ Each function should have a unique signature.

# Passing Arguments by Value or Reference

```cpp
#include <iostream>

using namespace std;

void increasePrice(double price) {
    price *= 1.2;
}
```

Same

```cpp
void increasePrice(double price) {
    double price;
    price *= 1.2;
}
```

- The parameter variable “price” is local to the function. (Its scope)

# Passing Arguments by Value or Reference

```cpp
#include <iostream>

using namespace std;

void increasePrice(double price) {
    double price;
    price *= 1.2;
}

int main() {
    double price = 100;
    increasePrice(price);
    cout << price;
```

- The parameter variable "price" is local to the function. (Its scope)

The scope of the variable price is different than the scope of argument price.

# Passing Arguments by Value or Reference

```cpp
#include <iostream>

using namespace std;

double increasePrice(double price) {
    price *= 1.2;
    return price;
}

int main() {
    double price = 100;
    price = increasePrice(price);
    cout << price;
```

- The parameter variable “price” is local to the function. (Its scope).
- By default, arguments are passed-by-value in C++
- The value is copied to the local variable in the function.

**Solution 1:**

- Return the value from the function.
- Use pass by value.

# Passing Arguments by Value or Reference

- Pass by reference is appropriate when the argument data is large.

```cpp
#include <iostream>

using namespace std;

void increasePrice(double& price) {
    price *= 1.2;
}

int main() {
    double price = 100;
    increasePrice( &: price);
    cout << price;

    return 0;
}
```

Reference parameter

```cpp
double increasePrice(double price) {
    price *= 1.2;
    return price;
}

int main() {
    double price = 100;
    price = increasePrice(price);
    cout << price;
}
```

**Solution 2:**

- Use pass by reference.

# Passing Arguments by Value or Reference

```cpp
#include <iostream>

using namespace std;

void greet(string& name) {
    cout << "Hello " << name << endl;
    name = "a";
}

int main() {
    string name = "Mosh";
    greet( &: name);
    cout << name;

    return 0;
```

- Pass-by-reference is appropriate when the argument data is large.

# Local versus Global Variables

```cpp
#include <iostream>

using namespace std;

// Global variable (global scope)
const double taxRate = .2;

double calculateTax(int sales) {
    return sales * taxRate;
}
```

- **Constant variables** that are used by multiple functions should be declared as **global variables**.

```cpp
int main() {
    // Local variable (local scope)
    int sales = 10'000;
    double tax = calculateTax(sales);
    cout << tax;

    return 0;
```

- Variables are accessible withing the block that they are declared.

# Declaring Functions

```cpp
#include <iostream>

using namespace std;

int main() {
    greet("Mosh");
    return 0;
}

void greet(string name) {
    cout << "Hello " << name;
}
```

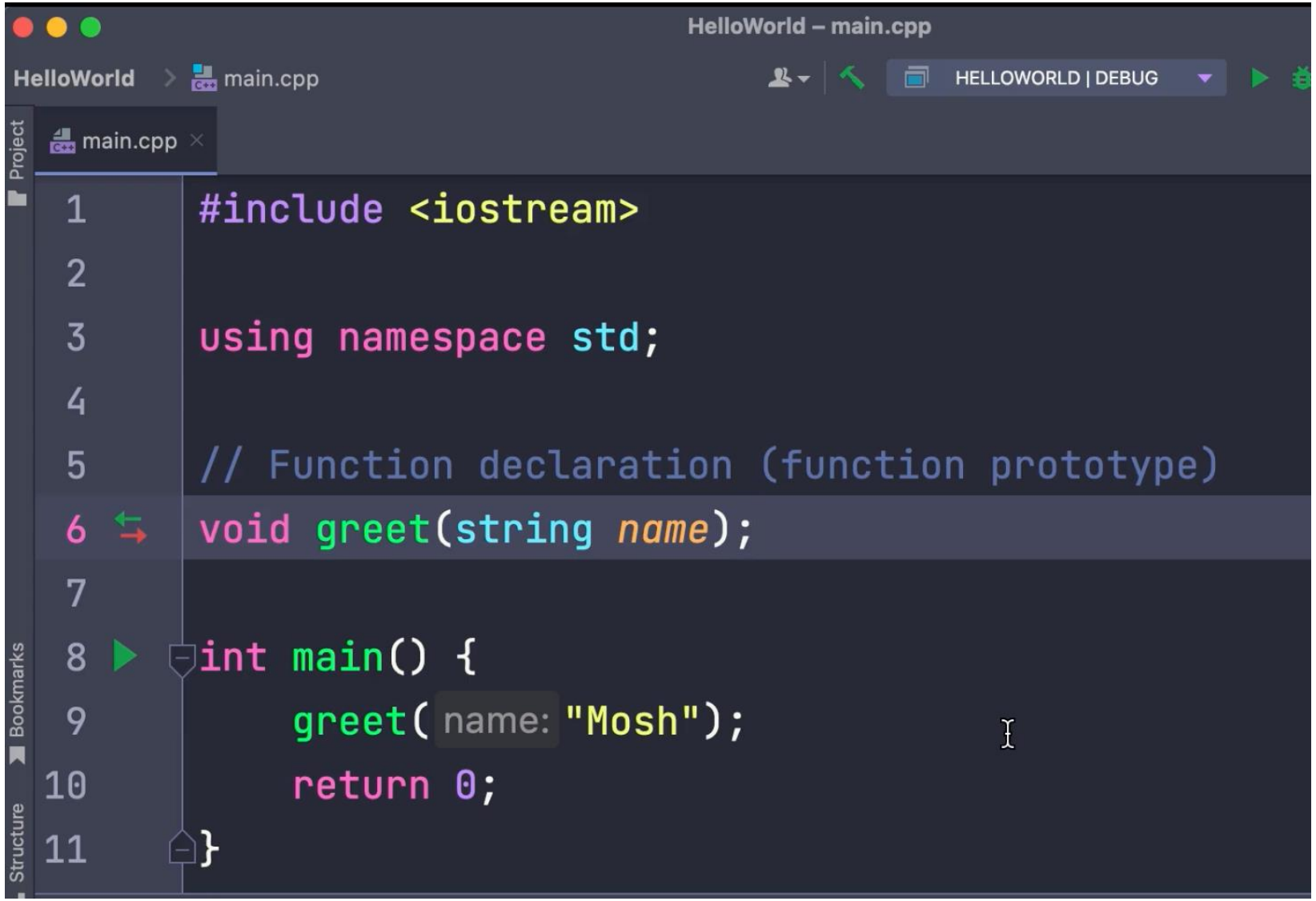
- By default, in C++, functions should be **defined** before the main function.

# Declaring Functions

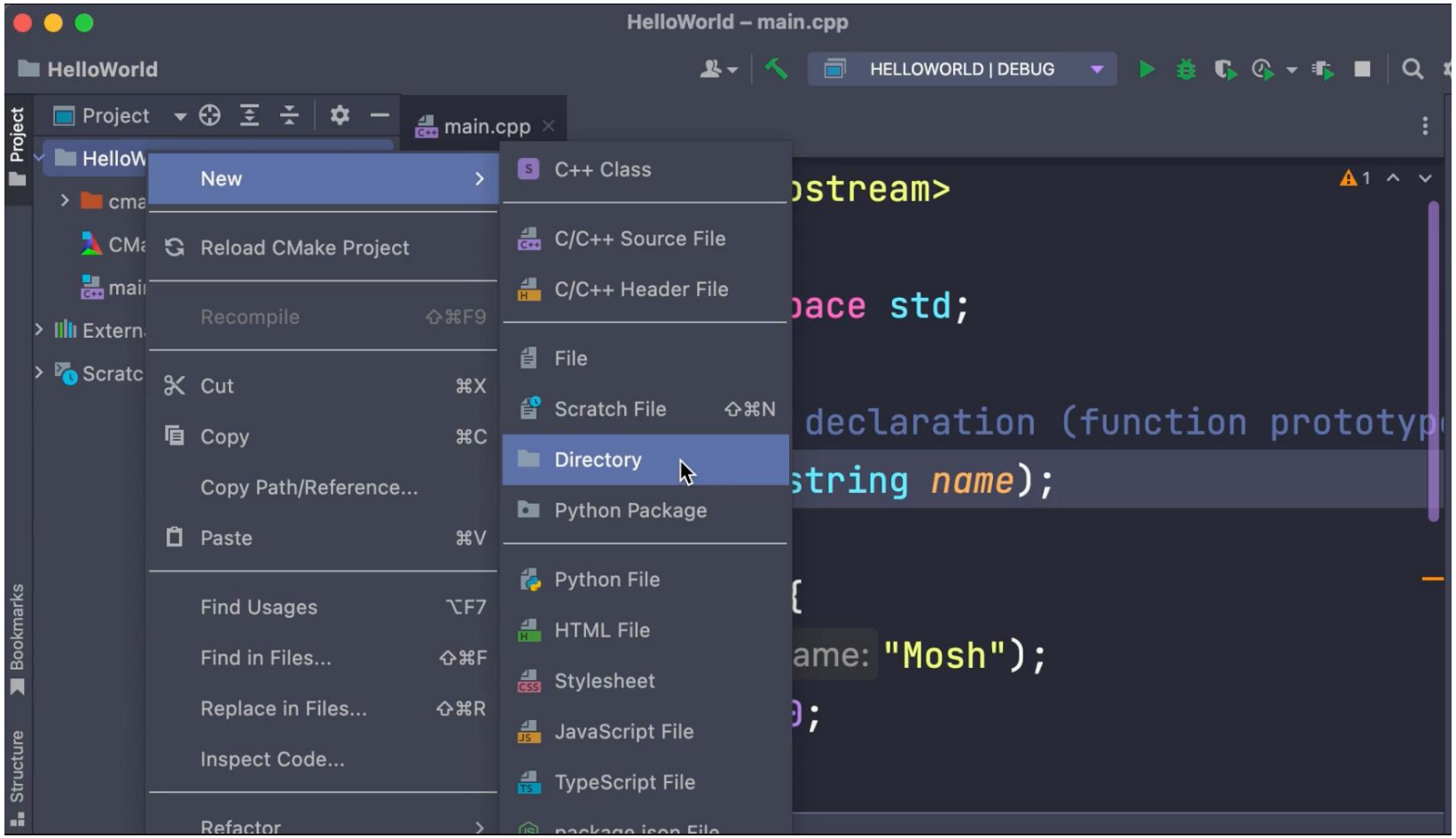
```
// Function declaration (function prototype)
void greet(string name);

int main() {
    greet( name: "Mosh");
    return 0;
}

// Function definition
void greet(string name) {
    cout << "Hello " << name;
}
```

- If we want to **define** a function after the main function, then we should **declare** the function first.

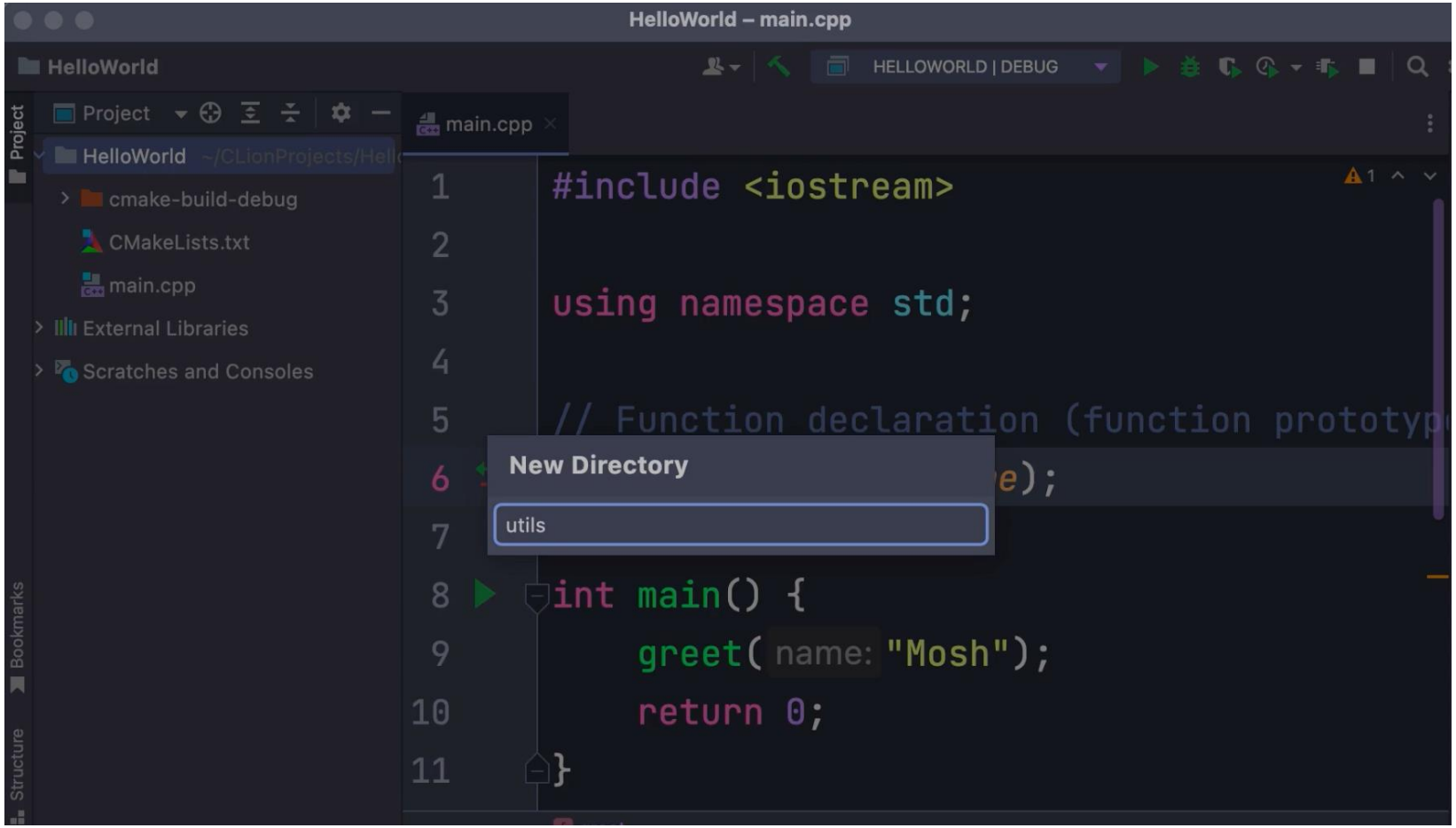
# Organizing Functions in Files



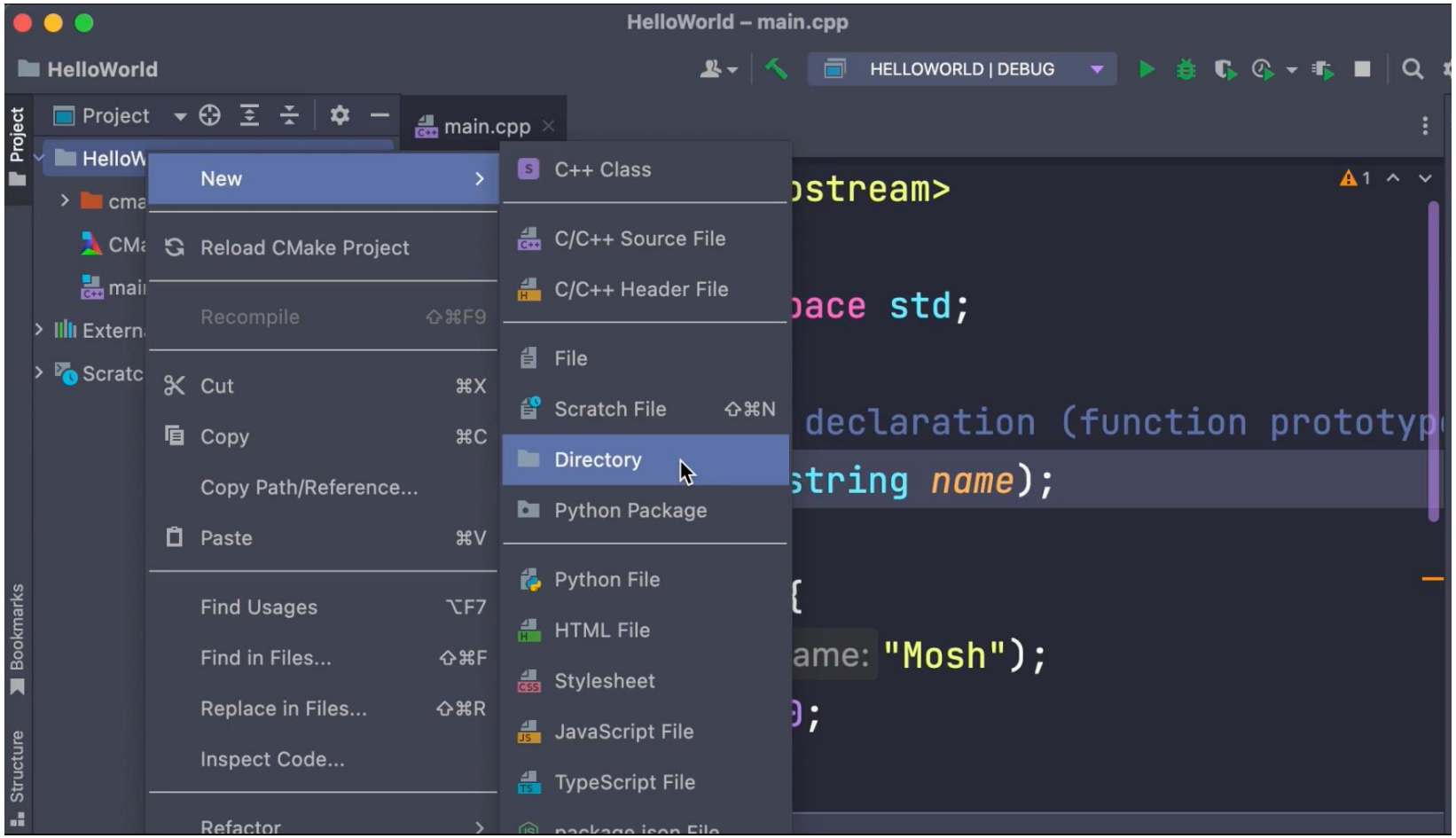
# Organizing Functions in Files



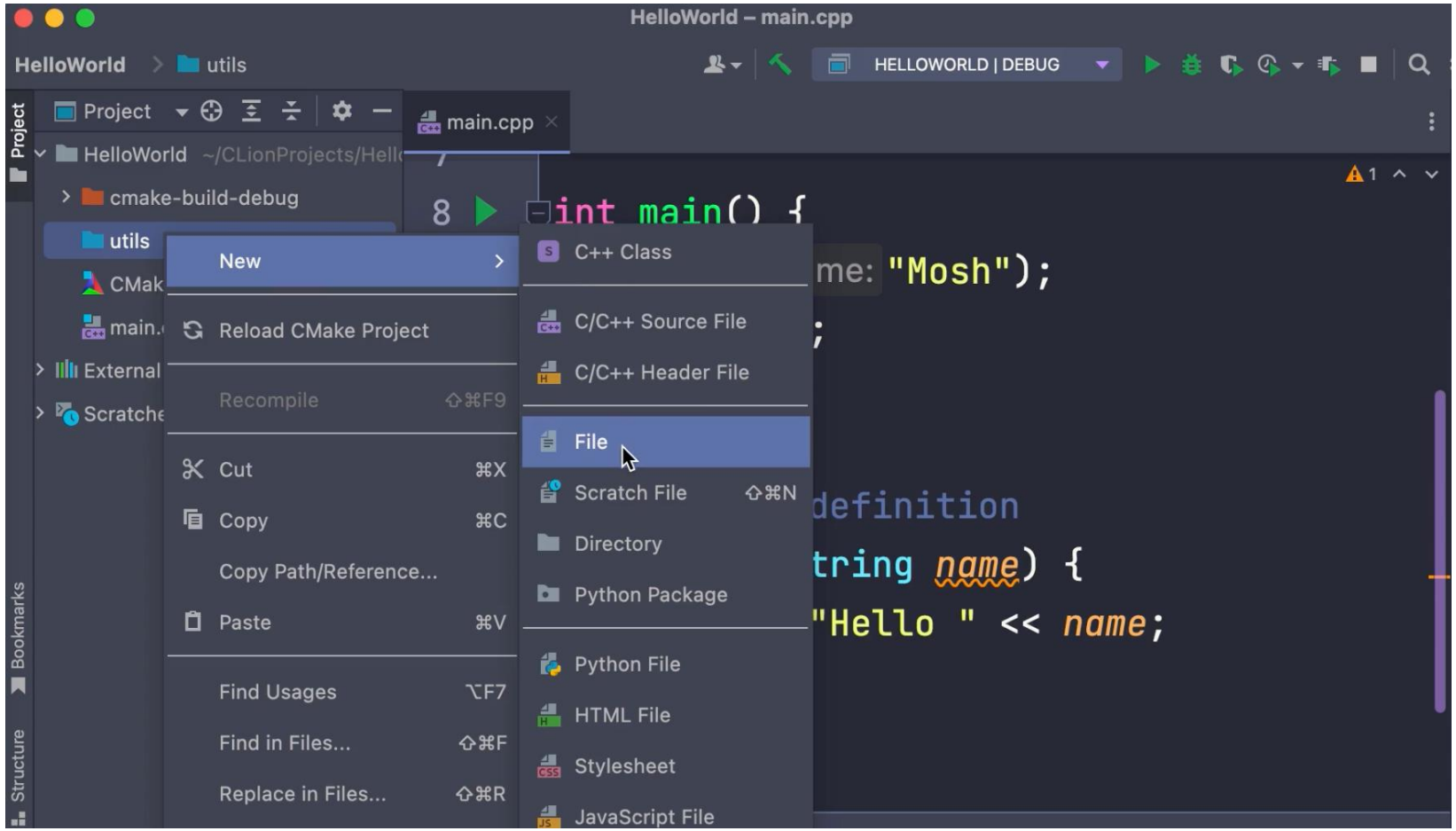
# Organizing Functions in Files



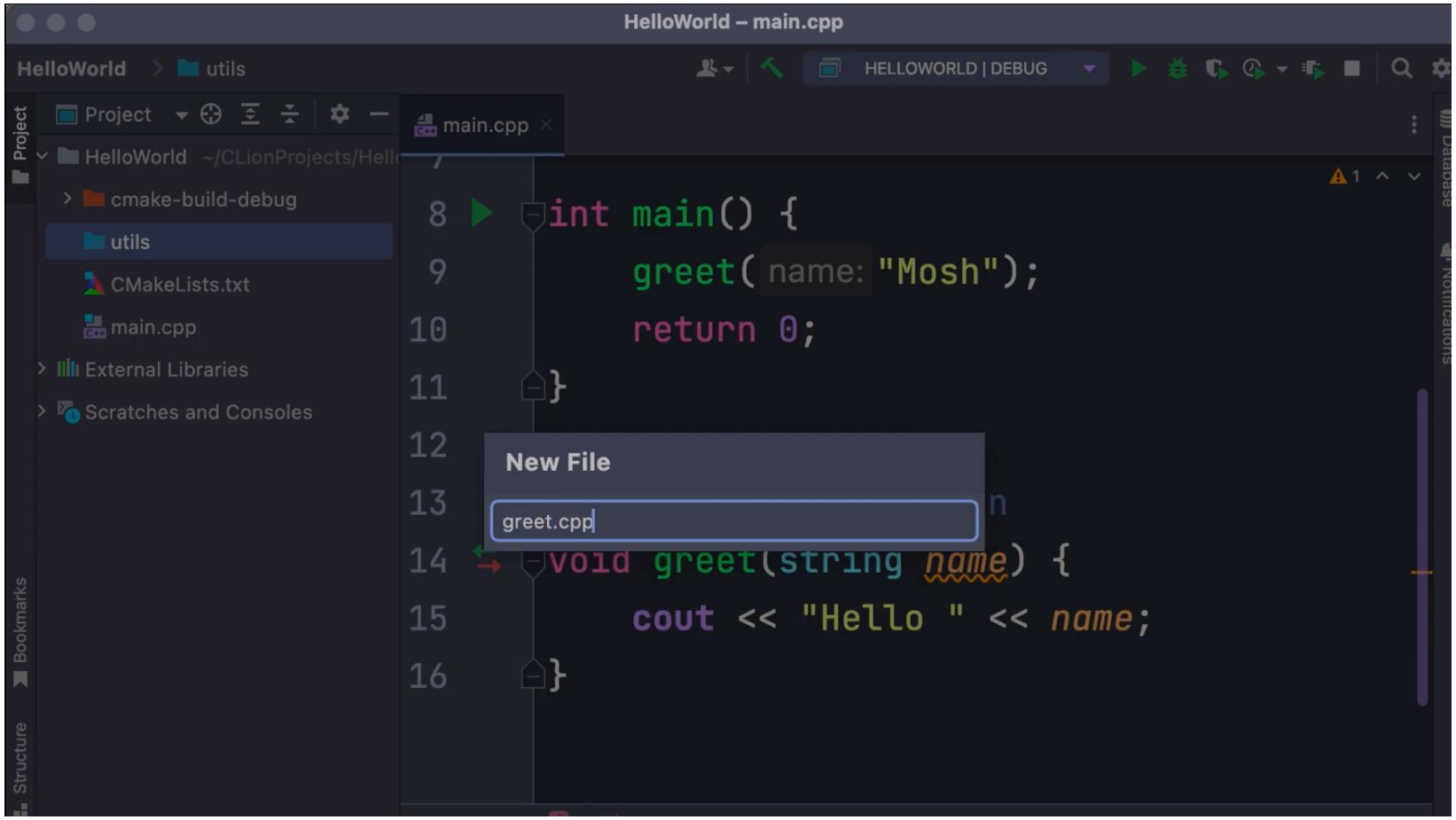
# Organizing Functions in Files



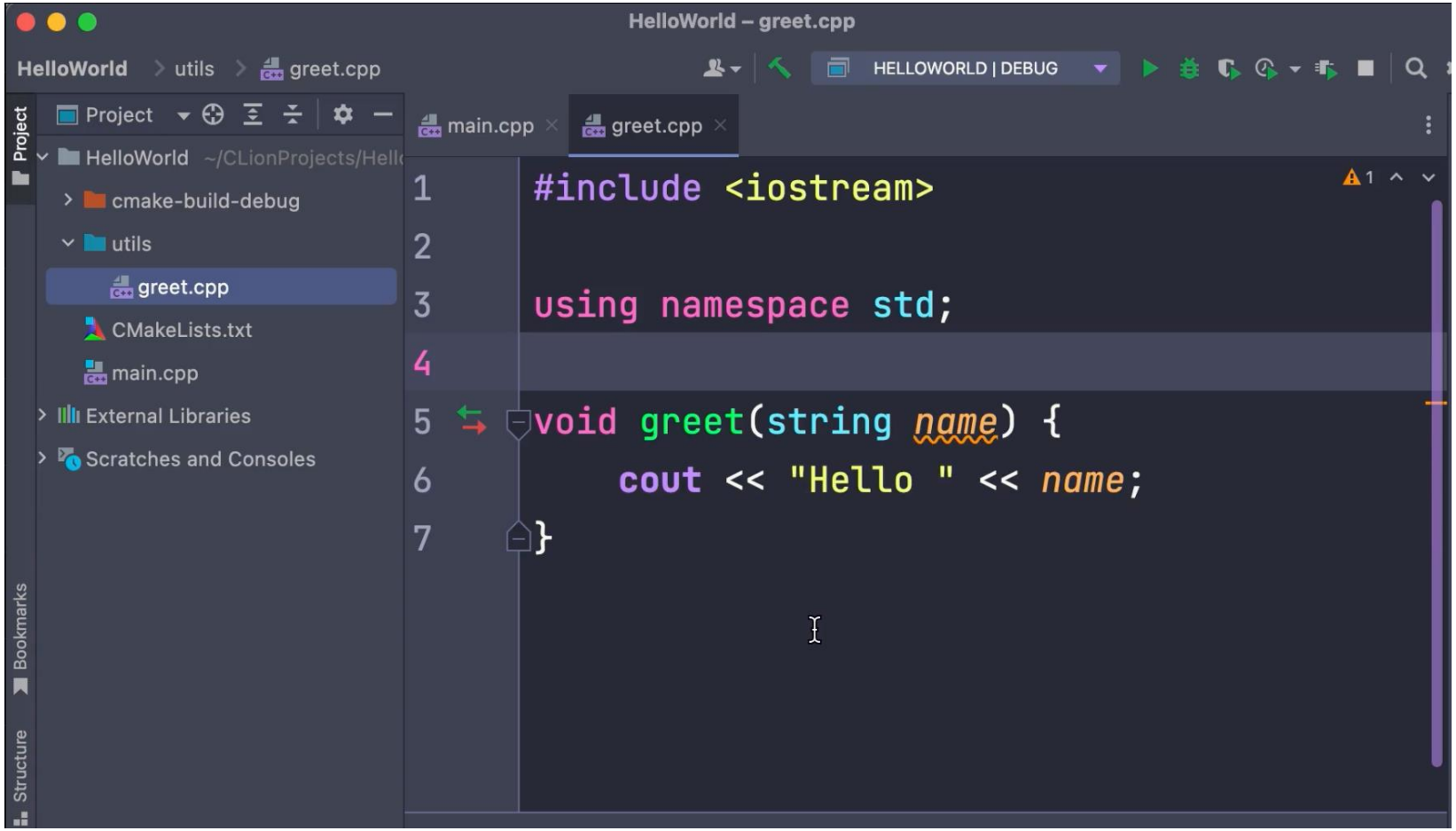
# Organizing Functions in Files



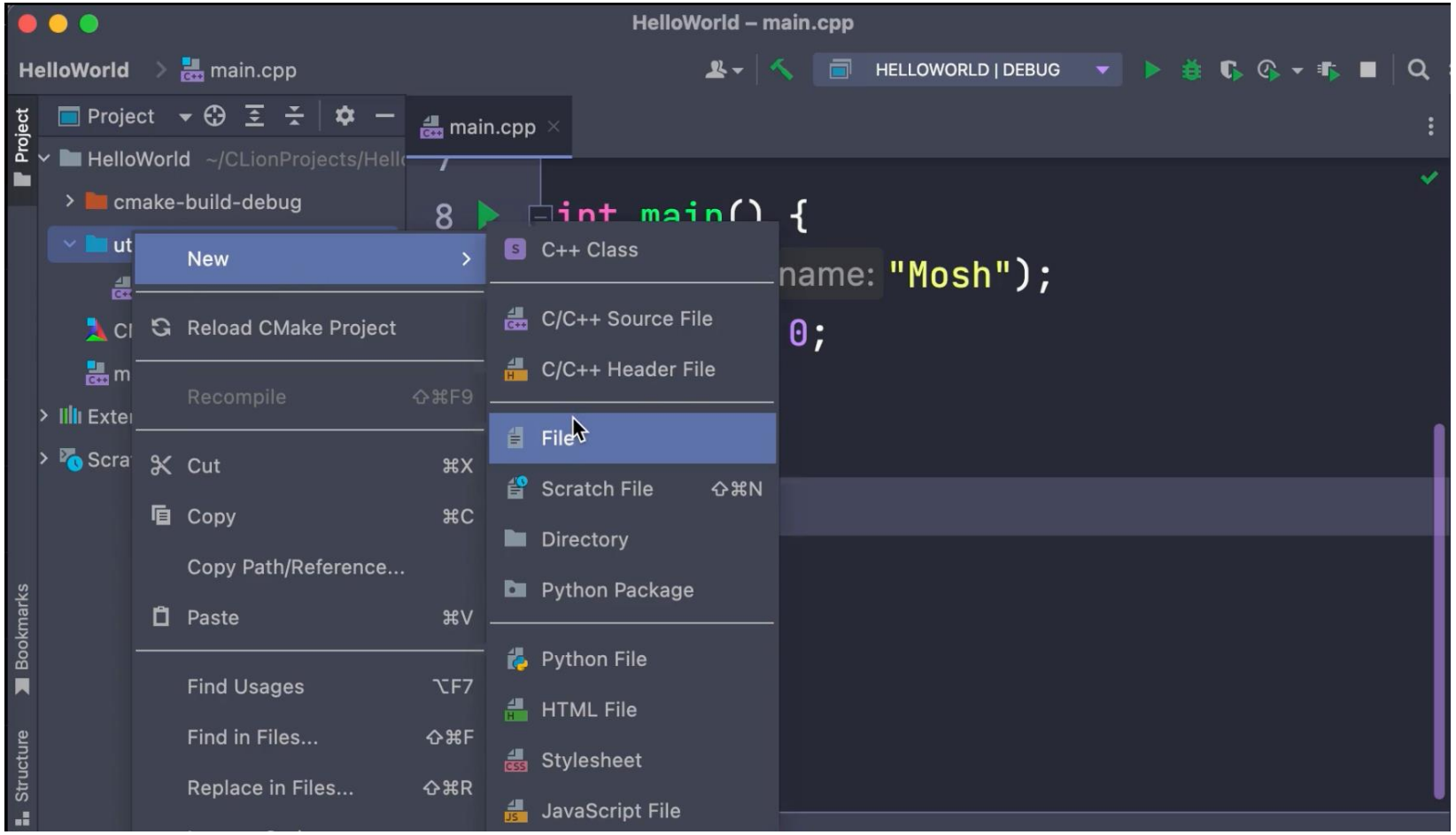
# Organizing Functions in Files



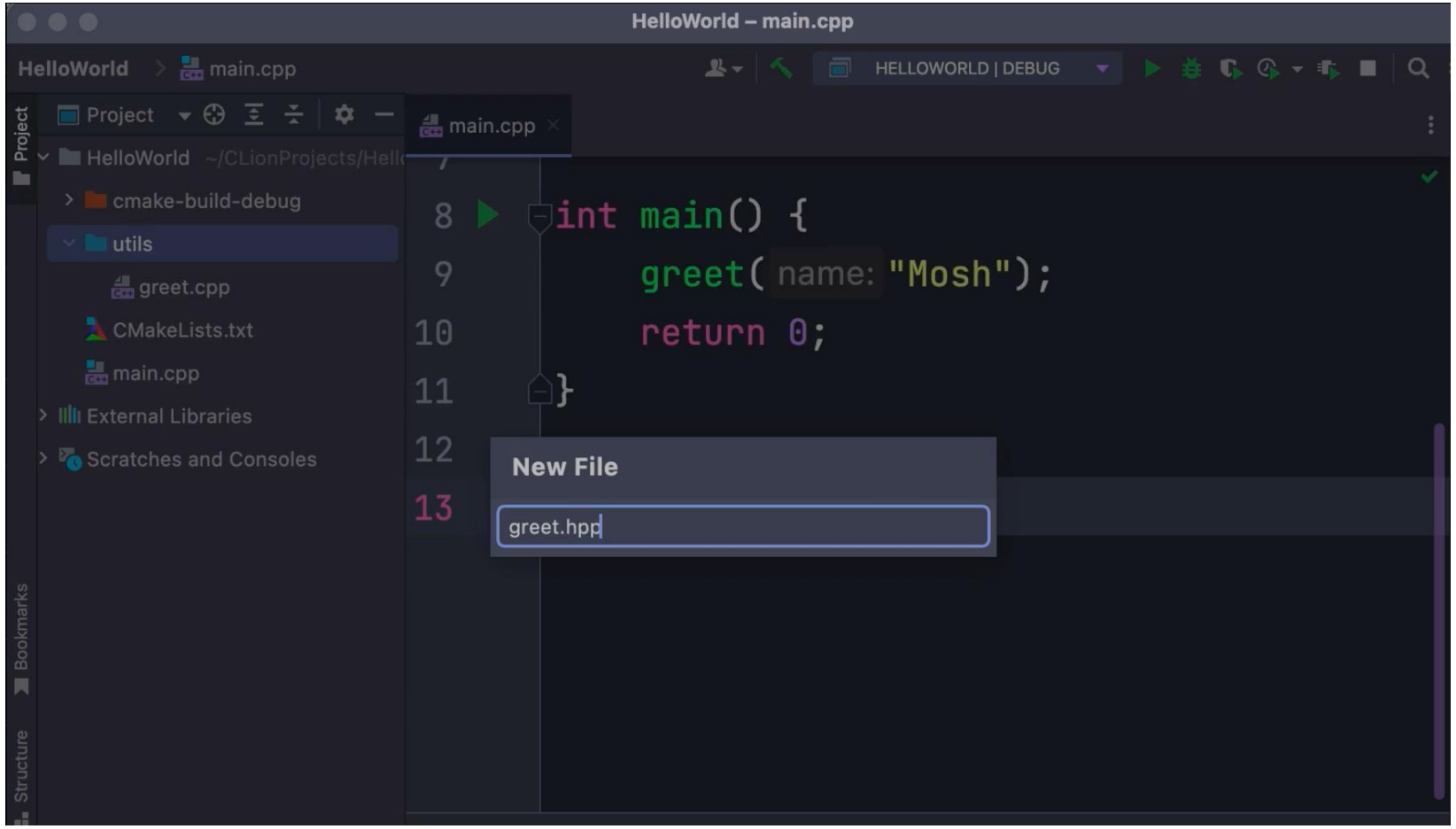
# Organizing Functions in Files



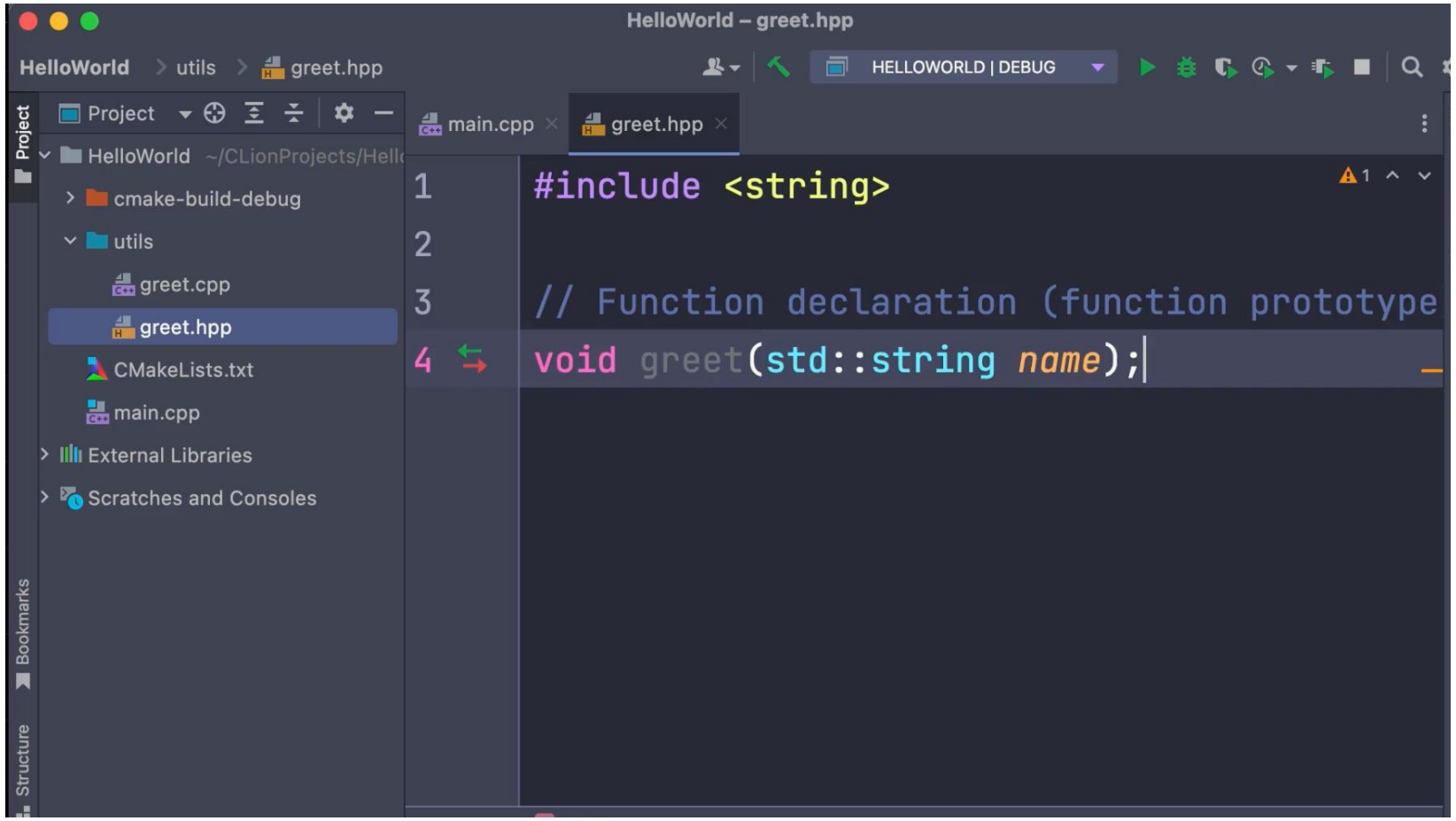
# Organizing Functions in Files



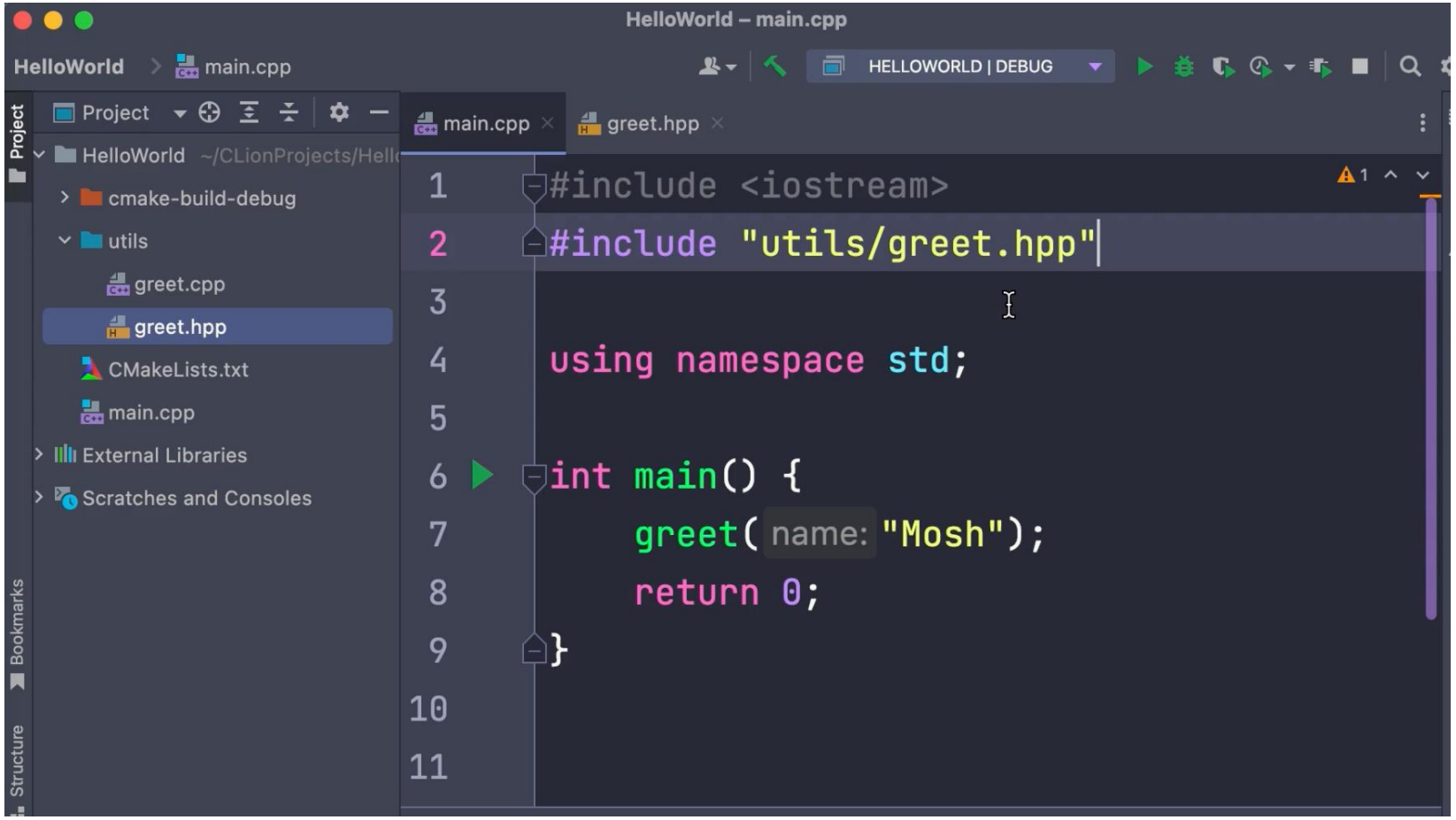
# Organizing Functions in Files



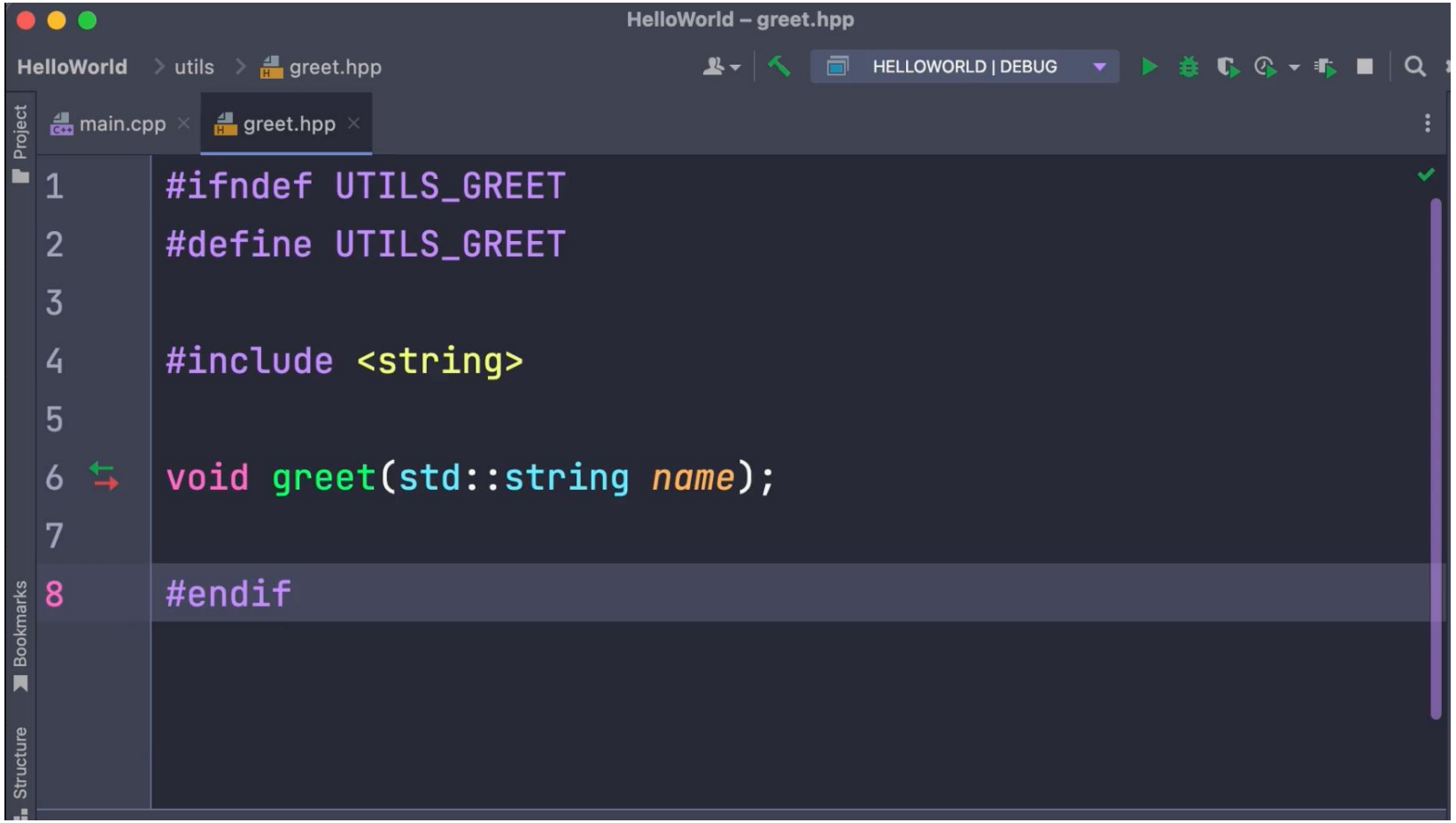
# Organizing Functions in Files



# Organizing Functions in Files

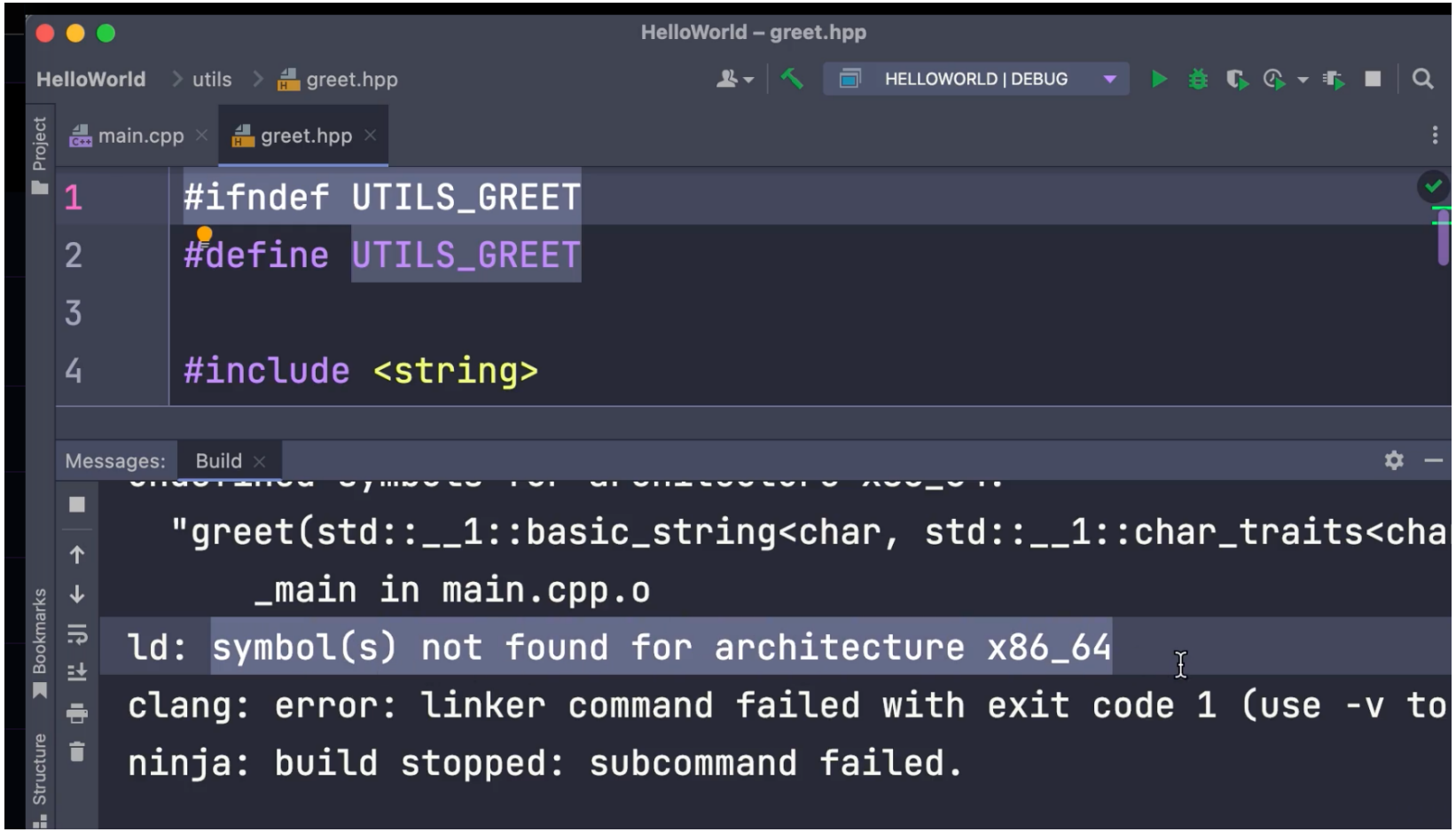


# Organizing Functions in Files



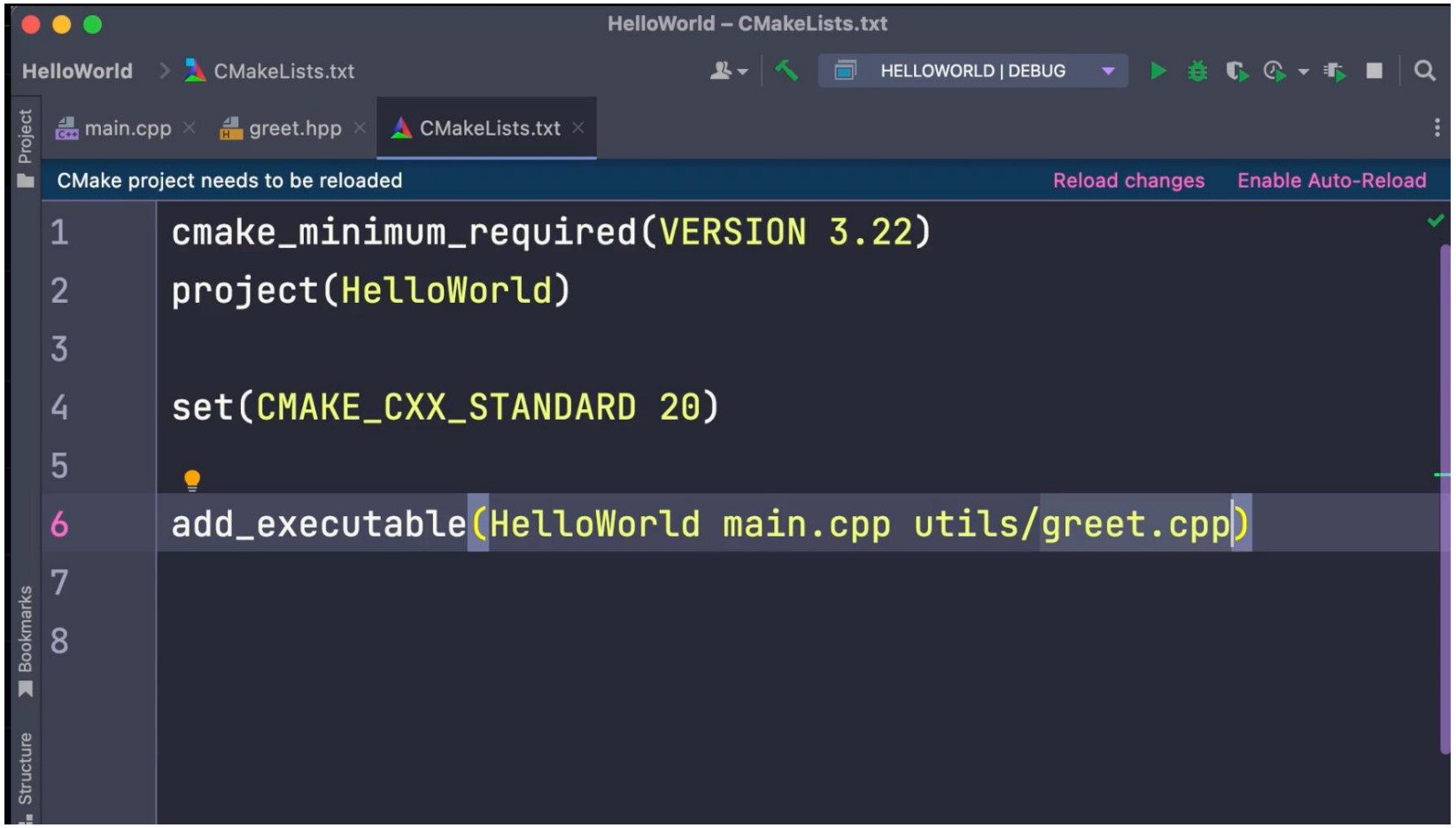

- Update the greet.hpp header file so that it will not be build multiple times by the compiler if it is included in multiple files.

# Organizing Functions in Files



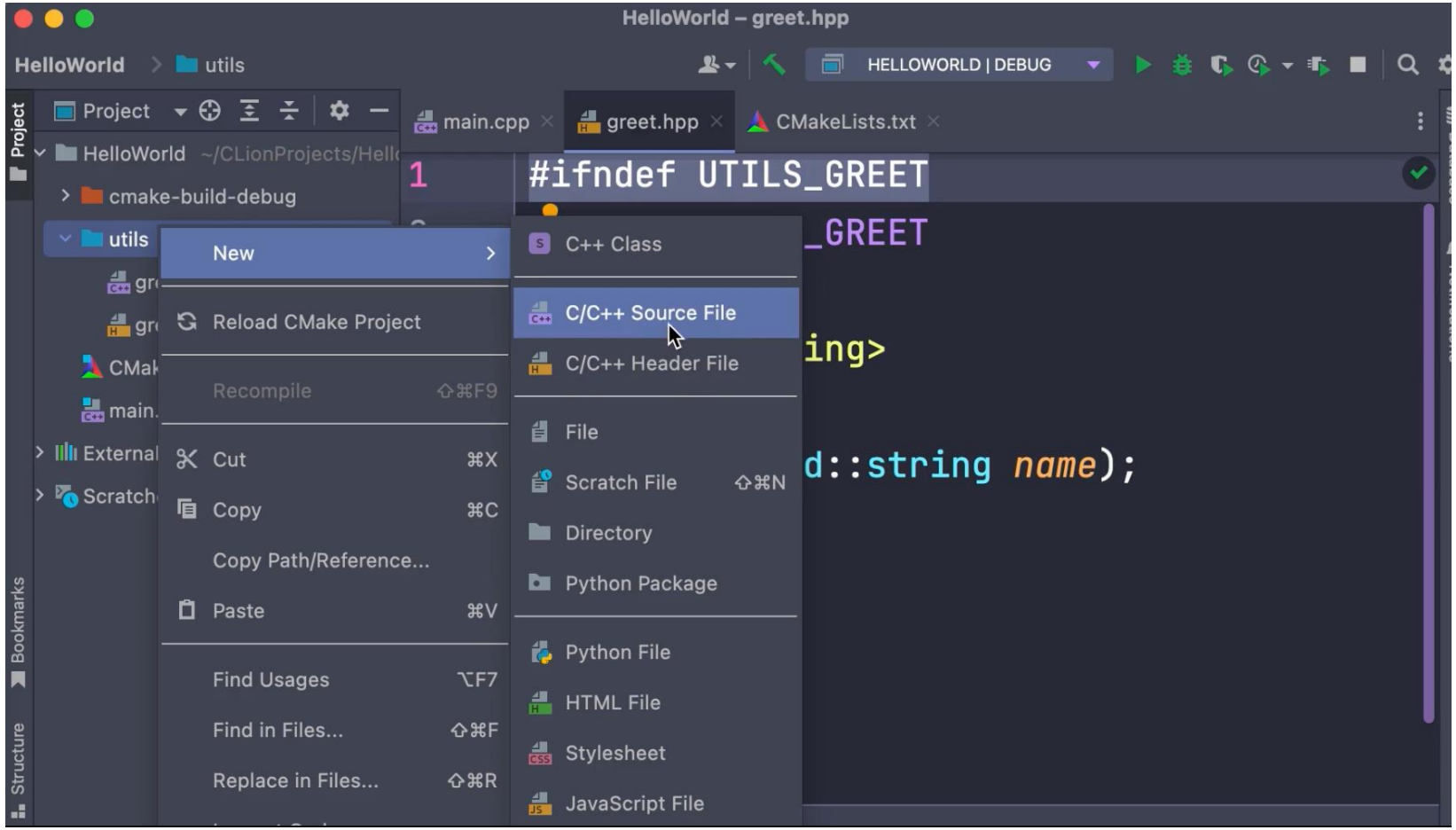

- ❑ When we run the project, we will get this error message!
- ❑ One more thing to do!

# Organizing Functions in Files



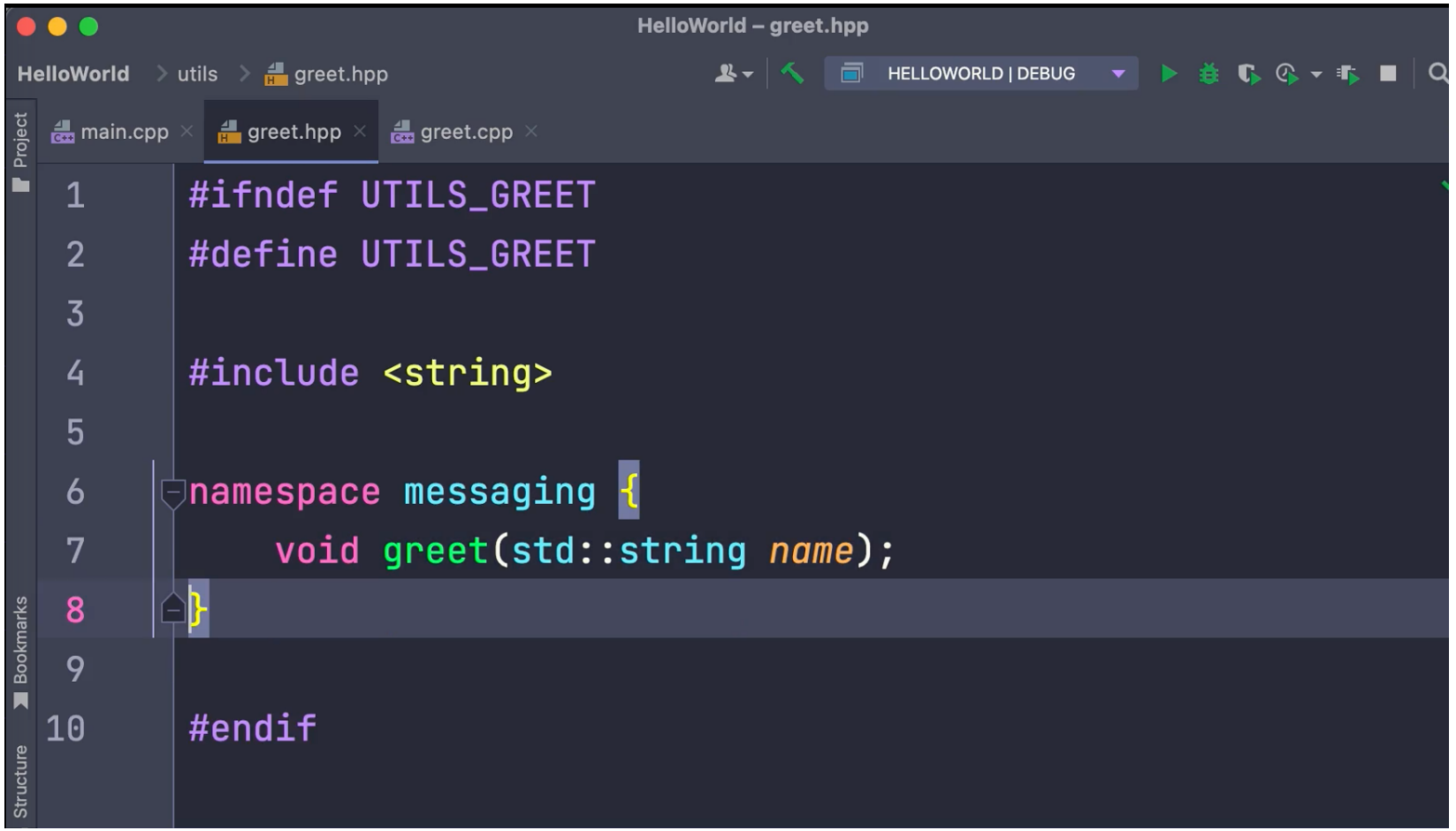

❑ Update the CMakeLists.txt file to include the greet.cpp file.

# Organizing Functions in Files



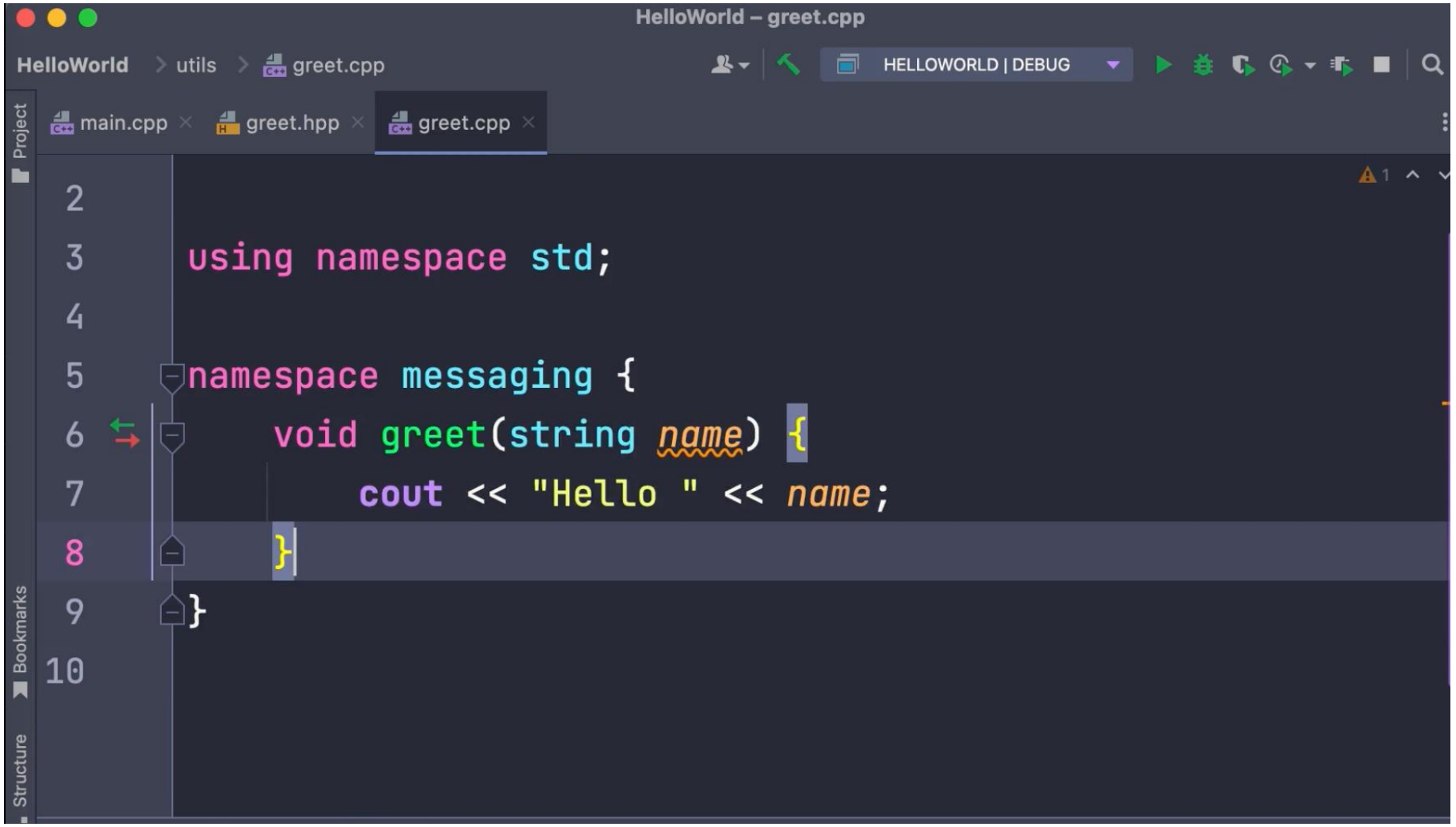

- Shortcut in CLion.

# Using Namespaces

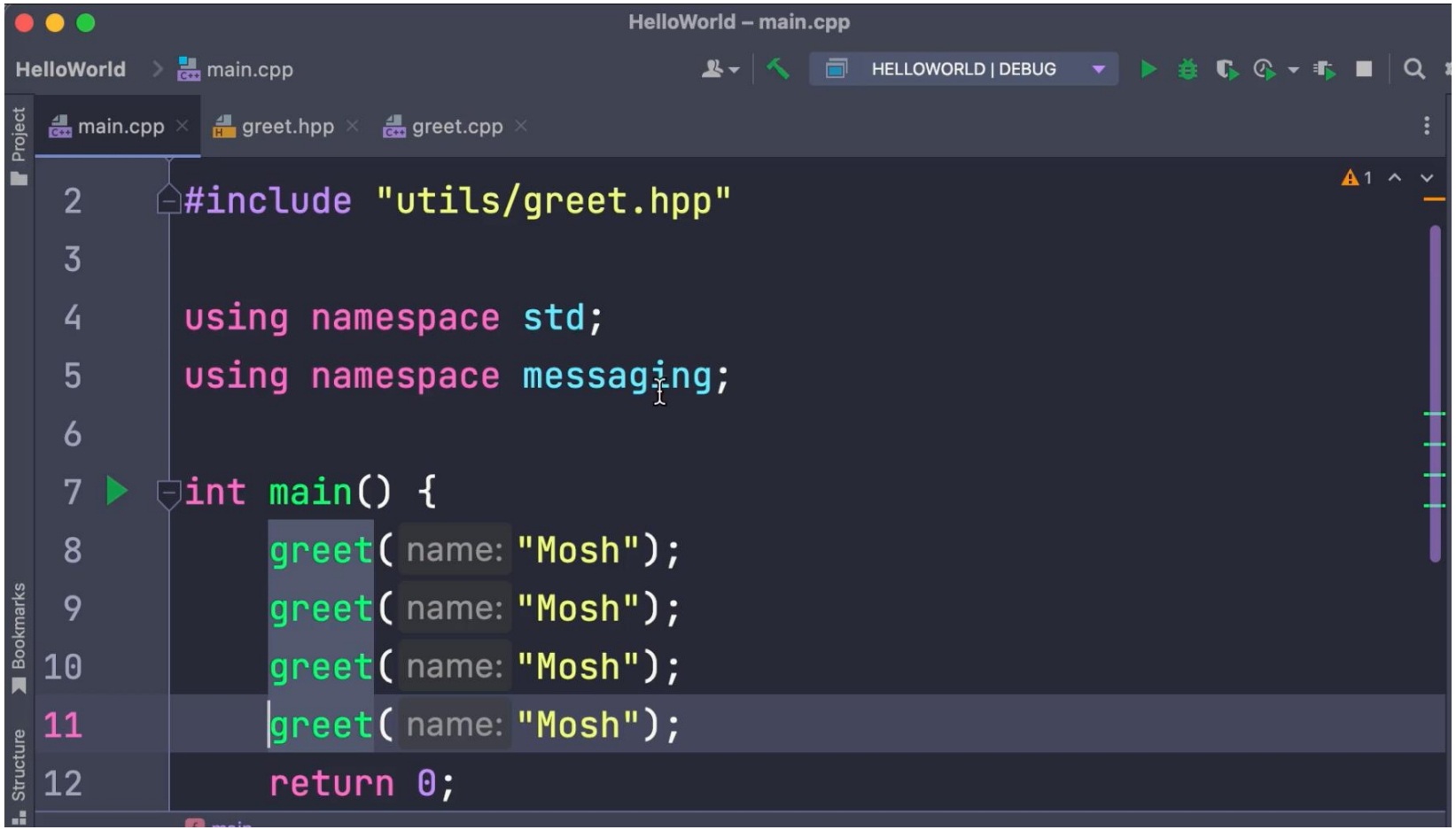



- Namespaces are used to prevent conflict of having functions with the same name developed multiple times.
- It is like a bucket where we put all our functions in.

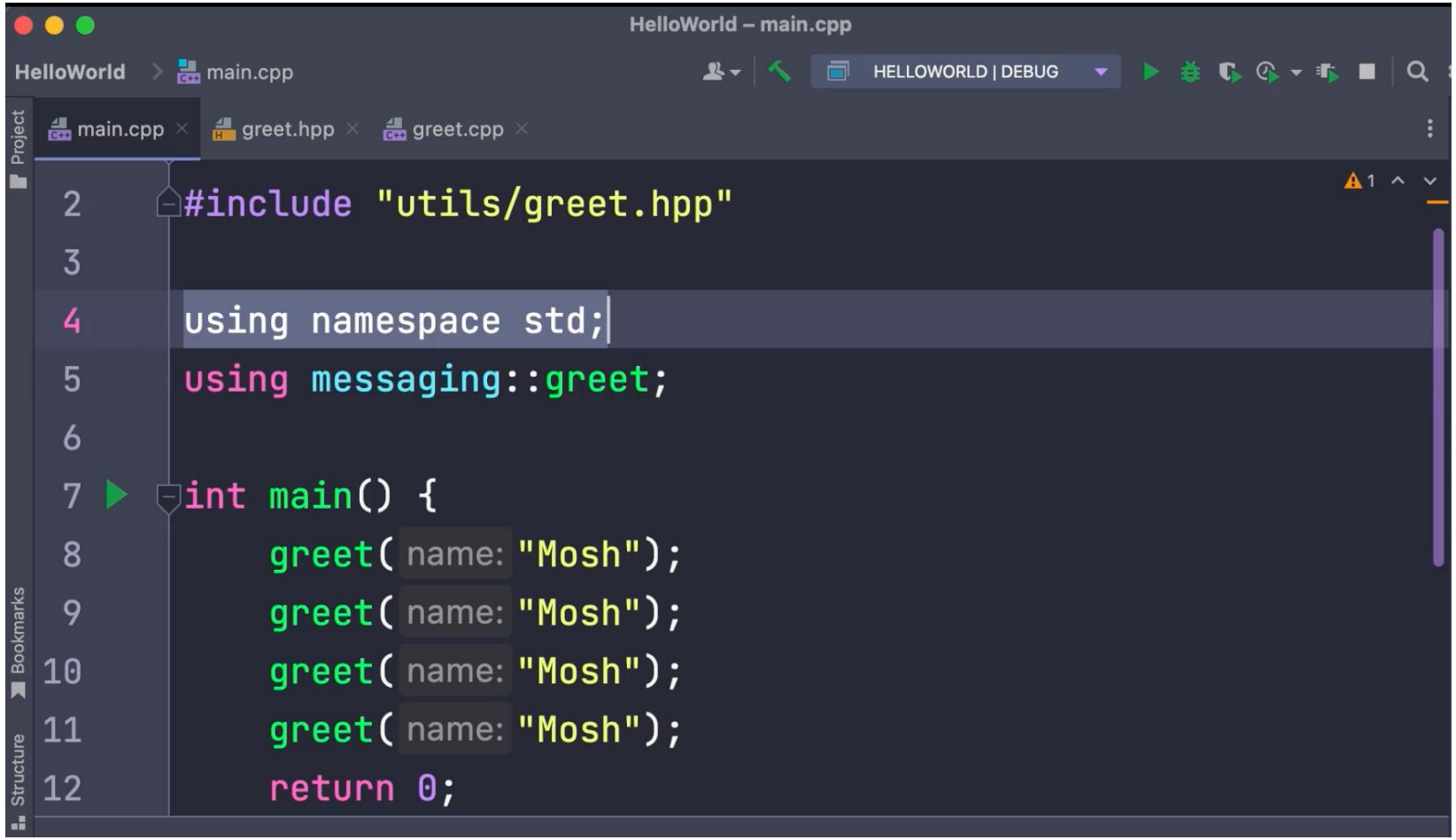
# Using Namespaces



# Using Namespaces



# Using Namespaces




- Include only greet function from the namespace instead of the entire messaging namespace functions.

# Debugging

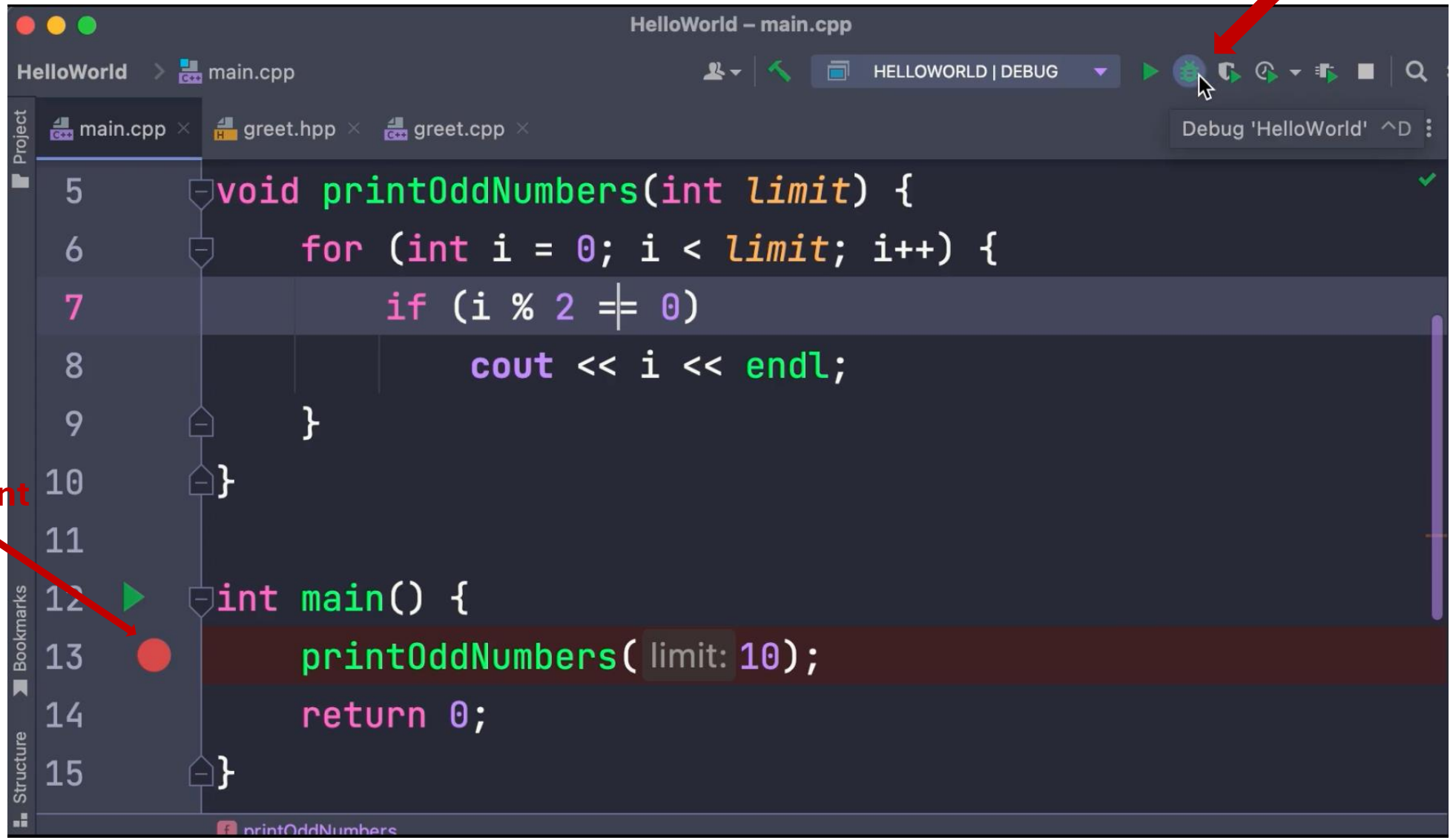
```cpp
#include <iostream>

using namespace std;

void printOddNumbers(int limit) {
    for (int i = 0; i < limit; i++) {
        if (i % 2 != 0)
            cout << i << endl;
    }
}

int main() {
    printOddNumbers( limit: 10);
    return 0;
}
```

Change to

```cpp
void printOddNumbers(int limit) {
    for (int i = 0; i < limit; i++) {
        if (i % 2 == 0)
            cout << i << endl;
    }
}

int main() {
    printOddNumbers( limit: 10);
    return 0;
}
```

# Debugging

Debug button

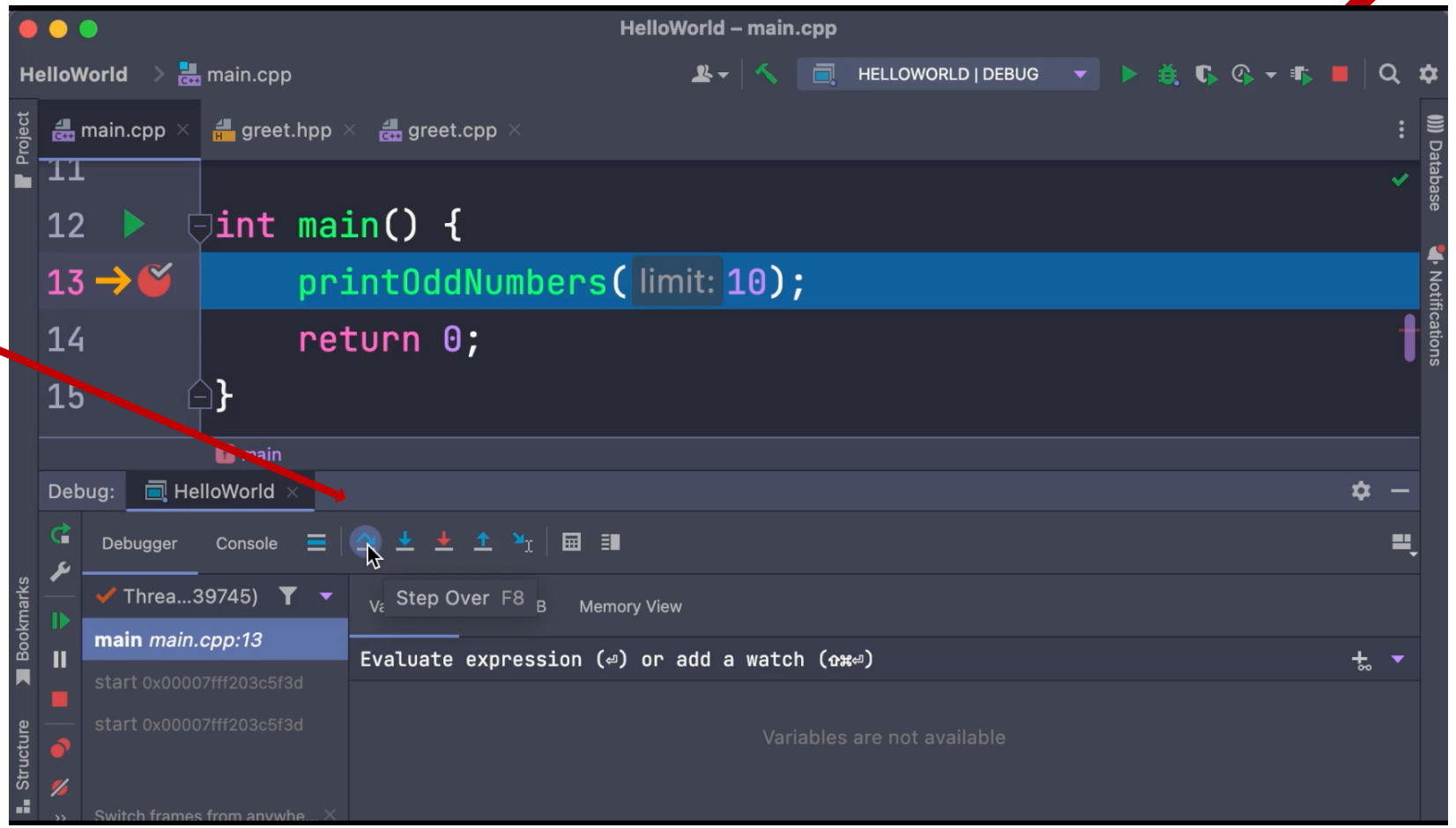



Break point

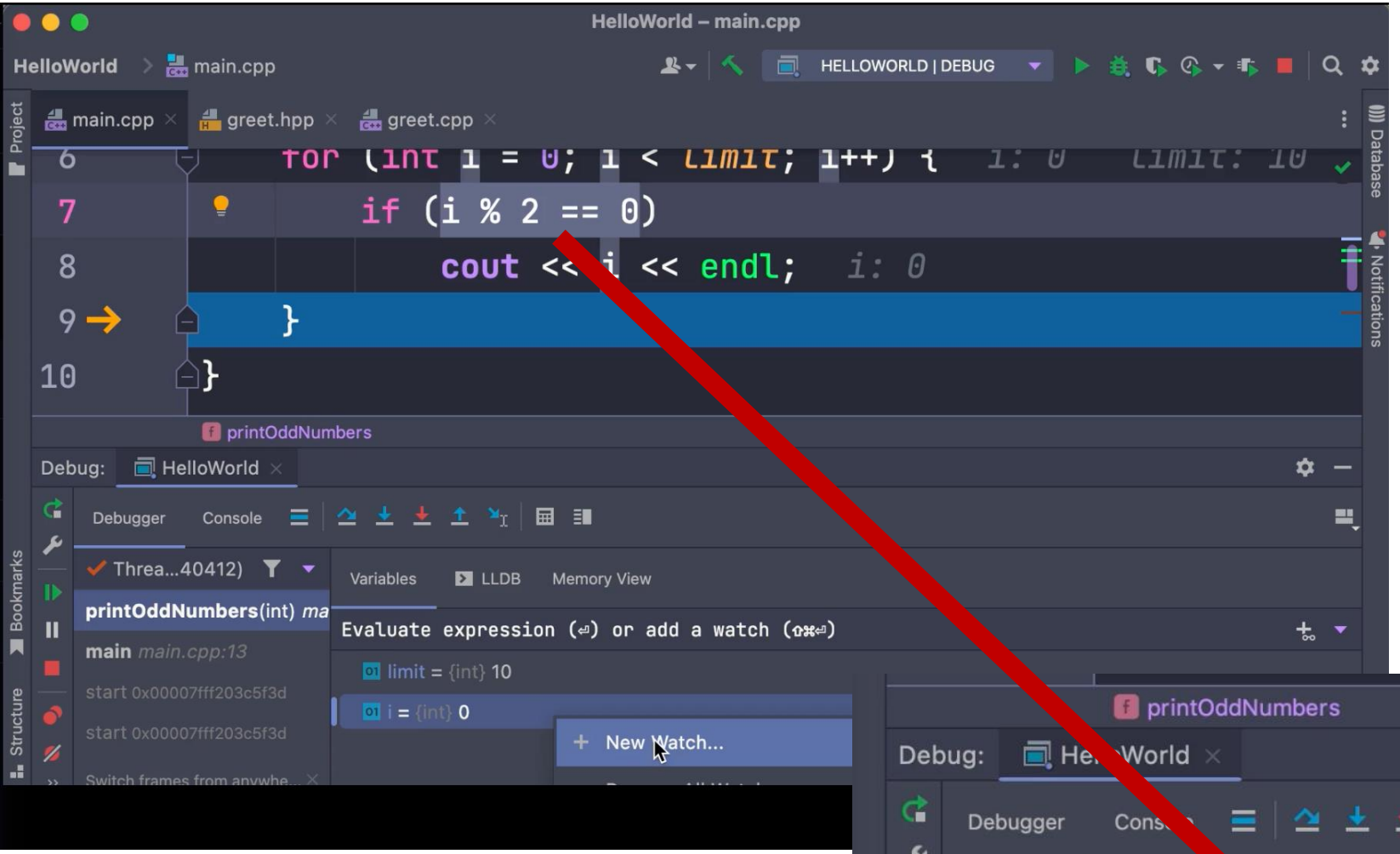
# Debugging

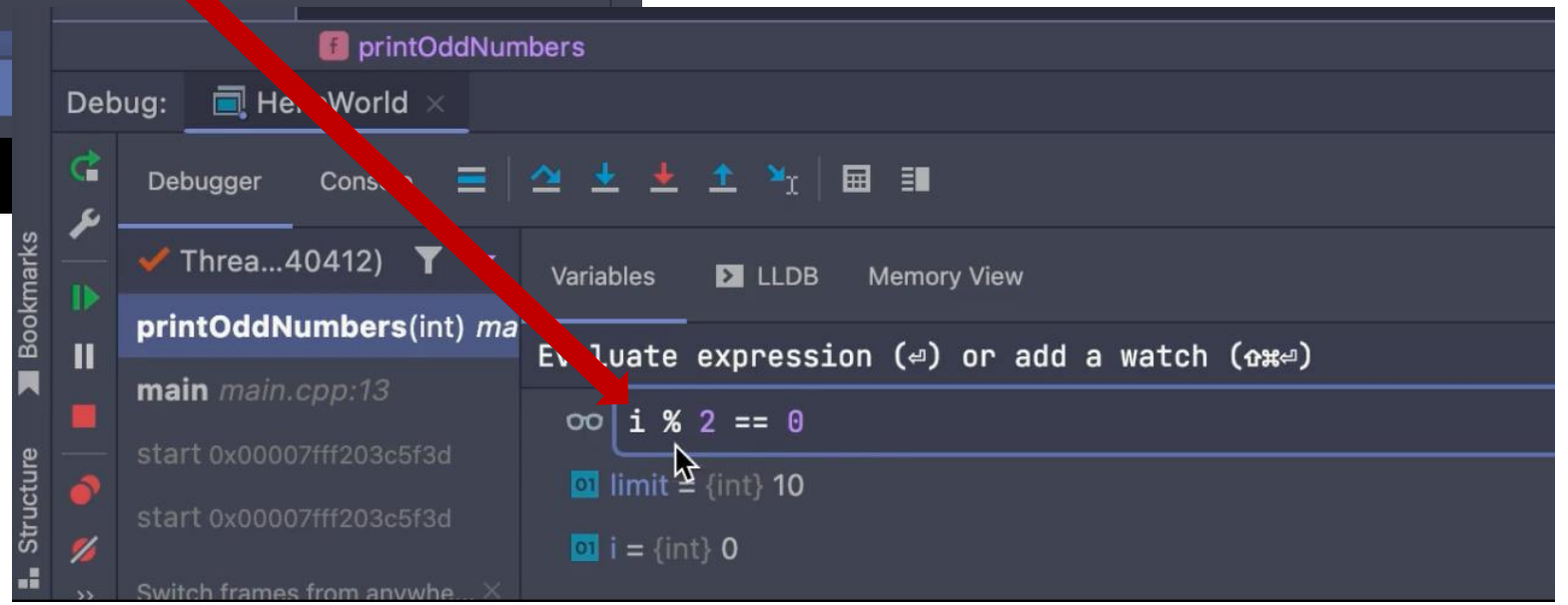
Debug button

Debugging step options



# Debugging




Copy the expression into a watch in the debugger.