

Basic Lambda: Define a lambda expression that takes two integers as arguments and returns their sum. Use auto to infer the return type.

```
#include <iostream>

int main() {

    auto sum = [](int a, int b) {                // Define a lambda expression that takes two
                                                integers and returns their sum

        return a + b;

    };

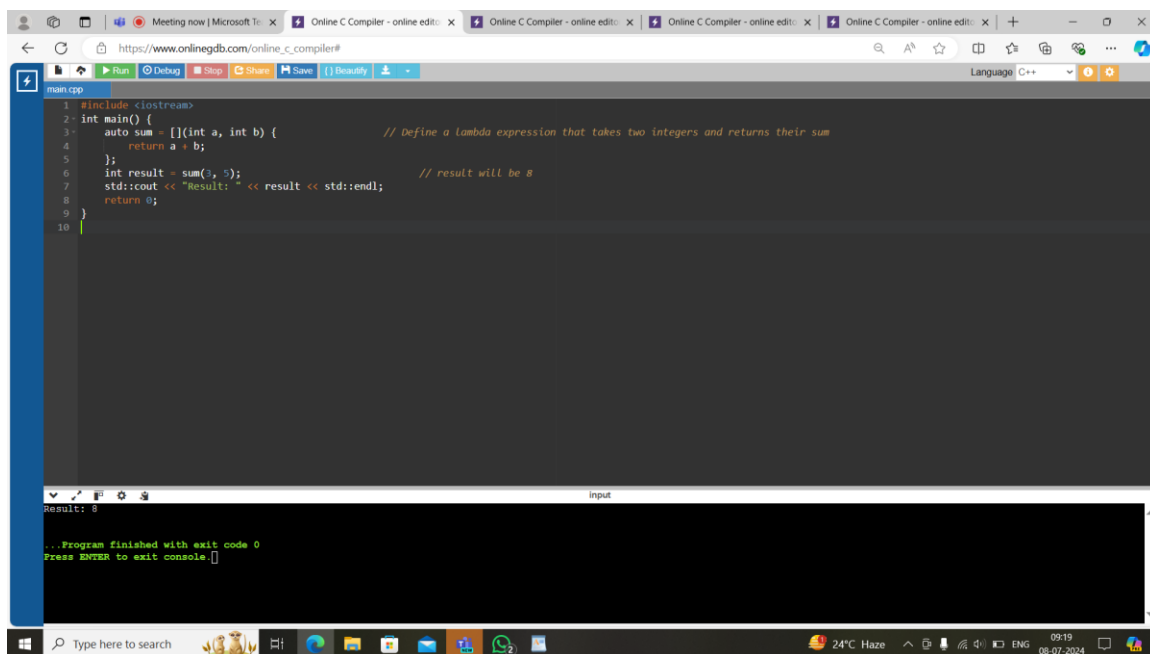
    int result = sum(3, 5);                      // result will be 8

    std::cout << "Result: " << result << std::endl;

    return 0;

}
```

OUTPUT :-

A screenshot of a web browser displaying an online C++ compiler. The browser's address bar shows the URL 'https://www.onlinegdb.com/online_c_compiler#'. The compiler interface includes a menu bar with options like 'Run', 'Debug', 'Stop', 'Share', 'Save', and 'Beautify'. The main area shows a C++ program with line numbers 1 through 10. The code is the same as provided in the previous block. Below the code editor, there is an 'Input' field and an 'Output' section. The output section shows 'Result: 8' and a message '...Program finished with exit code 0. Press ENTER to exit console.' The Windows taskbar is visible at the bottom of the screen.

Capture by Value: Write a lambda that captures an integer by value from the enclosing scope, squares it, and returns the result.

```
#include <iostream>
```

```

int main() {

    int a = 5;                                // Variable in the enclosing scope

    auto square = [a]() {

        return a * a;

    };

    int result = square();                    // Calling the lambda function

    std::cout << "Result: " << result << std::endl;

    return 0;

}

```

OUTPUT :-

The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler#. The code editor contains the following C++ code:

```

1 #include <iostream>
2 int main() {
3     int a = 5;                                // Variable in the enclosing scope
4     auto square = [a]() {
5         return a * a;
6     };
7     int result = square();                    // Calling the lambda function
8     std::cout << "Result: " << result << std::endl;
9     return 0;
10 }
11

```

The output window at the bottom shows:

```

Result: 25

...Program finished with exit code 0
Press ENTER to exit console.

```

The Windows taskbar at the bottom shows the date and time as 09:19 on 08-07-2024.

Capture by Reference: Create a lambda that captures a string by reference, appends a fixed prefix, and returns the modified string.

```

#include <iostream>

#include <string>

int main() {

```

```

std::string str = "Hello";

    auto appendPrefix = [&str](const std::string& prefix) -> std::string {           // Lambda
function capturing str by reference

    return prefix + str;

};

std::string modifiedStr = appendPrefix("Prefix: ");

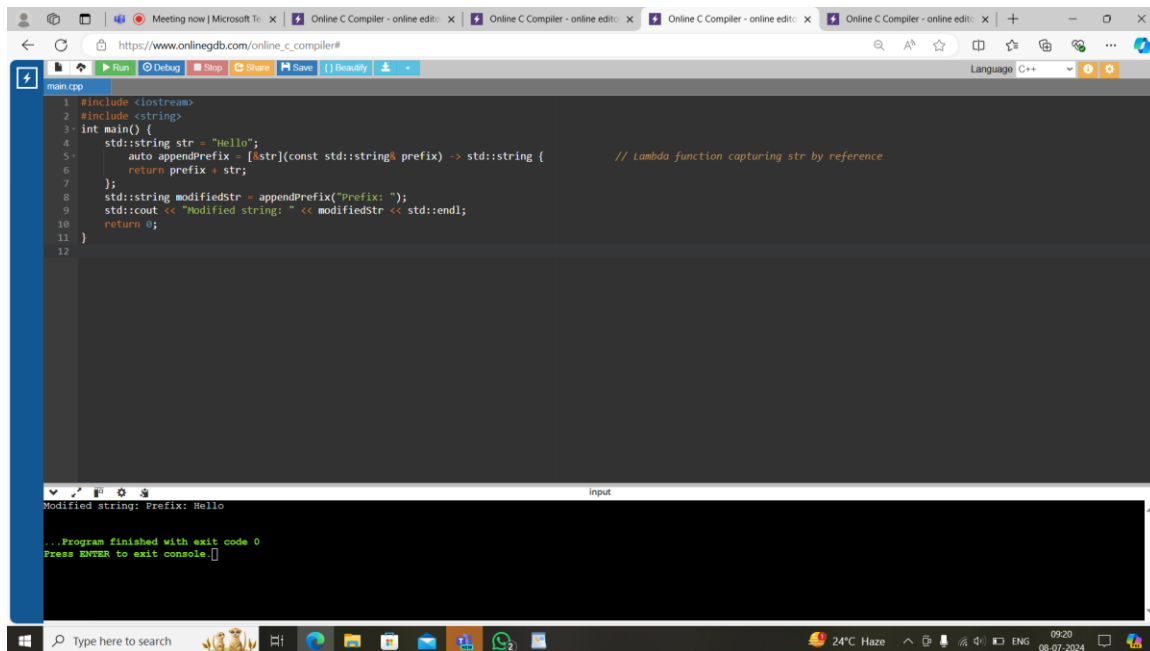
std::cout << "Modified string: " << modifiedStr << std::endl;

return 0;

}

```

OUTPUT :-



The screenshot shows a web browser window with an online C++ compiler. The code is pasted into the editor, and the output is displayed in the console. The output shows the modified string with the prefix 'Prefix: ' and the program finishing with exit code 0.

```

main.cpp
1 #include <iostream>
2 #include <string>
3 int main() {
4     std::string str = "Hello";
5     auto appendPrefix = [&str](const std::string& prefix) -> std::string {           // Lambda function capturing str by reference
6         return prefix + str;
7     };
8     std::string modifiedStr = appendPrefix("Prefix: ");
9     std::cout << "Modified string: " << modifiedStr << std::endl;
10    return 0;
11 }
12

Input
Modified string: Prefix: Hello

...Program finished with exit code 0
Press ENTER to exit console.

```

Multiple Captures: Construct a lambda that captures two variables (an integer and a boolean) by value and performs a conditional operation based on the boolean value.

```
#include <iostream>
```

```
int main() {
```

```
    int num = 10;
```

```

    bool flag = true;

    auto conditionalOperation = [num, flag]() {                // Lambda capturing num and
flag by value

        if (flag) {

            std::cout << "Flag is true. Number is: " << num << std::endl;

        } else {

            std::cout << "Flag is false." << std::endl;

        }

    };

    conditionalOperation();                                    // Calling the lambda

    return 0;

}

```

OUTPUT :-

```

1 #include <iostream>
2 int main() {
3     int num = 10;
4     bool flag = true;
5     auto conditionalOperation = [num, flag]() {                // lambda capturing num and flag by value
6         if (flag) {
7             std::cout << "Flag is true. Number is: " << num << std::endl;
8         } else {
9             std::cout << "Flag is false." << std::endl;
10        }
11    };
12    conditionalOperation();                                    // calling the lambda
13    return 0;
14 }
15

```

Flag is true. Number is: 10

...Program finished with exit code 0
Press ENTER to exit console.

TYPE CASTING :-

```
#include<iostream>
```

```
using namespace std;
```

```
int main(){
```

```
    double a =21.09399;
```

```
    float b = 10.20;
```

```
    int c;
```

```
    c= (int) a;
```

```
    cout<<"line 1 - value of (int)a is :"<<c<<endl;
```

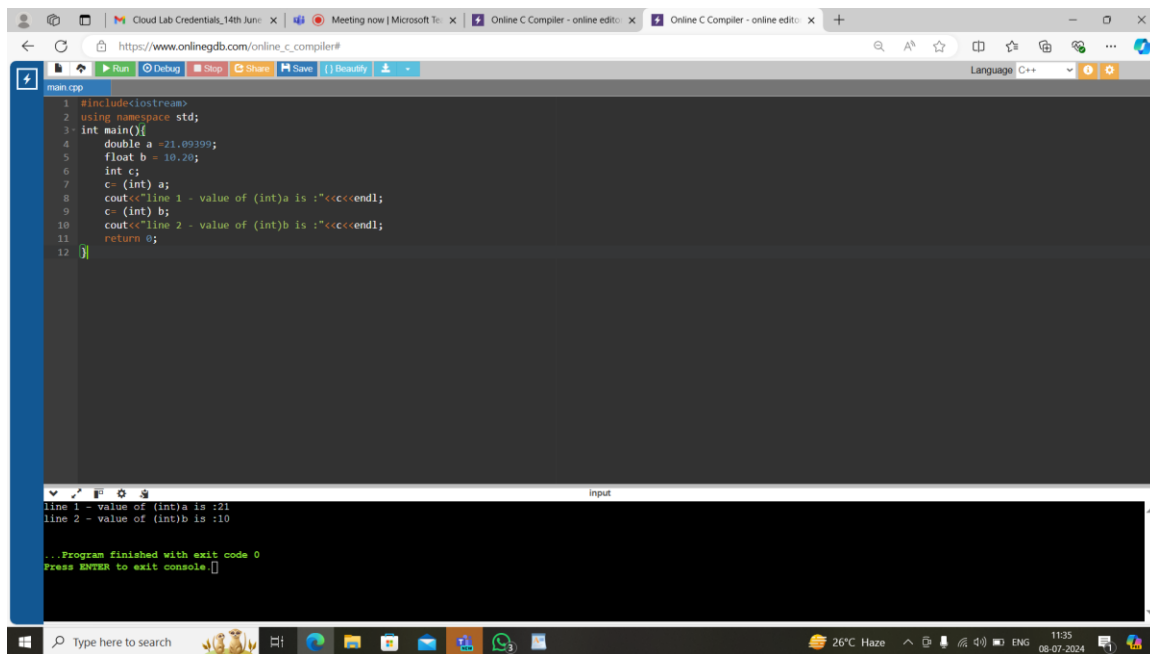
```
    c= (int) b;
```

```
    cout<<"line 2 - value of (int)b is :"<<c<<endl;
```

```
    return 0;
```

```
}
```

OUTPUT :-

A screenshot of a web browser displaying an online C++ compiler. The browser's address bar shows the URL 'https://www.onlinegdb.com/online_c_compiler#'. The compiler interface includes a menu bar with options like 'Run', 'Debug', 'Stop', 'Share', 'Save', and 'Beautify'. The main area shows a C++ program with the following code:

```
1 #include<iostream>
2 using namespace std;
3 int main(){
4     double a =21.09399;
5     float b = 10.20;
6     int c;
7     c= (int) a;
8     cout<<"line 1 - value of (int)a is :"<<c<<endl;
9     c= (int) b;
10    cout<<"line 2 - value of (int)b is :"<<c<<endl;
11    return 0;
12 }
```

The output window at the bottom shows the execution results:

```
line 1 - value of (int)a is :21
line 2 - value of (int)b is :10

...Program finished with exit code 0
Press ENTER to exit console
```

The Windows taskbar at the bottom indicates the system time as 11:35 on 08-07-2024, with a temperature of 26°C and a 'Haze' weather condition.

IMPLICIT TYPE CONVERSION :-

```
#include<iostream>
```

```
using namespace std;
```

```
int main()
```

```

{

    int x = 10;

    char y= 'a';

    x= x+y;

    float z= x+1.0;

    cout<<"x = "<< x <<endl;

    cout<<"y = "<< y <<endl;

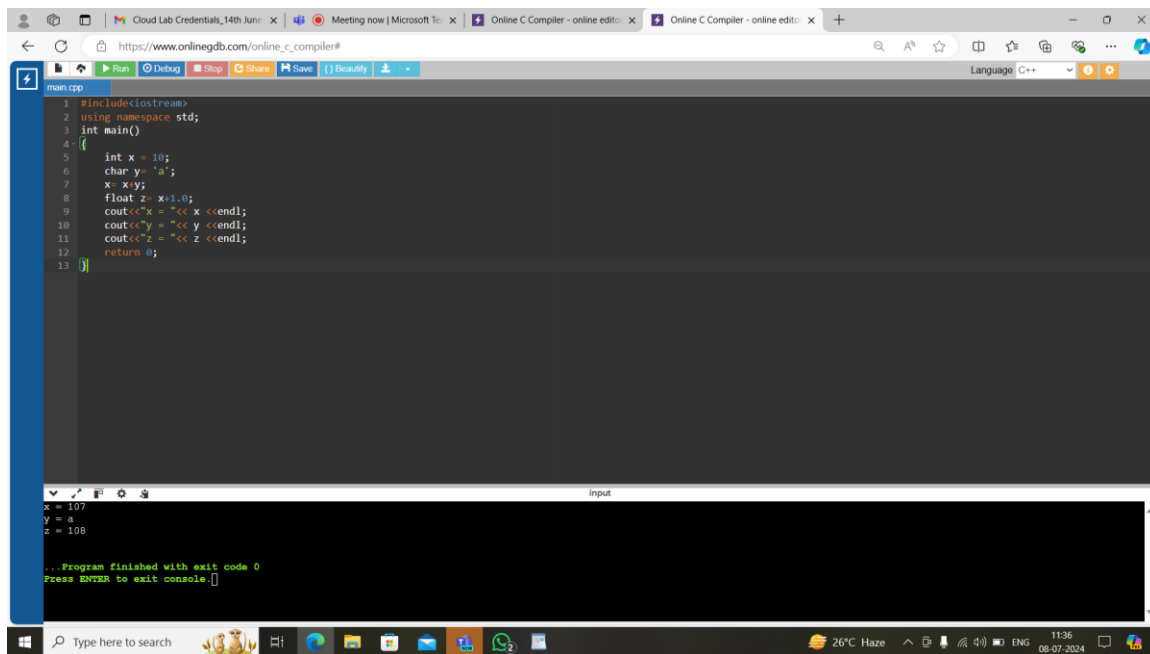
    cout<<"z = "<< z <<endl;

    return 0;

}

```

OUTPUT :-



The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler#. The code editor contains the following C++ code:

```

1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5     int x = 10;
6     char y= 'a';
7     x= x+y;
8     float z= x+1.0;
9     cout<<"x = "<< x <<endl;
10    cout<<"y = "<< y <<endl;
11    cout<<"z = "<< z <<endl;
12    return 0;
13 }

```

The output window shows the following results:

```

x = 107
y = a
z = 108

```

Below the output, it says: "...Program finished with exit code 0. Press ENTER to exit console."

EXPLICIT TYPE CONVERSION :-

```

#include<iostream>

using namespace std;

int main(){

    double x = 1.2;

```

```

int sum = (int)x+1;

cout<<"sum = "<<sum;

return 0;

}

```

OUTPUT :-

The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler#. The code editor contains the following C++ code:

```

1 #include<iostream>
2 using namespace std;
3 int main(){
4     double x = 1.2;
5     int sum = (int)x+1;
6     cout<<"sum = "<<sum;
7     return 0;
8 }

```

The output window shows the result of the program execution:

```

sum = 2
...Program finished with exit code 0
Press ENTER to exit console.

```

CONVERSION USING CAST OPERATOR :-

```

#include<iostream>

using namespace std;

int main(){

    float f = 3.5;

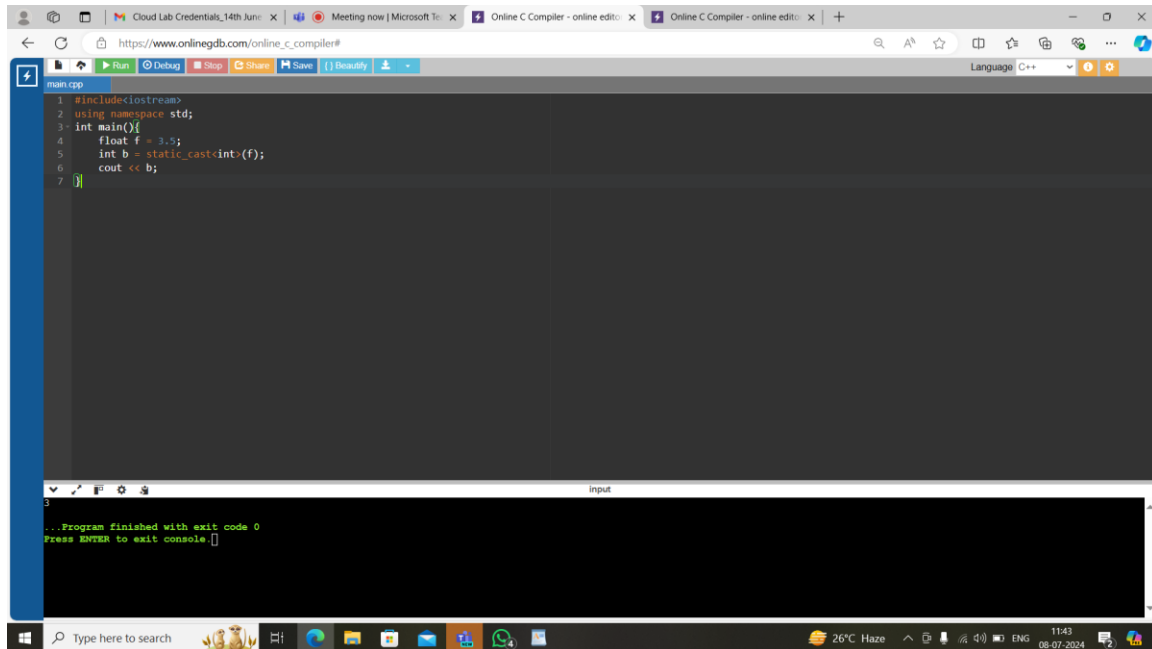
    int b = static_cast<int>(f);

    cout << b;

}

```

OUTPUT :-



The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler#. The browser tabs include 'Cloud Lab Credentials, 14th Jun...', 'Meeting now | Microsoft 365', and two instances of 'Online C Compiler - online edito...'. The compiler interface has a menu bar with 'Run', 'Debug', 'Stop', 'Share', 'Save', and 'Beautify'. The code editor contains the following C++ code:

```
1 #include<iostream>
2 using namespace std;
3 int main(){
4     float f = 3.5;
5     int b = static_cast<int>(f);
6     cout << b;
7 }
```

Below the code editor is an 'input' field and an 'output' window. The output window displays the message: '...Program finished with exit code 0' and 'Press ENTER to exit console.' The Windows taskbar at the bottom shows the search bar, task view, and system tray with a temperature of 26°C, time of 11:43, and date of 08-07-2024.

TYPE CASTING :-

```
#include<iostream>
```

```
using namespace std;
```

```
int main(){
```

```
    const int value = 10;
```

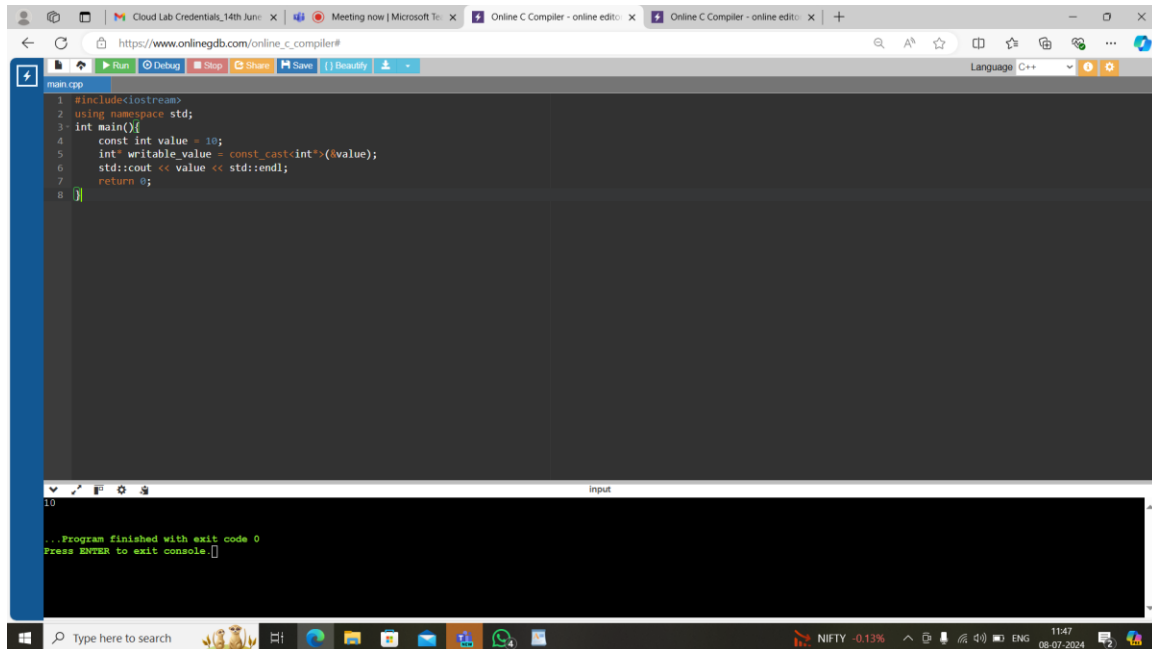
```
    int* writable_value = const_cast<int*>(&value);
```

```
    std::cout << value << std::endl;
```

```
    return 0;
```

```
}
```

OUTPUT :-



The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler#. The browser tabs include 'Cloud Lab Credentials, 14th Jun...', 'Meeting now | Microsoft 365', and two instances of 'Online C Compiler - online edito...'. The compiler interface has a top toolbar with buttons for Run, Debug, Stop, Share, Save, and Beautify. The language is set to C++. The code editor contains the following C++ code:

```
1 #include<iostream>
2 using namespace std;
3 int main(){
4     const int value = 10;
5     int* writable_value = const_cast<int*>(&value);
6     std::cout << value << std::endl;
7     return 0;
8 }
```

Below the code editor is an input field and an output console. The output console shows the message: "...Program finished with exit code 0 Press ENTER to exit console." The Windows taskbar at the bottom shows the search bar, task view, and various application icons, along with system tray icons for NIFTY, network, and time (11:47 on 06-07-2024).

DYNAMIC CASTING :-

```
#include <iostream>
```

```
#include <typeinfo>
```

```
class Base {
```

```
    public:
```

```
    virtual void whoami() {
```

```
        std::cout << "I am Base class object\n";
```

```
    }
```

```
};
```

```
class Derived : public Base {
```

```
    public:
```

```
    void whoami() override {
```

```
        std::cout << "I am Derived class object\n";
```

```
    }
```

```
};
```

```
int main() {
```

```

Base* base_ptr = new Derived;

Derived* derived_ptr = dynamic_cast<Derived*>(base_ptr);

if(derived_ptr != nullptr) {

    derived_ptr->whoami();

}

else {

    std::cout << "Cast failed: Base object is not actually Derived\n";

}

delete base_ptr;

return 0;

}

```

OUTPUT :-

The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler#. The code editor contains the following C++ code:

```

1 #include <iostream>
2 #include <typeinfo>
3 class Base {
4 public:
5     virtual void whoami() {
6         std::cout << "I am Base class object\n";
7     }
8 };
9 class Derived : public Base {
10 public:
11     void whoami() override {
12         std::cout << "I am Derived class object\n";
13     }
14 };
15 int main() {
16     Base* base_ptr = new Derived;
17     Derived* derived_ptr = dynamic_cast<Derived*>(base_ptr);
18     if(derived_ptr != nullptr) {
19         derived_ptr->whoami();
20     }
21     else {
22         std::cout << "Cast failed: Base object is not actually Derived\n";
23     }
24     delete base_ptr;
25     return 0;
26 }

```

The output window shows the result of the program execution:

```

I am Derived class object

...Program finished with exit code 0
Press ENTER to exit console.

```

INTERPRET CASTING :-

```

#include <iostream>

int main() {

    int value = 10;

```

```

float* float_ptr = reinterpret_cast<float*>(&value);

std::cout << *float_ptr << std::endl;

return 0;

}

```

OUTPUT :-

The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler#. The code editor contains the following C++ code:

```

1 #include <iostream>
2 int main() {
3     int value = 10;
4     float* float_ptr = reinterpret_cast<float*>(&value);
5     std::cout << *float_ptr << std::endl;
6     return 0;
7 }

```

The output window shows the result of the program execution:

```

1.4013e+44

```

Below the output, a message states: "Program finished with exit code 0. Press ENTER to exit console."

DYNAMIC TYPECASTING :-

```

#include <iostream>

#include <typeinfo>

class Base {
public:
    virtual void whoami() {
        std::cout << "I am Base class object\n";
    }
};

class Derived : public Base {
public:

```

```

    void whoami() override {

        std::cout << "I am Derived class object\n";

    }

};

int main() {

    double num = 3.14159;

    int integer_part = static_cast<int>(num);

    std::cout << "Original number: " << num << std::endl;

    std::cout << "Integer part: " << integer_part << std::endl;

    Base* base_ptr;

    Derived* derived_ptr = static_cast<Derived*>(base_ptr);

    if(dynamic_cast<Derived*>(base_ptr) != nullptr) {

        derived_ptr = static_cast<Derived*>(base_ptr);

        derived_ptr->whoami();

    }

    else {

        std::cout << "Warning: Base object might not be of type Derived\n";

    }

    Base* actual_derived_ptr = new Derived;

    derived_ptr = dynamic_cast<Derived*>(actual_derived_ptr);

    if(derived_ptr != nullptr) {

        derived_ptr->whoami();

    }

    else {

        std::cout << "Cast failed: Base object is not actually Derived\n";

    }

    delete actual_derived_ptr;

    int value = 10;

```

```

float* float_ptr = reinterpret_cast<float*>(&value);

return 0;

}

```

OUTPUT :-

The screenshot shows a web browser window with an online C++ compiler. The code in the editor is as follows:

```

1 #include <iostream>
2 #include <typeinfo>
3 class Base {
4 public:
5     virtual void whoami() {
6         std::cout << "I am Base class object\n";
7     }
8 };
9 class Derived : public Base {
10 public:
11     void whoami() override {
12         std::cout << "I am Derived class object\n";
13     }
14 };
15 int main() {
16     double num = 3.14159;
17     int integer_part = static_cast<int>(num);
18     std::cout << "Original number: " << num << std::endl;
19     std::cout << "Integer part: " << integer_part << std::endl;
20     Base* base_ptr;
21     Derived* derived_ptr = static_cast<Derived*>(base_ptr);
22     if (dynamic_cast<Derived*>(base_ptr) != nullptr) {
23         derived_ptr = static_cast<Derived*>(base_ptr);
24         derived_ptr->whoami();
25     }
26     else {
27         std::cout << "Warning: Base object might not be of type Derived\n";
28     }
29     Base* original_derived_ptr = new Derived;

```

The output window shows the following results:

```

Original number: 3.14159
Integer part: 3
Warning: Base object might not be of type Derived
I am Derived class object
...Program finished with exit code 0
Press ENTER to exit console.

```

1. Implicit Casting: Write a program that declares an int variable a with the value 10 and a float variable b with the value 3.14. Then, perform the division a / b and print the result. Explain how implicit casting works in this scenario.

```

#include <iostream>

int main() {

int a = 10;

float b = 3.14;

float result = a / b;

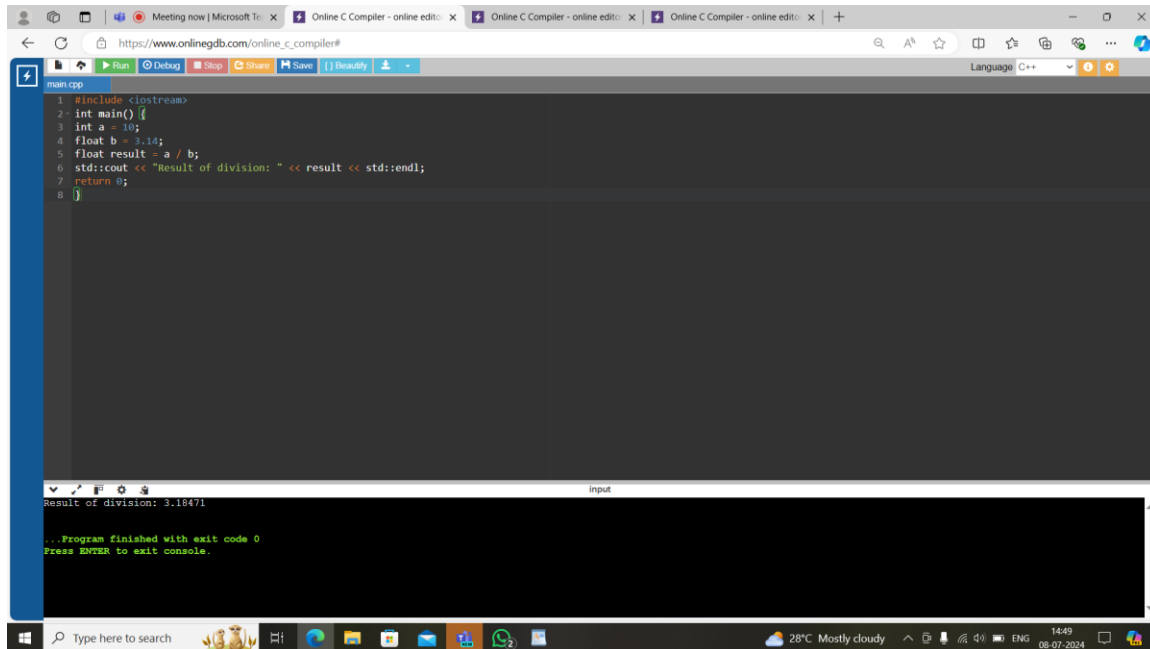
std::cout << "Result of division: " << result << std::endl;

return 0;

}

```

OUTPUT :-



```
1 #include <iostream>
2 int main() {
3     int a = 10;
4     float b = 3.14;
5     float result = a / b;
6     std::cout << "Result of division: " << result << std::endl;
7     return 0;
8 }
```

Result of division: 3.18471

...Program finished with exit code 0
Press ENTER to exit console.

Implicit Conversion :-

The variable a is of type int with the value 10.

The variable b is of type float with the value 3.14.

When you write a / b , the compiler sees that a is an int and b is a float.

Before performing the division, the compiler automatically converts a (which is 10) to a float. So, 10 becomes 10.0.

Then, the division $10.0 / 3.14$ is performed, resulting in a float value.

Result: The result of a / b is $10.0 / 3.14$, which is approximately 3.18471.

2. Explicit Casting - Data Loss: Declare an int variable x with the value 256 and a char variable y. Assign the value of x to y using explicit casting. Print the value of y. Discuss the data loss that might occur and how to avoid it if necessary.

```
#include <iostream>
```

```
int main() {
```

```
    int x = 256;
```

```
    char y = static_cast<char>(x);
```

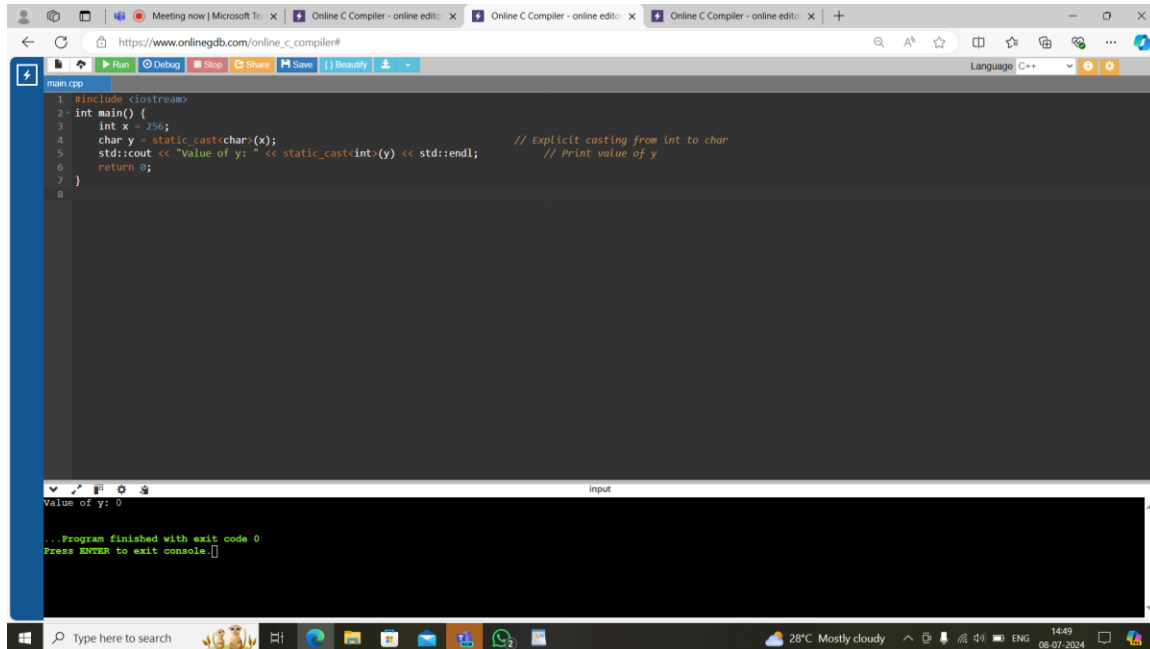
```
// Explicit casting from int to char
```

```
    std::cout << "Value of y: " << static_cast<int>(y) << std::endl;
```

```
// Print value of y
```

```
    return 0;
}
```

OUTPUT :-



The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler#. The code editor contains the following C++ code:

```
1 #include <iostream>
2 int main() {
3     int x = 256;
4     char y = static_cast<char>(x); // Explicit casting from int to char
5     std::cout << "Value of y: " << static_cast<int>(y) << std::endl; // Print value of y
6     return 0;
7 }
```

The output window shows the result of the program execution:

```
Value of y: 0
... Program finished with exit code 0
Press ENTER to exit console.
```

Understanding the Data Loss:

Value Assignment: `int x = 256;` assigns the value 256 to `x`, which is well within the range of `int`.

Explicit Casting: `char y = static_cast<char>(x);` casts the value of `x` (256) to a `char`. In C++, a `char` typically ranges from -128 to 127 (if signed) or 0 to 255 (if unsigned).

Data Loss: Since `char` is usually signed by default in most implementations, the value 256 (which is 0x100 in hexadecimal) exceeds the maximum value representable by a signed `char` (which is 127). Therefore, when 256 is cast to `char`, it undergoes what's called integer overflow and wraps around due to the limited range of `char`. The result of wrapping around depends on the platform's implementation, but typically it becomes 0 because of the two's complement representation.

Output: `std::cout << static_cast<int>(y);` will output the integer value corresponding to `y`. If `y` is 0 due to overflow, `std::cout` will print 0.

Avoiding Data Loss:

Check the Range: Before performing the cast, check if the value of `x` lies within the range of `char`. For a `char`, this would typically mean checking if `x` is between -128 and 127 (for signed `char`) or 0 to 255 (for unsigned `char`).

Use Conditional Logic: If `x` exceeds the range of `char`, you should handle this situation appropriately in your code. This might involve clamping the value or deciding on a default behavior based on your

application's requirements.

Consider Alternative Types: If possible, reconsider whether char is the appropriate type for your needs. If you anticipate needing values outside its range, consider using a larger data type such as int or short.

3. Explicit Casting - Range Conversion: Declare a double variable d with the value 123.456. Use explicit casting to convert d to an int variable i and print i. Explain the behavior when converting from a larger range to a smaller one.

```
#include <iostream>

int main() {

double d = 123.456;

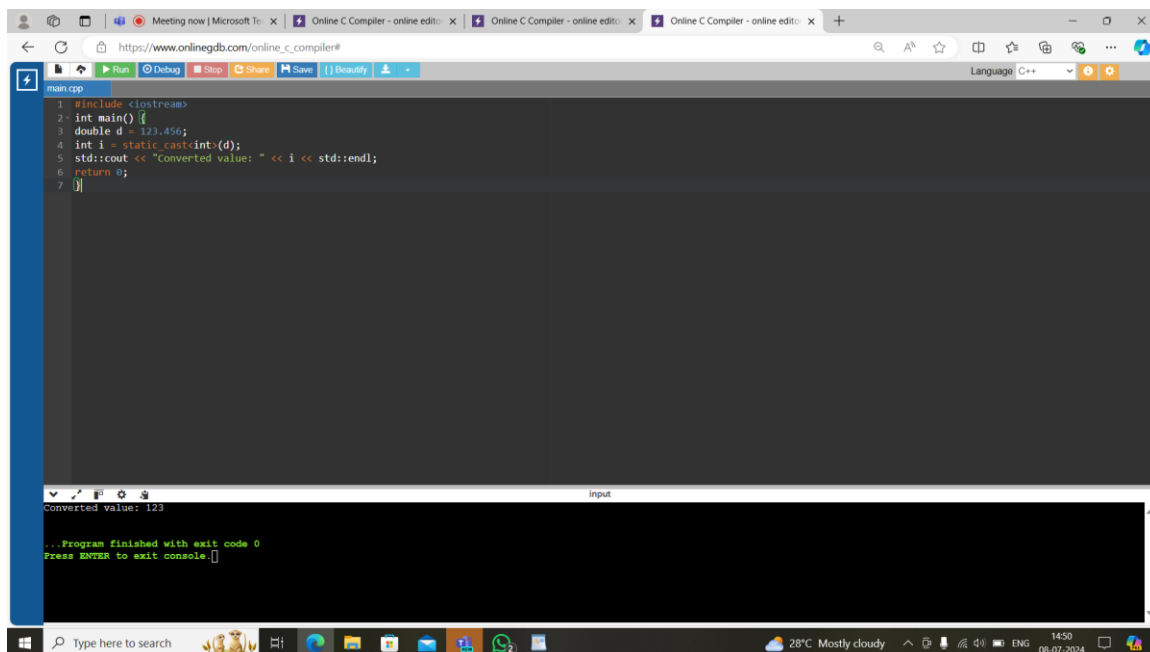
int i = static_cast<int>(d);

std::cout << "Converted value: " << i << std::endl;

return 0;

}
```

OUTPUT :-

A screenshot of a web browser displaying an online C++ compiler interface. The browser's address bar shows the URL 'https://www.onlinegdb.com/online_c_compiler#'. The compiler's toolbar includes buttons for 'Run', 'Debug', 'Stop', 'Share', 'Save', and 'Beautify'. The code editor shows the following C++ code:

```
1 #include <iostream>
2 int main() {
3     double d = 123.456;
4     int i = static_cast<int>(d);
5     std::cout << "Converted value: " << i << std::endl;
6     return 0;
7 }
```

The output window at the bottom displays the result: 'Converted value: 123'. Below the output, a message states: '...Program finished with exit code 0' and 'Press ENTER to exit console.' The browser's taskbar at the bottom shows the system clock as 14:50 on 08-07-2024, along with weather information (28°C, Mostly cloudy) and various system icons.

Explanation of behavior:

Conversion Process: When you cast double d to int i using static_cast<int>(d), the compiler truncates the

fractional part of d (123.456) and assigns the integer part to i.

Resulting Value: In this case, i will be assigned the value 123. This is because converting from double to int truncates towards zero; it simply drops the decimal part of the number.

Range Consideration:

Double (double): Typically holds a range of values much larger than int, and can represent both fractional and whole numbers with greater precision.

Integer (int): Has a smaller range compared to double, usually from -2147483648 to 2147483647 on most systems (32-bit integers).

Behavior with Larger to Smaller Conversion:

If the double value exceeds the range that can be represented by an int, the behavior is undefined in C++. This could result in unexpected values or errors depending on the compiler and system.

For values within the range of int, the conversion is straightforward, and the result is predictable

4. Casting Pointers - Same Type: Declare an int variable num and an int pointer ptr initialized with the address of num. Cast ptr to a float pointer fPtr using explicit casting. Is this casting safe? Why or why not?

```
#include <iostream>
```

```
int main() {
```

```
    int num = 10;
```

```
    int *ptr = &num;                                // Pointer to int, pointing to num
```

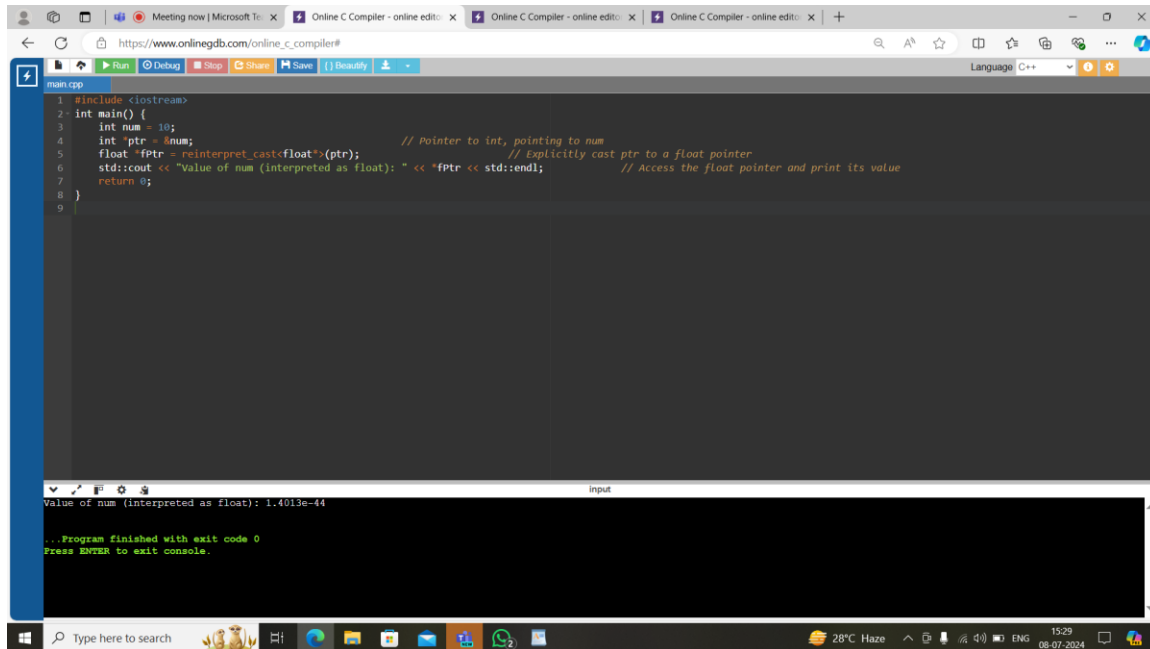
```
    float *fPtr = reinterpret_cast<float*>(ptr);        // Explicitly cast ptr to a
float pointer
```

```
    std::cout << "Value of num (interpreted as float): " << *fPtr << std::endl;    // Access
the float pointer and print its value
```

```
    return 0;
```

```
}
```

OUTPUT :-



```
1 #include <iostream>
2 int main() {
3     int num = 10;
4     int *ptr = &num; // Pointer to int, pointing to num
5     float *fPtr = reinterpret_cast<float*>(ptr); // Explicitly cast ptr to a float pointer
6     std::cout << "Value of num (interpreted as float): " << *fPtr << std::endl; // Access the float pointer and print its value
7     return 0;
8 }
9
```

Value of num (interpreted as float): 1.4013e-44

...Program finished with exit code 0
Press ENTER to exit console.

Is this casting safe? Why or why not?

Safety Considerations:

Type Size and Alignment: The safety of such a cast primarily depends on whether the sizes and alignments of the types involved (int and float in this case) are compatible on your system.

Strict Aliasing Rule: C++ has strict aliasing rules, which mean accessing an object through a pointer of a different type (after casting) can lead to undefined behavior if the types are not compatible.

Specific Issues:

Representation Difference: int and float typically have different representations in memory. Casting an int * to float * assumes that the memory layout and size of an int are compatible with that of a float, which is not always guaranteed.

Alignment: The alignment requirements for int and float might differ. For example, on some systems, int might be aligned on a 4-byte boundary whereas float might require 8-byte alignment. Improper alignment can cause alignment faults or undefined behavior.

5. Casting Pointers - Different Types: Declare an int variable num and a float variable fval. Initialize an int pointer intPtr with the address of num and a float pointer floatPtr with the address of fval. Can you safely cast intPtr to floatPtr? Explain.

```
#include <iostream>
```

```
int main() {
```

```
    int num = 20;
```

```
    // Declare and initialize an int variable
```

```

float fval = 3.14;           // Declare and initialize a float variable

int *intPtr = &num;          // intPtr points to the address of num

float *floatPtr = &fval;      // floatPtr points to the address of fval

std::cout << "Value through floatPtr: " << *floatPtr << std::endl;    // Accessing through
floatPtr (valid)

std::cout << "Value through intPtr: " << *intPtr << std::endl;          // Accessing
through intPtr (valid)

*intPtr = 30;                 // Modify the value of num through intPtr

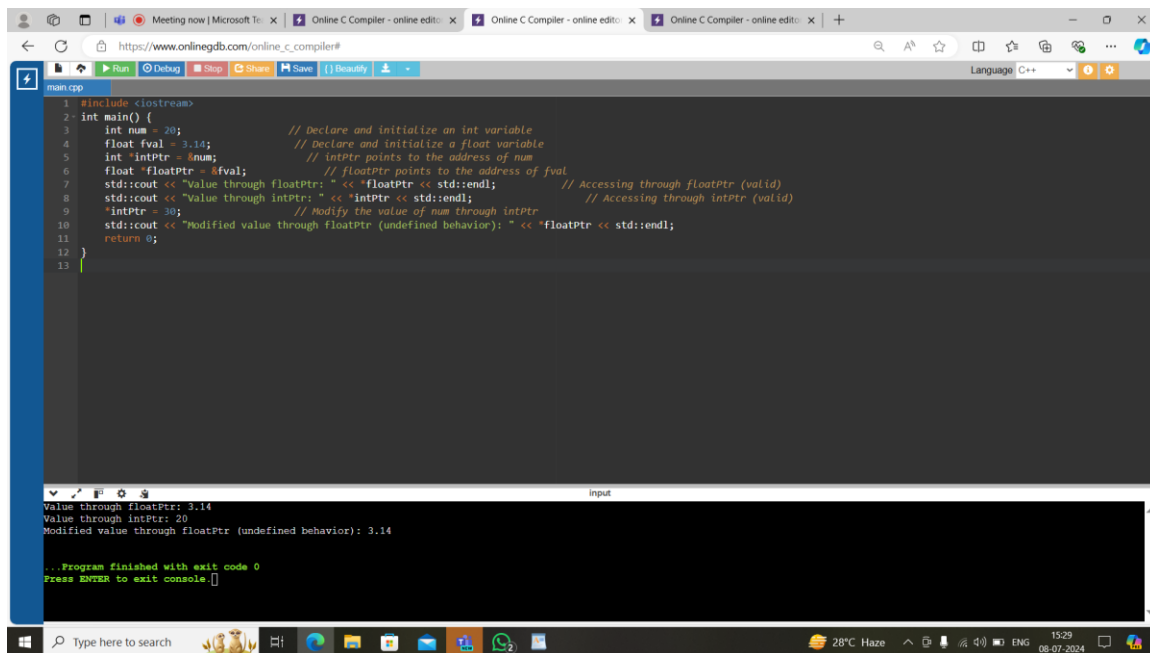
std::cout << "Modified value through floatPtr (undefined behavior): " << *floatPtr << std::endl;

return 0;

}

```

OUTPUT :-



The screenshot shows a web browser window with an online C++ compiler. The code is pasted into the editor, and the output is displayed in the console. The output shows the values of the variables as accessed through pointers, with a warning for undefined behavior when accessing the float variable through the integer pointer.

```

main.cpp
1 #include <iostream>
2 int main() {
3     int num = 20;           // declare and initialize an int variable
4     float fval = 3.14;      // declare and initialize a float variable
5     int *intPtr = &num;      // intPtr points to the address of num
6     float *floatPtr = &fval; // floatPtr points to the address of fval
7     std::cout << "Value through floatPtr: " << *floatPtr << std::endl; // Accessing through floatPtr (valid)
8     std::cout << "Value through intPtr: " << *intPtr << std::endl;      // Accessing through intPtr (valid)
9     *intPtr = 30;           // Modify the value of num through intPtr
10    std::cout << "Modified value through floatPtr (undefined behavior): " << *floatPtr << std::endl;
11    return 0;
12 }
13

Value through floatPtr: 3.14
Value through intPtr: 20
Modified value through floatPtr (undefined behavior): 3.14

...Program finished with exit code 0
Press ENTER to exit console.

```

Can you safely cast intPtr to floatPtr?

No, you cannot safely cast intPtr to floatPtr in C++.

Explanation:

Memory Representation and Alignment:

int and float typically have different sizes and representations in memory. For instance, an int might be 4 bytes (32 bits) and a float might also be 4 bytes (32 bits), but they are stored differently (integer vs

floating-point representation).

Casting `intPtr` to `floatPtr` assumes that the memory layout and interpretation for an `int` can be directly used as a `float`, which is incorrect. The internal representation of an `int` and a `float` are different due to how each type stores and interprets its bits.

Strict Aliasing Rule:

In C++, there's a strict aliasing rule which dictates that you cannot safely access an object of one type through a pointer of another type, except for certain specific cases (like accessing `char*` through another type's pointer). Violating this rule can lead to undefined behavior.

Potential Consequences:

If you were to cast `intPtr` to `floatPtr` and then dereference `floatPtr` to read or modify the float value, you would interpret the memory of `num` (which holds an `int`) as if it were a `float`. This misinterpretation can lead to incorrect results or crashes, depending on the underlying representation and alignment requirements.

6. Casting References - Same Type: Declare an int variable x and an int reference refX assigned to x. Cast refX to a float reference refF. What happens in this case?

```
#include <iostream>
```

```
int main() {
```

```
    int x = 10;
```

```
    int& refX = x;                // refX is a reference to x
```

```
    float& refF = (float&)refX;    // Casting refX to float reference
```

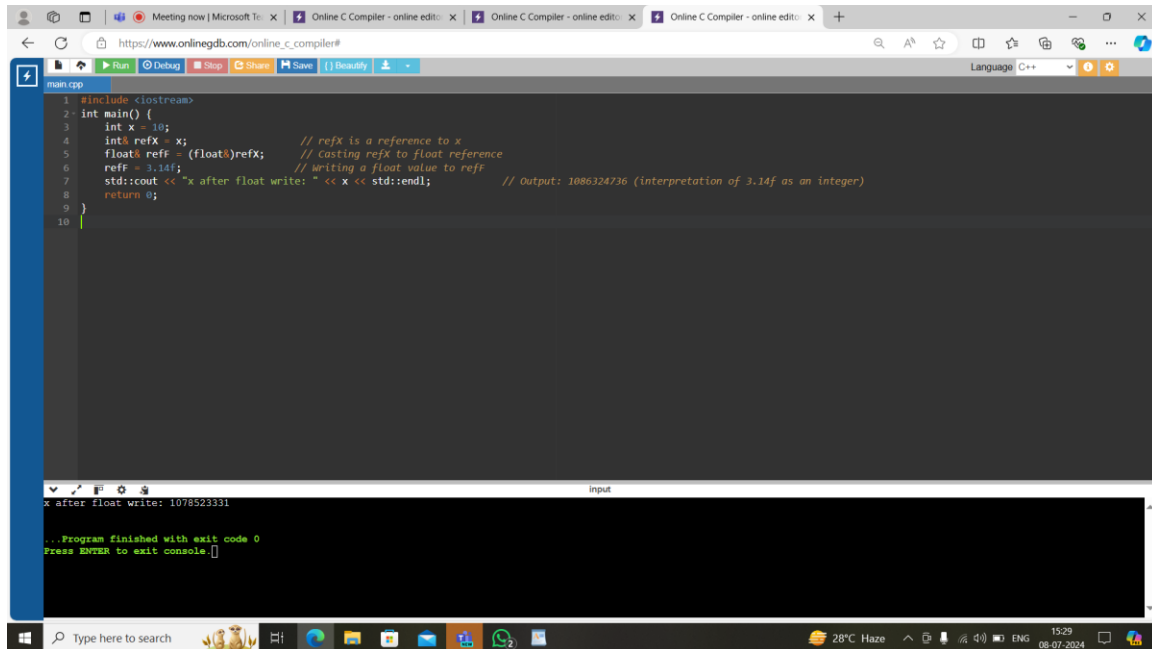
```
    refF = 3.14f;                 // Writing a float value to refF
```

```
    std::cout << "x after float write: " << x << std::endl;    // Output: 1086324736
    (interpretation of 3.14f as an integer)
```

```
    return 0;
```

```
}
```

OUTPUT :-



```
1 #include <iostream>
2 int main() {
3     int x = 10;
4     int& refX = x; // refX is a reference to x
5     float& refF = (float&)refX; // Casting refX to float reference
6     refF = 3.14f; // Writing a float value to refF
7     std::cout << "x after float write: " << x << std::endl; // Output: 1078523331 (interpretation of 3.14f as an integer)
8     return 0;
9 }
10
```

x after float write: 1078523331

...Program finished with exit code 0
Press ENTER to exit console.

What happens in this case:

Type Reinterpretation: The `reinterpret_cast<float&>(refX)` attempts to reinterpret the reference `refX`, which is originally an `int` reference, as a `float` reference (`refF`).

Memory Interpretation: This operation does not change the actual value stored in `x` or `refX`; instead, it changes how the compiler views that memory when accessed through `refF`. It essentially tells the compiler to treat the bits representing the integer as if they were a floating-point number.

Potential Issues:

Undefined Behavior: The C++ standard does not define the exact behavior of accessing an `int` as a `float` (or vice versa) through reinterpretation casts like this. Different compilers may handle this situation differently.

Compatibility: This approach assumes that the representation of an `int` and a `float` are compatible in terms of memory layout, which is generally not guaranteed across different platforms and compilers.

Use Cases:

Low-level Memory Manipulation: Reinterpretation casts are typically used in low-level programming or specific optimization scenarios where you need to access the same memory in different ways (e.g., for type punning or accessing union members).

7. Casting References - Different Types: Declare an `int` variable `x` and a `float` variable `f`. Initialize an `int` reference `refX` with `x`. Can you cast `refX` to refer to `f`? Why or why not?

```
#include <iostream>
```

```
int main() {
```

```

int x = 10;

float f = 3.14;

int& refX = x;                // Initialize int reference refX with x

std::cout << "x: " << x << std::endl; // Output: x: 10

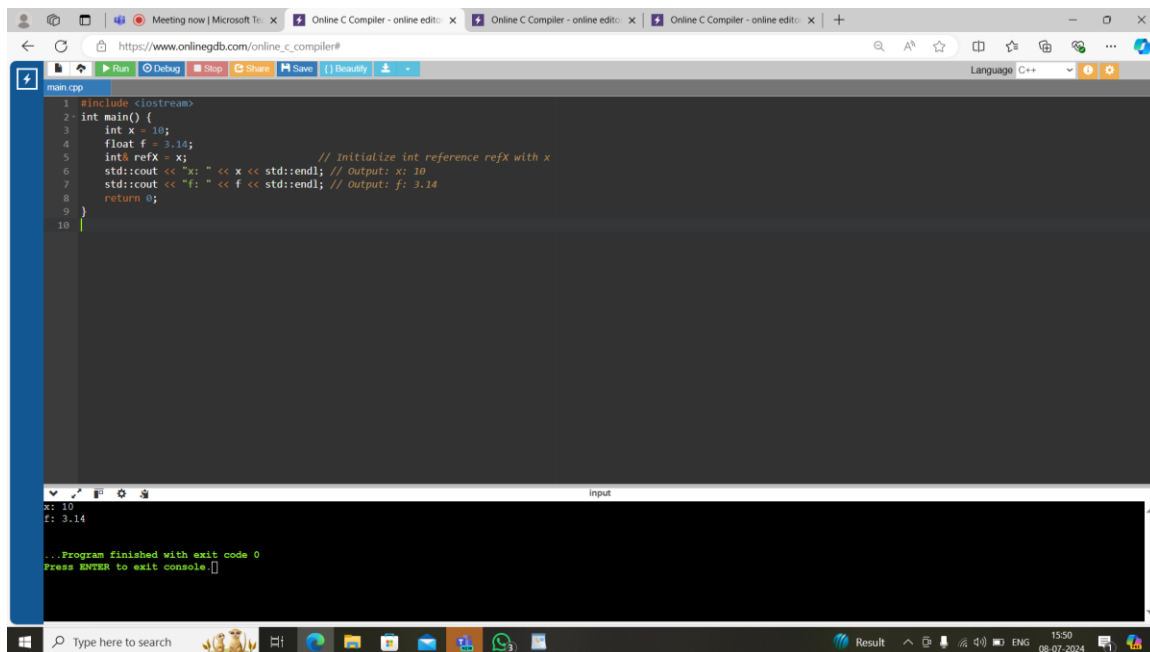
std::cout << "f: " << f << std::endl; // Output: f: 3.14

return 0;

}

```

OUTPUT :-



The screenshot shows a web browser window with an online C++ compiler. The code is pasted into the editor, and the 'Run' button has been clicked. The output window displays the results of the program execution.

```

1 #include <iostream>
2 int main() {
3     int x = 10;
4     float f = 3.14;
5     int& refX = x;                // Initialize int reference refX with x
6     std::cout << "x: " << x << std::endl; // Output: x: 10
7     std::cout << "f: " << f << std::endl; // Output: f: 3.14
8     return 0;
9 }
10

```

Input:

```

x: 10
f: 3.14

```

...Program finished with exit code 0
Press ENTER to exit console.

No, you cannot cast refX to refer to f.

In C++, once a reference is initialized to refer to a particular variable type (in this case, int), it cannot be re-bound to refer to a different type (such as float). This is because references are essentially aliases for existing objects, and their type is fixed once they are initialized.

Here's a breakdown of why you cannot cast refX to refer to f:

Type Compatibility: refX is initialized as an int&, meaning it can only refer to an int variable (x in this case). It cannot directly refer to a float variable (f).

No Implicit Conversion: C++ does not allow implicit conversions between different types of references. Even though int and float might have the same storage size or similar characteristics, the language requires explicit type matching for references.

Type Safety: Allowing such casts would violate type safety principles in C++, where the type of a

reference is meant to be strictly adhered to once initialized. Casting refX to float& would potentially lead to undefined behavior or data corruption, as refX was not initialized to refer to a float.

8. Challenge: Area Calculation (Implicit vs. Explicit): Write two functions to calculate the area of a rectangle. One function should take two int arguments for width and height and return an int area. The other function should take two double arguments and return a double area. Discuss the implications of using implicit and explicit casting in these functions.

```
#include <iostream>

int calculate_area_int(int width, int height) {                // Function to calculate area with
    integer arguments

    int area = width * height;

    return area;
}

double calculate_area_double(double width, double height) {   // Function to calculate area
    with double arguments

    double area = width * height;

    return area;
}

int main() {

    int width_int = 5;

    int height_int = 3;

    int area_int = calculate_area_int(width_int, height_int);

    std::cout << "Area with integer arguments: " << area_int << std::endl;

    double width_double = 5.5;

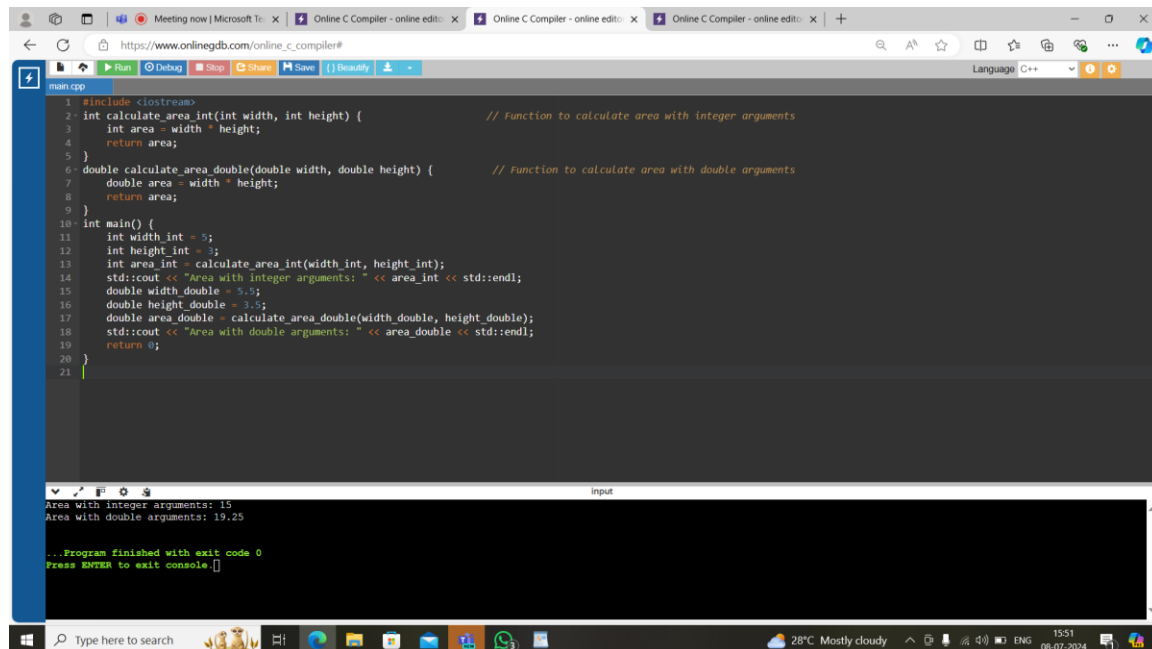
    double height_double = 3.5;

    double area_double = calculate_area_double(width_double, height_double);

    std::cout << "Area with double arguments: " << area_double << std::endl;

    return 0;
}
```

OUTPUT :-



```
1 #include <iostream>
2 int calculate_area_int(int width, int height) {           // Function to calculate area with integer arguments
3     int area = width * height;
4     return area;
5 }
6 double calculate_area_double(double width, double height) { // Function to calculate area with double arguments
7     double area = width * height;
8     return area;
9 }
10 int main() {
11     int width_int = 5;
12     int height_int = 3;
13     int area_int = calculate_area_int(width_int, height_int);
14     std::cout << "Area with integer arguments: " << area_int << std::endl;
15     double width_double = 5.5;
16     double height_double = 3.5;
17     double area_double = calculate_area_double(width_double, height_double);
18     std::cout << "Area with double arguments: " << area_double << std::endl;
19     return 0;
20 }
21
```

Area with integer arguments: 15
Area with double arguments: 19.25
... Program finished with exit code 0
Press ENTER to exit console

Implications of Implicit and Explicit Casting

Implicit Casting: This occurs when the programming language automatically converts data from one type to another without requiring explicit instructions from the programmer.

Explicit Casting: This involves the programmer explicitly converting data from one type to another.

9. Challenge: Temperature Conversion (Casting and Rounding): Create a program that takes a temperature in Celsius as input from the user. Use explicit casting and appropriate rounding techniques to convert it to Fahrenheit and print the result.

```
#include <iostream>
```

```
#include <cmath>
```

```
int main() {
```

```
    double celsius, fahrenheit;           // Declare variables to store temperature in
    Celsius and Fahrenheit
```

```
    std::cout << "Enter temperature in Celsius: ";
```

```
    std::cin >> celsius;
```

```
    fahrenheit = static_cast<double>(celsius) * 9 / 5 + 32;           // Convert Celsius to
    Fahrenheit using the formula:  $F = C * 9/5 + 32$ 
```

```
    fahrenheit = std::round(fahrenheit);
```

```
    std::cout << "Temperature in Fahrenheit: " << fahrenheit << " degrees\n";
```



```

    return 0;
}

```

OUTPUT :-

```

1 #include <iostream>
2 #include <cmath>
3 int main() {
4     double celsius, fahrenheit;           // Declare variables to store temperature in Celsius and Fahrenheit
5     std::cout << "Enter temperature in Celsius: ";
6     std::cin >> celsius;
7     fahrenheit = static_cast<double>(celsius) * 9 / 5 + 32; // Convert Celsius to Fahrenheit using the formula: F = C * 9/5 + 32
8     fahrenheit = std::round(fahrenheit);
9     std::cout << "Temperature in Fahrenheit: " << fahrenheit << " degrees\n";
10    return 0;
11 }
12

```

input

```

Enter temperature in Celsius: 75
Temperature in Fahrenheit: 167 degrees

...Program finished with exit code 0
Press ENTER to exit console.

```

Explicit Casting: The `static_cast<double>(celsius)` explicitly casts `celsius` to `double` to ensure that the multiplication and division operations yield a `double` result, preserving precision.

Rounding: The `std::round` function is used to round the `double` result of the Fahrenheit calculation to the nearest whole number before printing.

Include Directive: The `<cmath>` header is included for the `std::round` function used for rounding the Fahrenheit temperature.

10. Challenge: Pointer Arithmetic with Casting (Safe vs. Unsafe): Demonstrate safe and unsafe pointer arithmetic with casting. Explain the potential consequences of unsafe pointer manipulation.

Safe Pointer Arithmetic with Casting :

```
#include <iostream>
```

```
int main() {
```

```
    int arr[] = {10, 20, 30, 40, 50};
```

```

int *ptr = arr;                // Pointer to the first element of the array

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

    int value = *(ptr + i);      // Equivalent to arr[i]

    std::cout << "Value at index " << i << ": " << value << std::endl;

}

return 0;

}

```

OUTPUT :-

The screenshot shows a web browser window with the URL https://www.onlinegdb.com/online_c_compiler#. The code editor contains the following C++ code:

```

1 #include <iostream>
2 int main() {
3     int arr[] = {10, 20, 30, 40, 50};
4     int *ptr = arr; // Pointer to the first element of the array
5     for (int i = 0; i < 5; ++i) {
6         int value = *(ptr + i); // Equivalent to arr[i]
7         std::cout << "Value at index " << i << ": " << value << std::endl;
8     }
9     return 0;
10 }
11

```

The output window shows the following results:

```

Value at index 0: 10
Value at index 1: 20
Value at index 2: 30
Value at index 3: 40
Value at index 4: 50
...Program finished with exit code 0
Press ENTER to exit console.

```

Unsafe Pointer Arithmetic with Casting

```

#include <iostream>

int main() {

    int arr[] = {10, 20, 30, 40, 50};

    int *ptr = arr;                // Pointer to the first element of the array

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

        char *charPtr = reinterpret_cast<char*>(ptr);

        char value = *(charPtr + i * sizeof(int));

        std::cout << "Byte at index " << i << ": " << static_cast<int>(value) << std::endl;

    }

}

```

```

    }

    return 0;

}

```

OUTPUT :-

The screenshot shows a web browser window with an online C++ compiler. The code in the editor is as follows:

```

1 #include <iostream>
2 int main() {
3     int arr[] = {10, 20, 30, 40, 50};
4     int *ptr = arr; // Pointer to the first element of the array
5     for (int i = 0; i < 5; ++i) {
6         char *charPtr = reinterpret_cast<char*>(ptr);
7         char value = *(charPtr + i * sizeof(int));
8         std::cout << "Byte at index " << i << ": " << static_cast<int>(value) << std::endl;
9     }
10    return 0;
11 }
12

```

The output in the console is:

```

Byte at index 0: 10
Byte at index 1: 20
Byte at index 2: 30
Byte at index 3: 40
Byte at index 4: 50
...Program finished with exit code 0
Press ENTER to exit console.

```

Potential Consequences of Unsafe Pointer Manipulation :-

Undefined Behavior: Incorrect pointer arithmetic or casting can result in undefined behavior. This means the program's behavior is unpredictable and may vary between different compilers, platforms, or even different runs on the same system.

Memory Corruption: Accessing memory out of bounds or with an incorrect type can corrupt memory. This can lead to crashes or data corruption, which are difficult to debug.

Security Vulnerabilities: Incorrect pointer manipulation can introduce security vulnerabilities such as buffer overflows or injection attacks. This is especially critical in systems handling sensitive data or running in environments where security is paramount.

Platform Dependence: Pointer behavior can differ across different hardware architectures or compilers. Code that works on one platform may behave differently or fail on another.

VECTOR :-

```
#include<iostream>
```

```
#include<vector>
```

```
using namespace std;

int main(){

    vector<int> vec;

    int i;

    cout<<"vector size = "<<vec.size()<<endl;

    for(i = 0;i < 5; i++){

        vec.push_back(i);

    }

    cout<<"extended vector size = "<< vec.size()<<endl;

    for(i = 0;i < 5; i++){

        cout<<"value of vec["<<i<<"]=" "<< vec[i]<<endl;

    }

    vector<int>::iterator v= vec.begin();

    while(v!= vec.end()){

        cout << "value of v = " << *v <<endl;

        v++;

    }

    return 0;

}
```

OUTPUT :-

```
Meeting now | Microsoft To... Online C Compiler - online edit...  
https://www.onlinegdb.com/online_c_compiler#  
Run Debug Stop Share Save Beautify  
Language C++  
main.cpp  
1 #include<iostream>  
2 #include<vector>  
3 using namespace std;  
4 int main(){  
5     vector<int> vec;  
6     int i;  
7     cout<<"vector size = "<<vec.size()<<endl;  
8     for(i = 0;i < 5; i++){  
9         vec.push_back(i);  
10    }  
11    cout<<"extended vector size = "<< vec.size()<<endl;  
12    for(i = 0;i < 5; i++){  
13  
14        cout<<"value of vec["<<i<<"] = "<< vec[i]<<endl;  
15    }  
16    vector<int>::iterator v= vec.begin();  
17    while(v!= vec.end()){  
18        cout << "value of v = " << *v <<endl;  
19        v++;  
20    }  
21    return 0;  
22 }  
input  
value of vec[0]= 0  
value of v = 0  
value of v = 1  
value of v = 2  
value of v = 3  
value of v = 4  
... Program finished with exit code 0  
Press ENTER to exit console.[]  
Type here to search 29°C Mostly cloudy 10:44 06-07-2024
```