1.)#include <iostream>

using namespace std;

class User {

private:

string name;

string Picture;

public:

// Constructor

User(const std::string &name, const std::string &Pic) : name(name), Picture(Pic) {}

// Getter for username

std::string getname() const {

return name;

}

// Getter for profile picture

std::string getPicture() const {

return Picture;

}

};

int main() {

// Creating an object of User class

User user("Haritha", "profile.jpg");

// Accessing the private members using getters

std::cout << "Username: " << user.getname() << std::endl;

std::cout << "Picture: " << user.getPicture() << std::endl;

return 0;

}

3.) #include <iostream>

using namespace std;

class User;

void basicInteract(const User& user1, const User& user2);

class User { // User class definition

private:

std::string username;

public:

User(const std::string& name) : username(name) {} // Constructor

std::string getUsername() const { // Getter for username

return username;

}

friend void basicInteract(const User& user1, const User& user2); // Declare basicInteract as a friend function

};

void basicInteract(const User& user1, const User& user2) { // Definition of basicInteract function

std::cout << user1.getUsername() << " interacts with " << user2.getUsername() << "." << std::endl;

}

int main() { // Main function for testing

User user1("Haritha"); // Create two User objects

User user2("sneha");

basicInteract(user1, user2); // Call basicInteract function

return 0;

}

4.) #include <iostream>

using namespace std;

class User;

class Post;

void interact(User& user, Post& post);

void interact(User& follower, User& followed);

class User {

public:

User(const std::string& name) : name(name) {}

std::string getName() const { return name; }

private:

std::string name;

};

class Post {

public:

Post(const std::string& content) : content(content) {}

std::string getContent() const { return content; }

private:

std::string content;

};

void interact(User& user, Post& post) { // Overloaded interact functions

std::cout << user.getName() << " liked the post: \"" << post.getContent() << "\"" << std::endl;

}

void interact(User& follower, User& followed) {

std::cout << follower.getName() << " started following " << followed.getName() << std::endl;

}

int main() {

User arjun("Haritha");

User ram("navya");

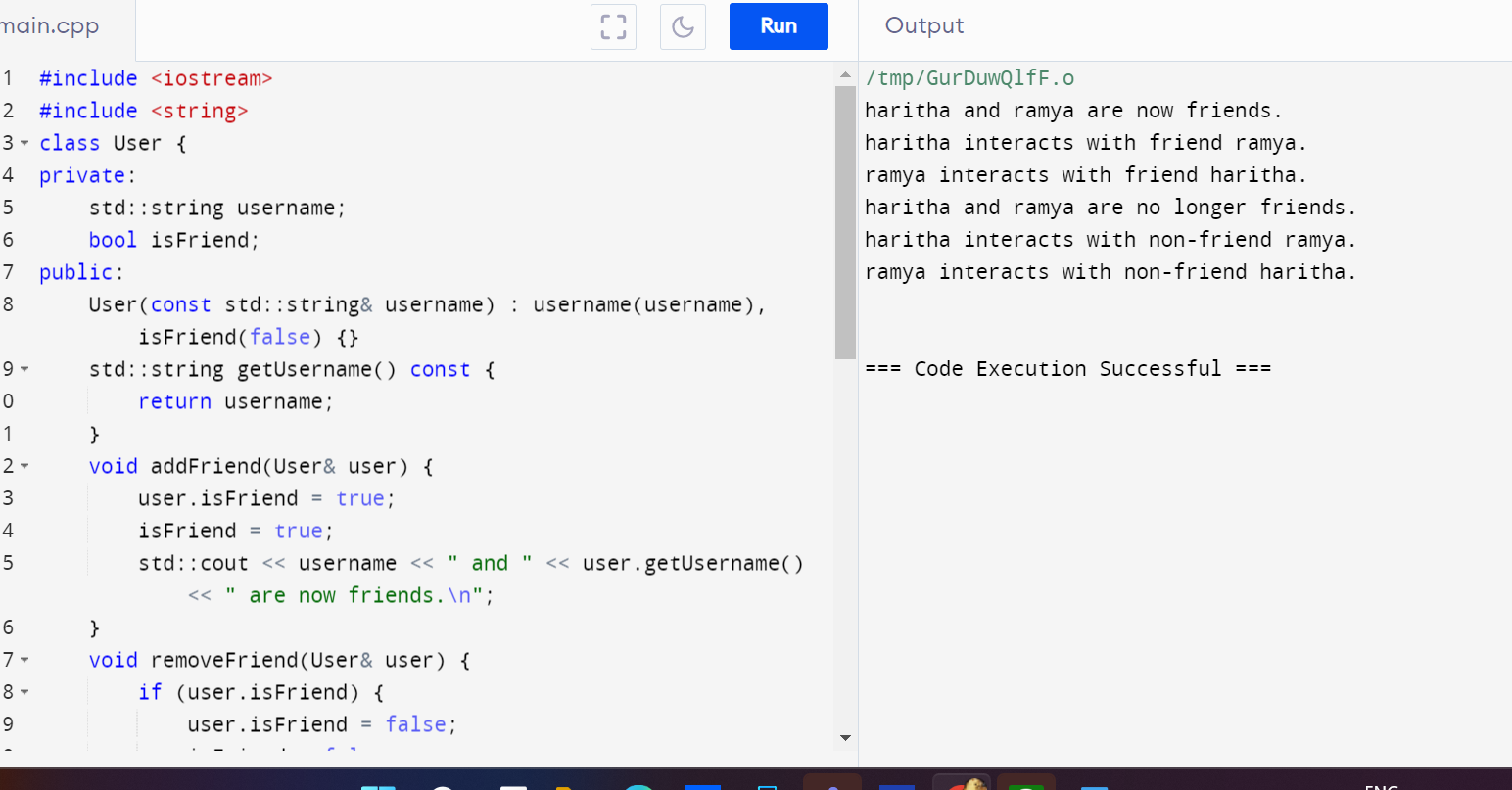
Post post("Wallpaper");

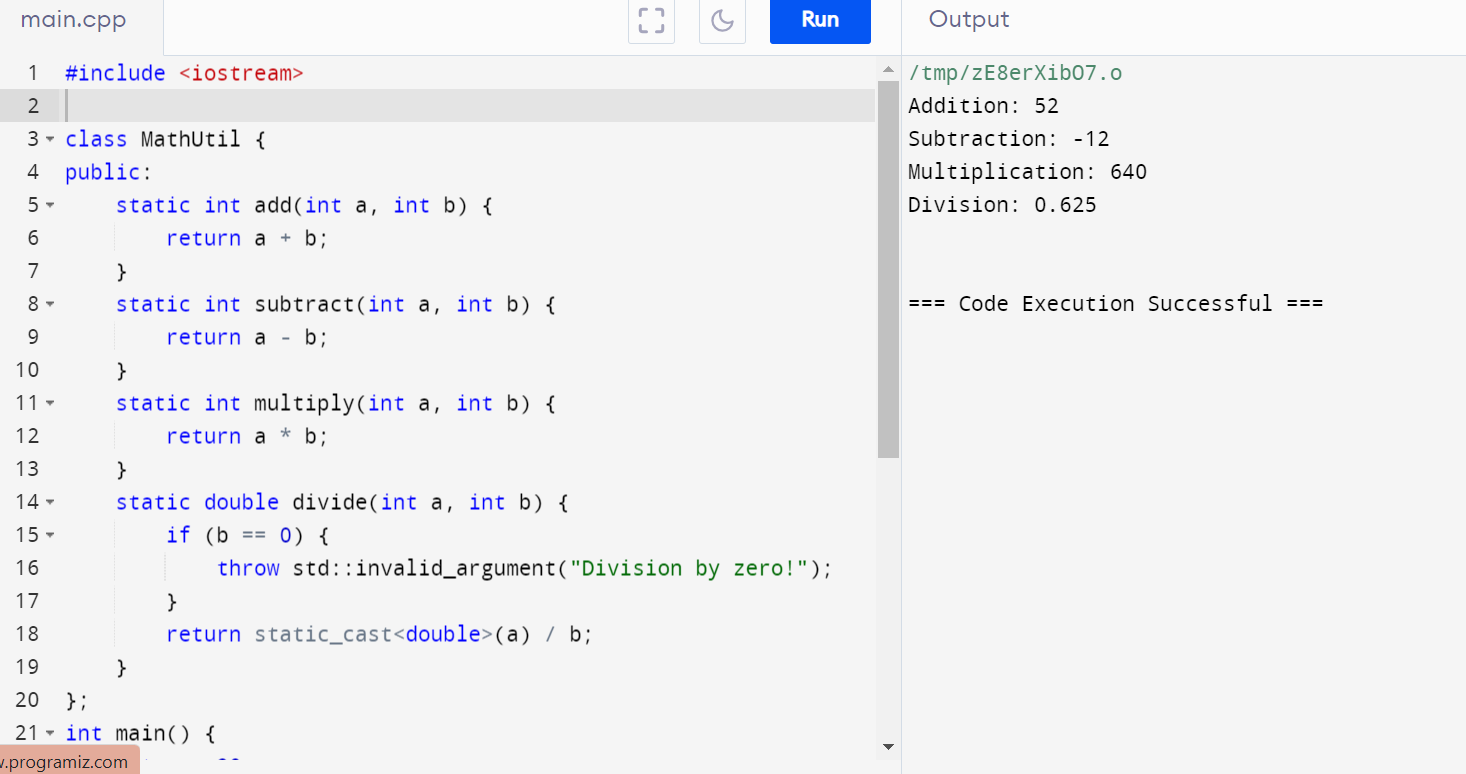
interact(arjun, post); // Arjun liked the post: "Hello, world!"

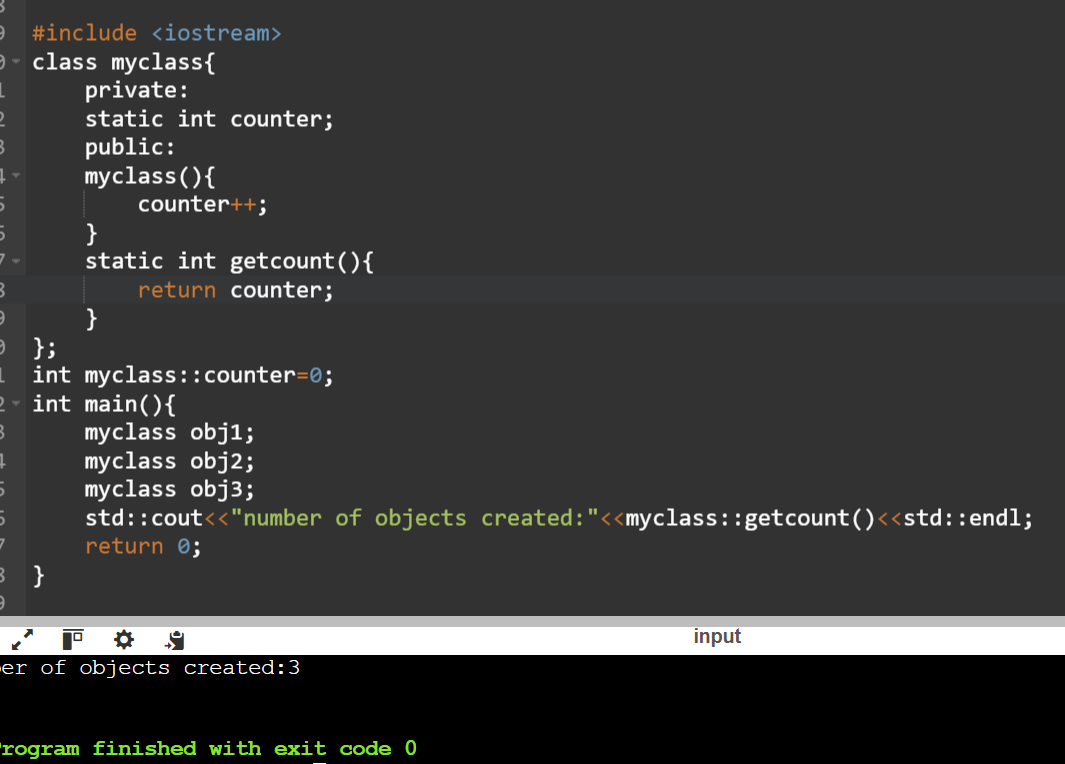
interact(arjun,ram);

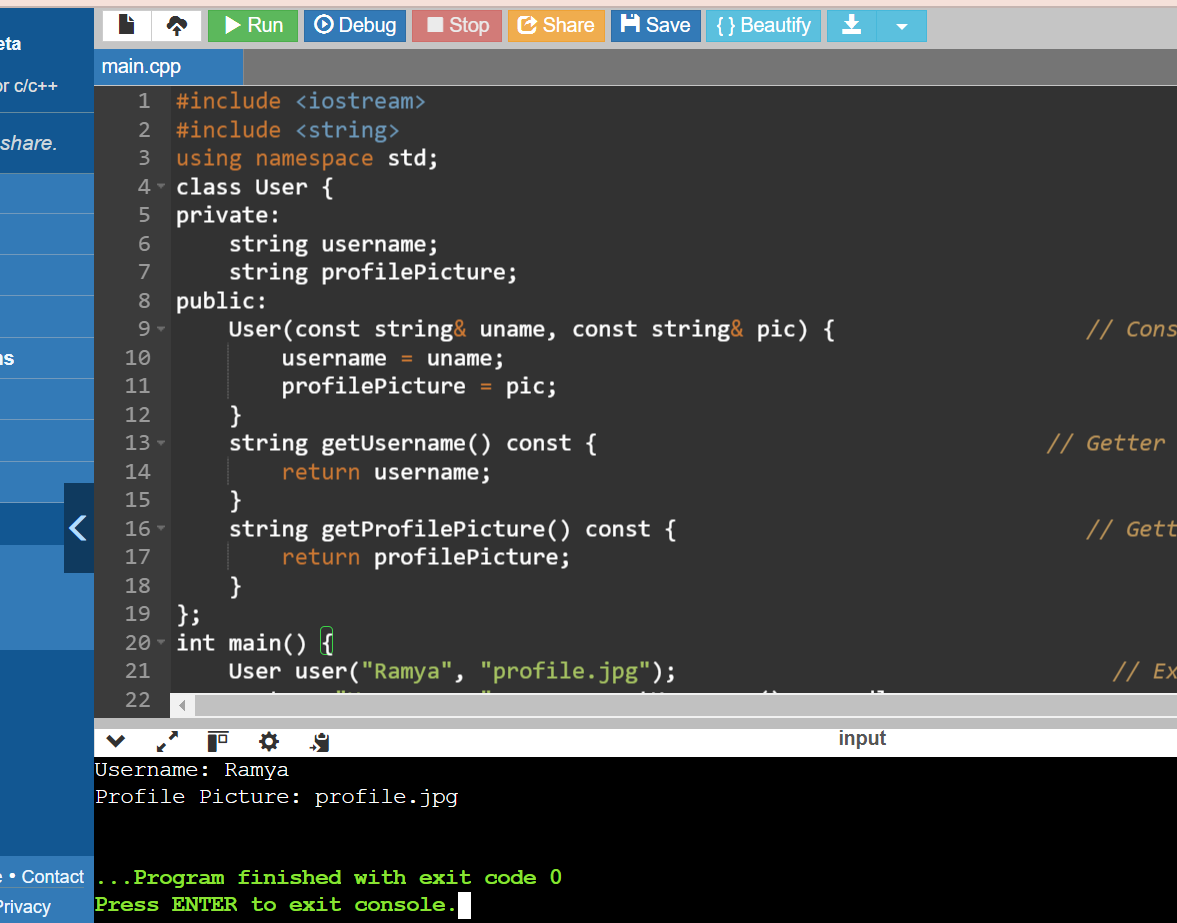
return 0;

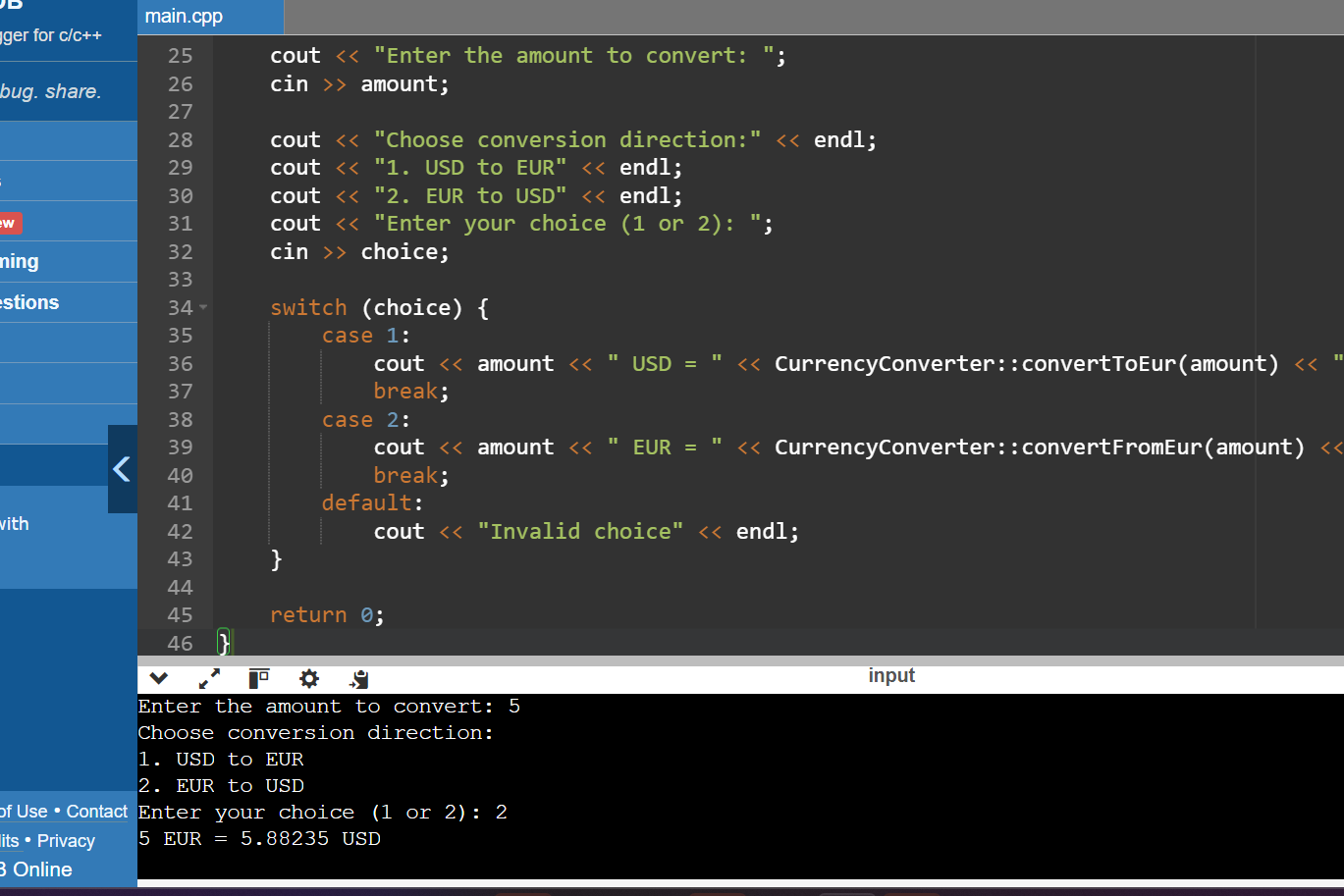
}

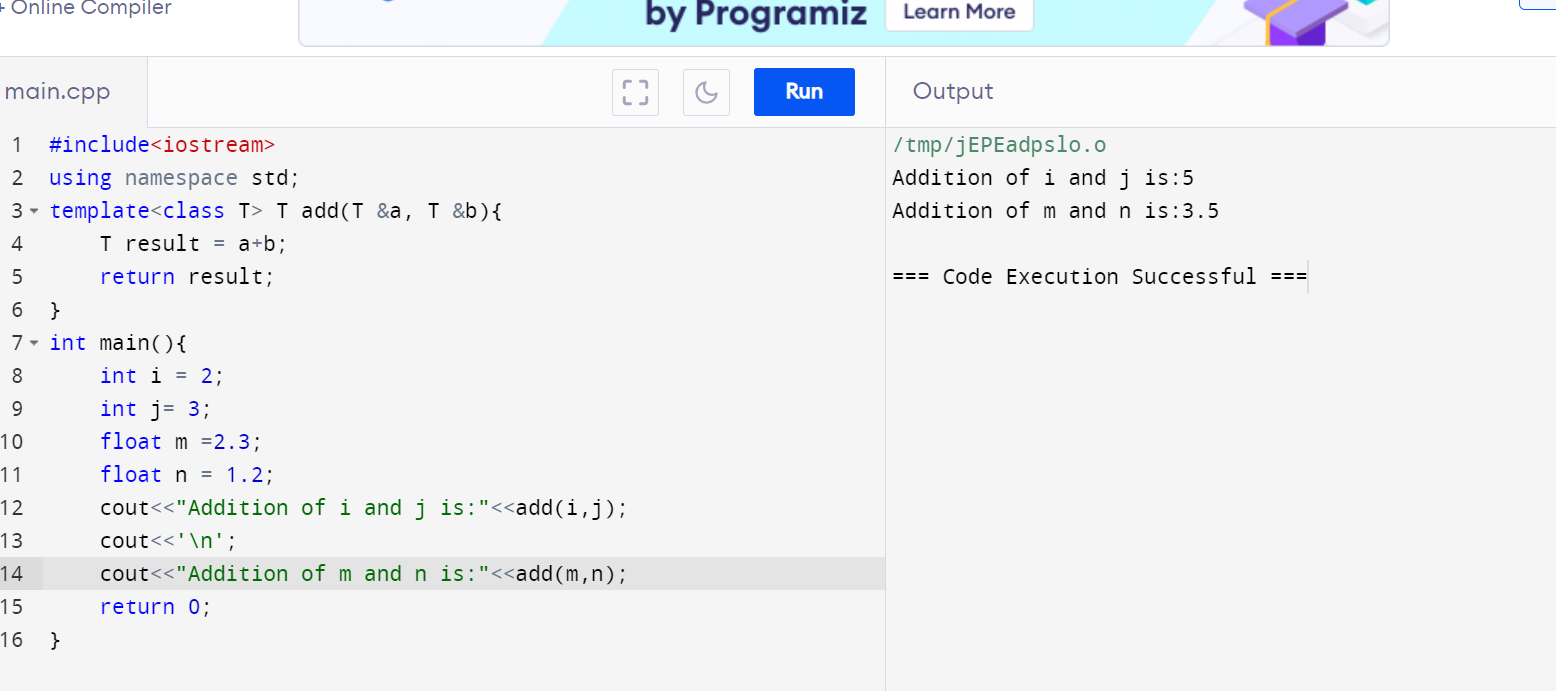


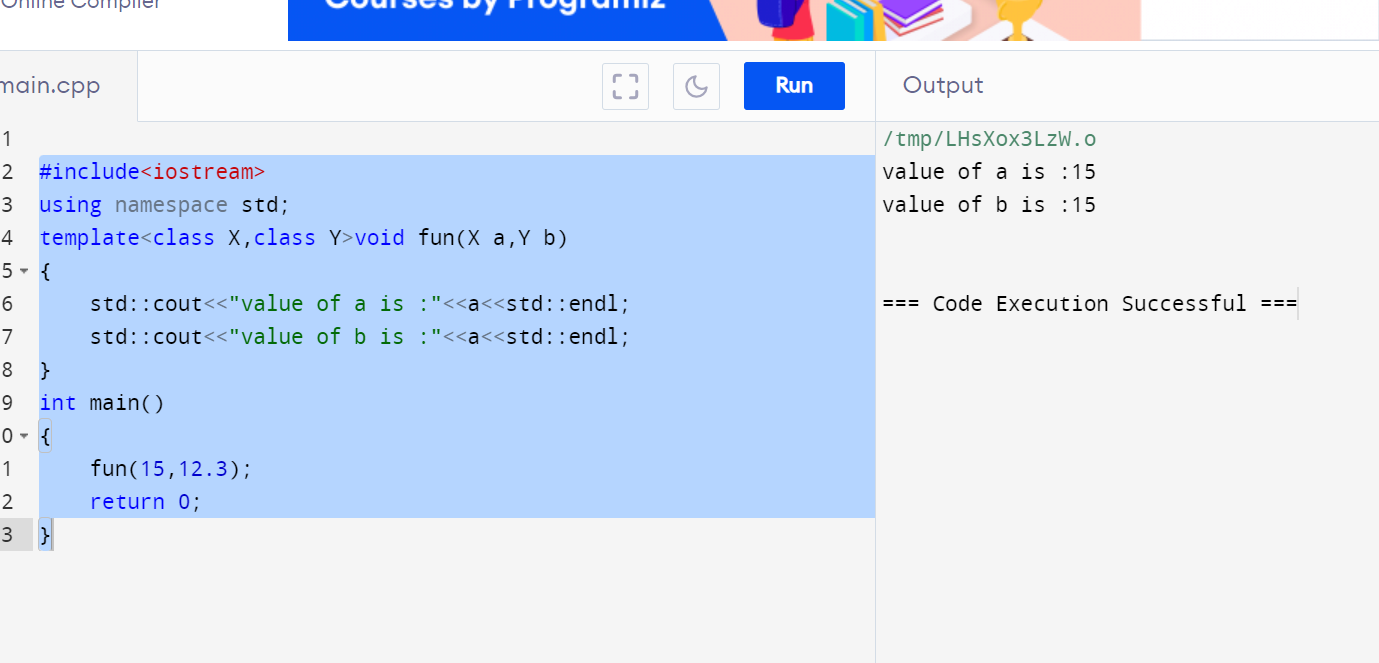


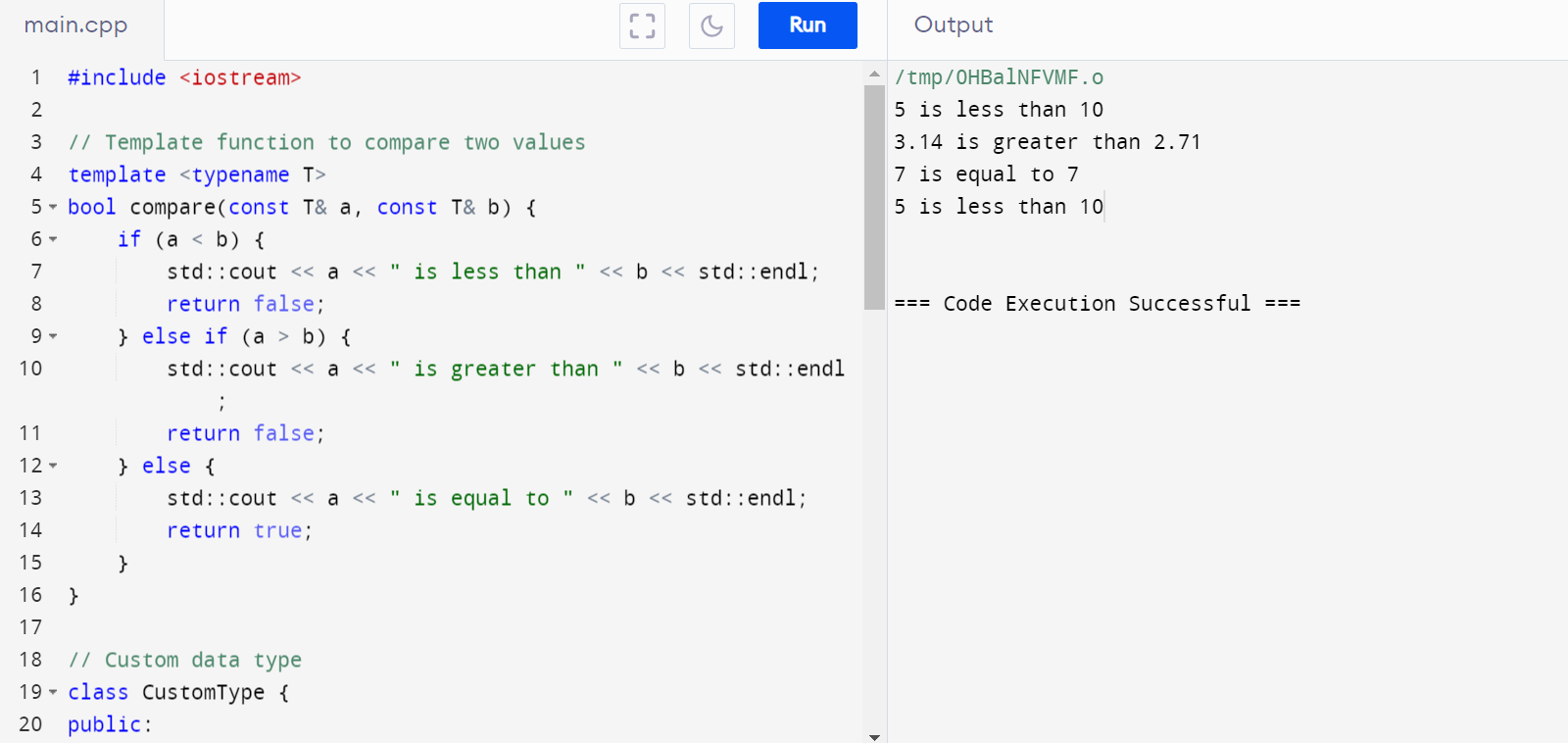


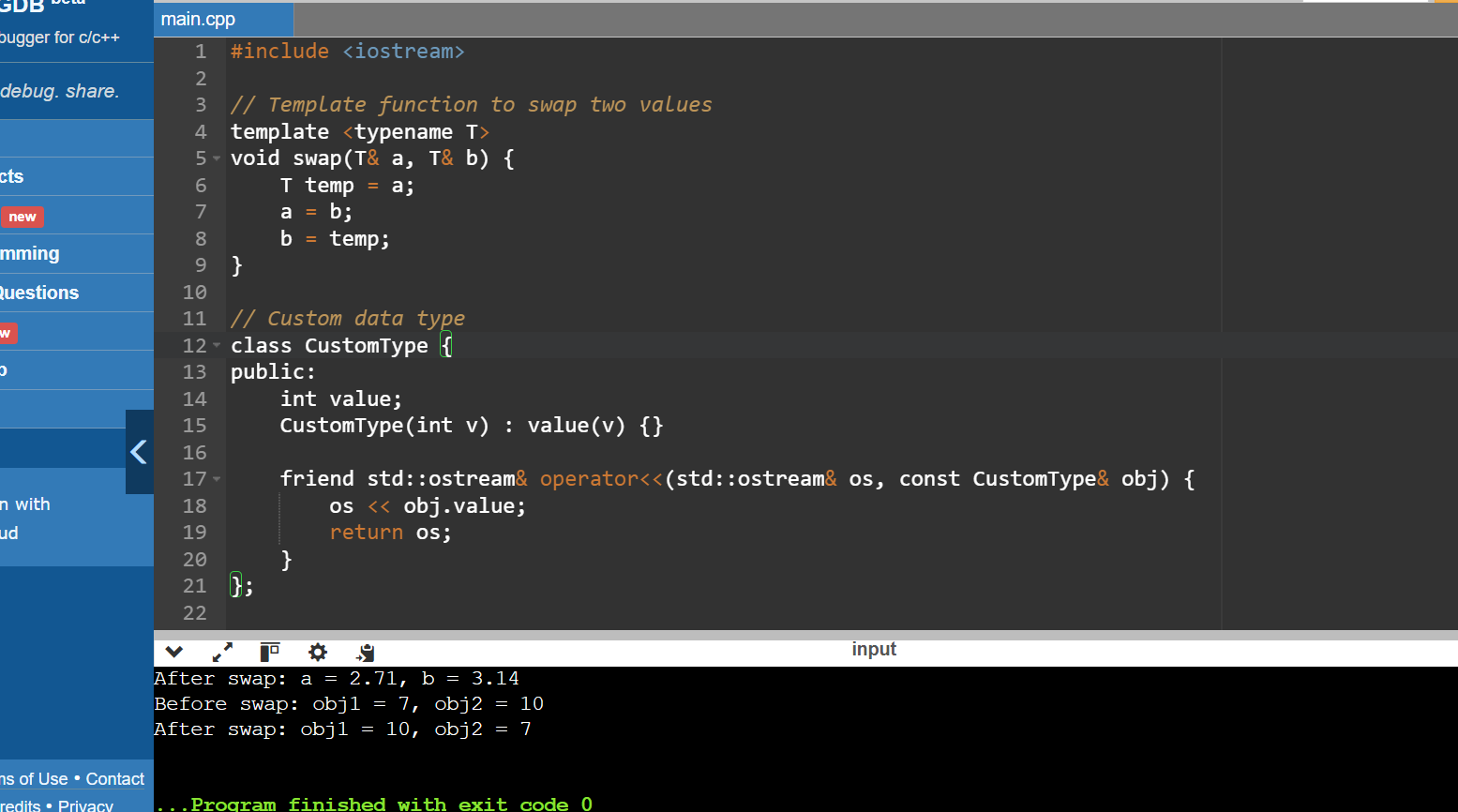












Potential Limitations

 **Complex Data Structures**:

* When dealing with complex data structures (e.g., large objects, containers like std::vector, std::map), swapping can be less efficient because it involves copying the entire content of the structures.
* To avoid inefficiency, std::swap from the Standard Library or move semantics (C++11 and later) should be considered.

 **Deep vs. Shallow Copy**:

* The default swap implementation performs a shallow copy. For data structures containing dynamically allocated memory, this might lead to problems such as double deletion if deep copies are required.

 **Resource Management**:

* If the objects manage resources (e.g., file handles, network connections), the simple swap might not handle resource management correctly. Custom swap functions or std::swap (which can be specialized) should be used.

 **Exception Safety**:

* The swap function might not be exception-safe if the copy constructor or assignment operator throws exceptions. Using move operations can help make the swap function exception-safe.

