**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>

using namespace std;

int main() {

auto sum = [](int a, int b) { return a + b; };

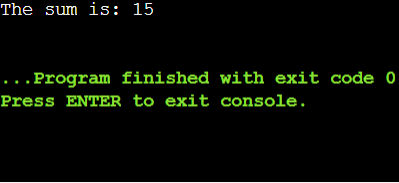
int result = sum(7, 8);

cout << "The sum is: " << result << endl;

return 0;

}

**Output:**



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

#include <iostream>

using namespace std;

int main() {

int x = 10;

auto square = [x] () { return x \* x; };

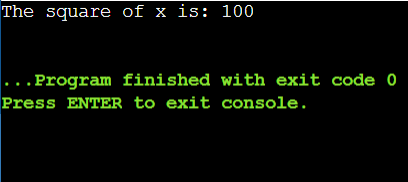
int result = square();

cout << "The square of x is: " << result << endl;

return 0;

}

**Output:**



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

#include <iostream>

using namespace std;

int main() {

string str = "Kavya";

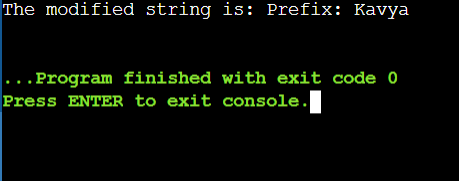
auto modifyString = [&str]() { return "Prefix: " + str; };

cout << "The modified string is: " << modifyString() << endl;

return 0;

}

**Output:**



**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>

using namespace std;

int main() {

int num = 5;

bool condition = true;

auto lambda = [num, condition] () {

if (condition) {

return num \* 2;

} else {

return num \* 3;

}

};

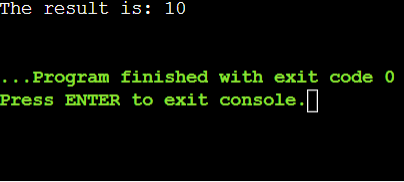
int result = lambda();

cout << "The result is: " << result << endl;

return 0;

}

**Output:**



**Type Casting:**

#include <iostream>

using namespace std;

int main() {

double a = 21.09399;

float b = 10.20;

int c;

c = (int) a;

cout<<"Line1 - Value of (int)a is:"<<c<<endl;

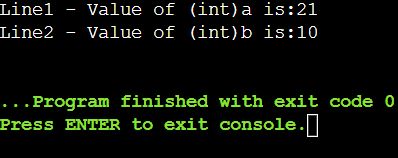
c = (int) b;

cout<<"Line2 - Value of (int)b is:"<<c<<endl;

return 0;

}

**Output:**



**Implicit Type Conversation:**

#include <iostream>

using namespace std;

int main() {

int x=10;

char y='a';

x=x+y;

float z = x + 10;

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

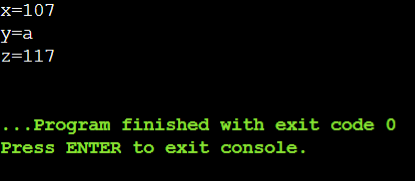
cout<<"y="<<y<<endl;

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

return 0;

}

**Output:**



**Explicit Type Conversion:**

#include <iostream>

using namespace std;

int main() {

double x = 1.2;

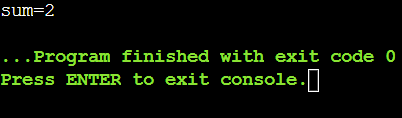
int sum = (int)x+1;

cout<<"sum="<<sum;

return 0;

}

**Output:**



**Conversion Using Cast Operator:**

#include <iostream>

using namespace std;

int main() {

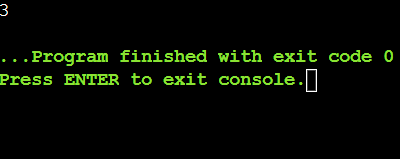
float f =3.5;

int b=static\_cast<int>(f);

cout<<b;

}

**Output:**



#include <iostream>

using namespace std;

int main() {

const int value = 10;

int\* writable\_value = const\_cast<int\*>(&value);

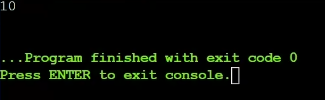
\*writable\_value=20;

cout<<value<<endl;

return 0;

}

**Output:**



#include <iostream>

using namespace std;

class Base{

public:

virtual void whoami(){

cout<<"I am a Base class object\n";

}

};

class Derived : public Base {

public:

void whoami()override{

cout<<"I am a 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{

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

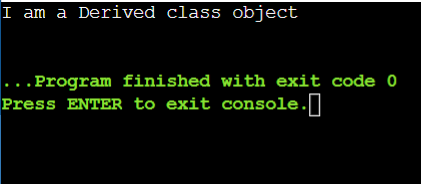
}

delete base\_ptr;

return 0;

}

**Output:**



#include <iostream>

using namespace std;

int main(){

int value = 10;

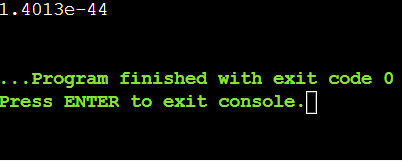
float\* float\_ptr = reinterpret\_cast<float\*>(&value);

cout<<\*float\_ptr<<endl;

return 0;

}

**Output:**



#include <iostream>

using namespace std;

class Base{

public:

virtual void whoami(){

cout<<"I am a Base class object\n";

}

};

class Derived:public Base {

public:

void whoami() override {

cout<<"I am a Derived class object\n";

}

};

int main(){

double num = 3.14150;

int integer\_part = static\_cast<int>(num);

cout<<"original number:"<<num<<endl;

cout<<"Integer part:"<<integer\_part<<endl;

Base \*base\_ptr;

Derived\* derived\_ptr = static\_cast<Derived\*>(base\_ptr);

if(dynamic\_cast<Derived\*>(base\_ptr) != nullptr){

derived\_ptr->whoami();

}

else{

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{

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

}

delete actual\_derived\_ptr;

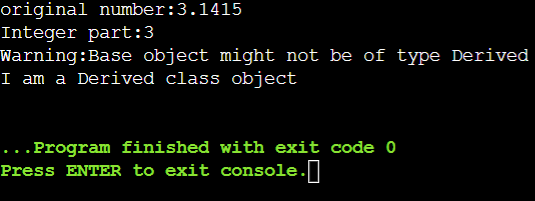
int value =10;

float\*float\_ptr = reinterpret\_cast<float\*>(&value);

return 0;

}

**Output:**



**1. const\_cast (expr)**

**Purpose: Casts away the const or volatile qualifier from an expression. This allows modifying a supposedly constant variable, but be cautious as it can break code that relies on const-correctness.**

**Use Cases: This is generally discouraged as it can lead to unexpected behavior. However, it might be necessary in rare cases when working with legacy code or APIs that don't handle const correctly It seems like you're describing the const\_cast operator in C++. This operator is indeed used to cast away the const or volatile qualifiers from pointers or references. It's typically discouraged because it can lead to undefined behavior if misused, especially when attempting to modify a variable that was originally declared as const.**

**Here's a summary based on your description:**

**Purpose: Casts away the const or volatile qualifier from an expression.**

**Use Cases: It's generally discouraged due to potential undefined behavior. However, it might be necessary in scenarios such as interacting with legacy code or APIs that were not designed to handle const correctly.**

#include <iostream>

using namespace std;

void printString(char\* str) {

cout << str << endl;

}

int main() {

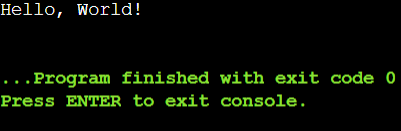
const char\* constStr = "Hello, World!";

printString(const\_cast<char\*>(constStr));

return 0;

}

**Output:**



**2. dynamic\_cast (expr)**

**Purpose: Performs a runtime check to see if a pointer or reference to a base class can be safely cast to a derived class type. If the cast fails (i.e., the object isn't actually of the derived type), it returns nullptr.**

**Use Cases: This is particularly useful for working with polymorphism in inheritance hierarchies. It ensures type safety and avoids potential errors from incorrect casting.**

#include <iostream>

Using namespace std;

class Base {

public:

virtual ~Base() {}

virtual void whoami() {

cout << "I am a Base class object\n";

}

};

class Derived : public Base {

public:

void whoami() override {

cout << "I am a Derived class object\n";

}

};

class AnotherDerived : public Base {

public:

void whoami() override {

cout << "I am AnotherDerived class object\n";

}

};

void identifyObject(Base\* basePtr) {

if (Derived\* derivedPtr = dynamic\_cast<Derived\*>(basePtr)) {

derivedPtr->whoami();

} else if (AnotherDerived\* anotherDerivedPtr = dynamic\_cast<AnotherDerived\*>(basePtr)) {

anotherDerivedPtr->whoami();

} else {

cout << "Unknown object type\n";

}

}

int main() {

Base\* base1 = new Derived;

Base\* base2 = new AnotherDerived;

Base\* base3 = new Base;

identifyObject(base1);

identifyObject(base2);

identifyObject(base3);

delete base1;

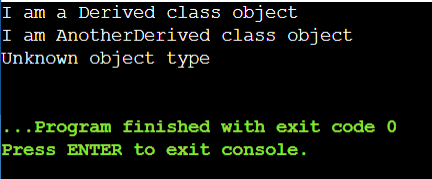
delete base2;

delete base3;

return 0;

}

**Output:**



3**. reinterpret\_cast (expr)**

**Purpose: Reinterprets the bit pattern of an expression as a different type. This allows casting pointers to different pointer types, converting pointers to integers and vice versa (low-level operations). However, it's very powerful and can lead to undefined behavior if not used carefully.**

**Use Cases: This is for advanced scenarios like memory manipulation or interfacing with low-level hardware. Use it with extreme caution as it bypasses type checking**

#include <iostream>

#include <cstdint>

using namespace std;

void manipulateMemory(uintptr\_t address) {

int\* intPtr = reinterpret\_cast<int\*>(address);

\*intPtr = 42;

}

int main() {

int value = 10;

cout << "Before manipulation: " << value << endl;

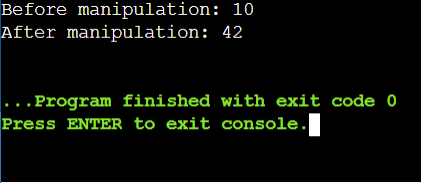
uintptr\_t address = reinterpret\_cast<uintptr\_t>(&value);

manipulateMemory(address);

cout << "After manipulation: " << value << endl;

return 0;

}



**4. static\_cast (expr)**

**Purpose: Performs a basic type conversion between compatible types. It's similar to implicit conversions but allows explicit control.**

**Use Cases: This is commonly used for converting between related data types like int to float or casting a base class pointer to a derived class pointer (upcasting). It's generally safe as long as the conversion is valid**

#include <iostream>

using namespace std;

class Base {

public:

virtual void whoami() {

cout << "I am a Base class object\n";

}

};

class Derived : public Base {

public:

void whoami() override {

cout << "I am a Derived class object\n";

}

};

int main() {

// Numeric conversion

int intValue = 42;

float floatValue = static\_cast<float>(intValue);

cout << "Integer: " << intValue << ", Float: " << floatValue << std::endl;

Derived derivedObj;

Base\* basePtr = static\_cast<Base\*>(&derivedObj);

basePtr->whoami();

Derived& derivedRef = derivedObj;

Base& baseRef = static\_cast<Base&>(derivedRef);

baseRef.whoami();

enum class Color { RED, GREEN, BLUE };

Color color = Color::GREEN;

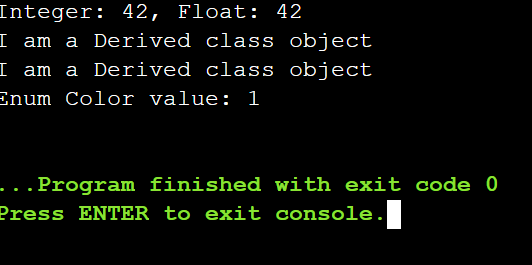
int colorValue = static\_cast<int>(color);

cout << "Enum Color value: " << colorValue << endl;

return 0;

}

**Output:**



**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>

using namespace std;

int main() {

int a = 10;

float b =3.14;

float result = a / b;

cout << "The result is: " << result <<endl;

return 0;

}

1.The compiler automatically converts the int value a to a float to match the type of b.

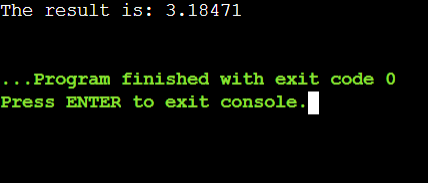
2. The int value a is promoted to a float, which is a wider type, to perform the division operation.

3. The division is performed as a floating-point operation, resulting in a float value.

4. The result of the division is a float value, which is stored in the variable result.

5. The compiler handles the casting automatically, without the need for an explicit cast.

**Output:**



**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>

using namespace std;

int main() {

int x = 256;

char y = (char) x;

cout << "The value of y is: " << (int) y << endl;

return 0;

}

**Data Loss:**

The data loss occurs because a char can only store values up to 255 (unsigned) or 127 (signed). When the value 256 is cast to a char, it gets truncated to 0 (or a negative value if char is signed). This means the original value is lost, and only the lower 8 bits (1 byte) are preserved.

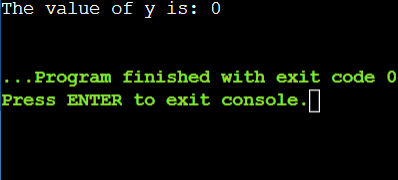
To avoid data loss:

Use a larger data type that can store the full range of values, like int or unsigned int.

Check the value before casting to ensure it fits within the range of the target data type.

Use a casting technique that preserves the original value, like static\_cast or reinterpret\_cast (although these might not always work as intended).

**Output:**



**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>

using namespace std;

int main() {

double d = 123.456;

int i = (int) d; // explicit casting

cout << "The value of i is: " << i << endl;

return 0;

}

d is a double variable with the value 123.456.

i is an int variable.

The value of d is assigned to i using explicit casting (int) d.

-The value of i is printed.

Behavior:

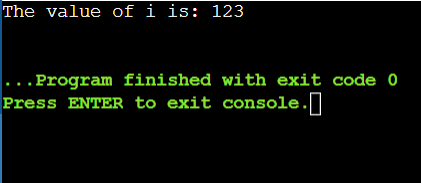
When converting from a larger range (like double) to a smaller range (like int), the excess information is lost. In this case:

The decimal part (.456) is truncated (discarded).

The resulting value is rounded towards zero (123).

This behavior is known as "truncation" and is a common consequence of converting from a larger range to a smaller one.

**Output:**



**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?**

No, this casting is not safe.

When you cast an int\* (integer pointer) to a float\* (float pointer) using explicit casting, you are essentially telling the compiler to treat the memory location pointed to by ptr as if it contained a float value instead of an int value.

However, the memory location still contains an int value, not a float value. If you dereference fPtr and try to use the value as a float, you will get undefined behavior, likely resulting in a garbage value.

This type of casting is known as "pointer type punning" and is generally considered unsafe, as it can lead to unexpected behavior and bugs.

To avoid this issue, you should instead cast the value pointed to by ptr to a float, like this:

float f = (float) \*ptr;

This will perform a safe conversion of the int value to a float value.

#include <iostream>

using namespace std;

int main(){

int num = 5;

int\* ptr = &num;

float\* fPtr = (float\*) ptr;

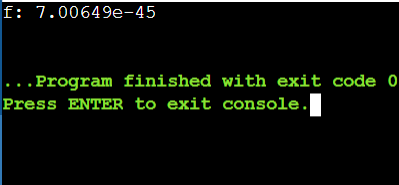
float f = \*fPtr;

cout << "f: " << f << endl;

return 0;

}

**Output:**



**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.**

No, you cannot safely cast intPtr to floatPtr.

intPtr points to an int variable (num), while floatPtr points to a float variable (fval). The types of the variables being pointed to are different, and the memory layouts of int and float are not compatible.

Casting intPtr to floatPtr would allow you to access the memory location of num as if it contained a float value, which is not the case. This would result in undefined behavior, likely leading to garbage values or unexpected behavior.

To safely work with both int and float values, you should use separate pointers or references that match the underlying types, or use a union or struct that contains both types, depending on your specific use case**.**

#include <iostream>

using namespace std;

union Data {

int i;

float f;

};

int main() {

Data data;

data.i = 5;

cout << "int value: " << data.i <<endl;

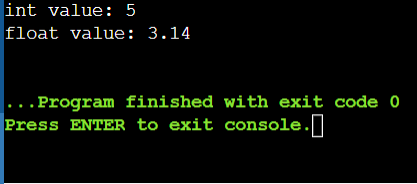
data.f = 3.14;

cout << "float value: " << data.f <<endl;

return 0;

}

**Output:**



**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>

using namespace std;

int main() {

int x = 5;

int& refX = x;

float& refF = (float&) refX;

float y = refF;

return 0;

}

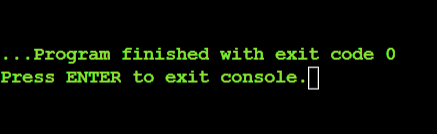
We declare an int variable x and an int reference refX assigned to x.

We try to cast refX to a float reference refF, which is not allowed.

The cast compiles, but the behavior is undefined.

Assigning refF to a float variable y results in a garbage value.

**Output:**



**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?**

No, you cannot cast an int reference refX to refer to a float variable f.

References are not convertible between different types, and the types of the variables being referenced are different (int vs float). Casting refX to refer to f would be like trying to make a square peg fit into a round hole - the memory layouts and types are incompatible.

The compiler will error, something like: "cannot bind 'float' lvalue to 'int' reference".

To reference the float variable f, you need to create a new float reference, like: float& refF = f;.

#include <iostream>

using namespace std;

int main() {

int x = 5;

float f = 3.14;

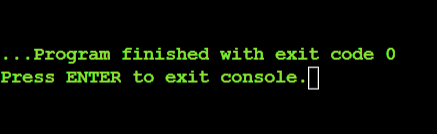
int& refX = x;

float& refF = f;

return 0;

}

**Output:**



**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.**

**Implicit Casting:**

#include <iostream>

using namespace std;

int area\_int(int w, int h) {

return w \* h;

}

double area\_double(double w, double h) {

return w \* h;

}

int main() {

int width = 5;

int height = 3;

cout << "Area (int): " << area\_int(width, height) << endl;

double width\_double = 5.5;

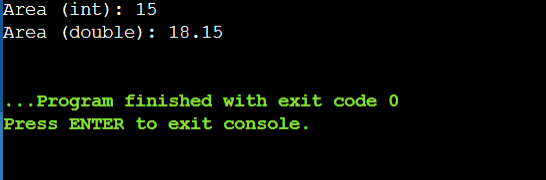
double height\_double = 3.3;

cout << "Area (double): " << area\_double(width\_double, height\_double) <<endl;

return 0;

}

**Output:**



**Explicit Casting:**

#include <iostream>

using namespace std;

int area\_int\_explicit(int w, int h) {

return static\_cast<int>(w \* h);

}

double area\_double\_explicit(double w, double h) {

return static\_cast<double>(w \* h);

}

int main() {

int width = 5;

int height = 3;

cout << "Area (int): " << area\_int\_explicit(width, height) <<endl;

double width\_double = 5.5;

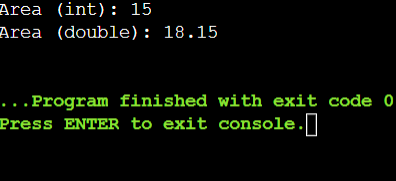
double height\_double = 3.3;

cout << "Area (double): " << area\_double\_explicit(width\_double, height\_double) <<endl;

return 0;

}

**Output:**



**The implications of using implicit casting are:**

In the area\_int function, if the result of w \* h exceeds the range of int, it will be truncated, leading to incorrect results.

In the area\_double function, the result of w \* h will be promoted to double automatically, which ay lead to precision loss if the original values were integers.

**The implications of using explicit casting are:**

In the area\_int\_explicit function, the result of w \* h is explicitly cast to int, which ensures that the result is truncated to an integer value, avoiding potential precision issues.

In the area\_double\_explicit function, the result of w \* h is explicitly cast to double, which ensures that the result is promoted to a double value, preserving precision.

**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>

using namespace std;

int main() {

double celsius;

cout << "Enter temperature in Celsius: ";

cin >> celsius;

double fahrenheit = (celsius \* 9.0 / 5.0) + 32.0;

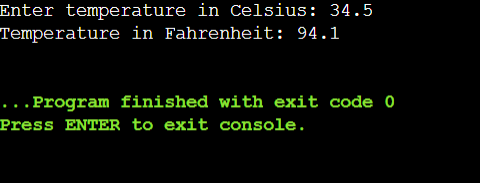
fahrenheit = round(fahrenheit \* 100.0) / 100.0; // Round to two decimal places

cout << "Temperature in Fahrenheit: " << fahrenheit <<endl;

return 0;

}

**Output:**



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

**Unsafe Pointer Arithmetic with Casting**

Unsafe pointer arithmetic involves casting pointers to different types and manipulating them without proper bounds checking

#include <iostream>

using namespace std;

int main() {

int arr[] = {1, 2, 3, 4, 5};

int \*ptr = arr;

char\* charPtr = reinterpret\_cast<char\*>(ptr);

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

cout << "Element " << i << ": " << static\_cast<int>(\*charPtr) << endl;

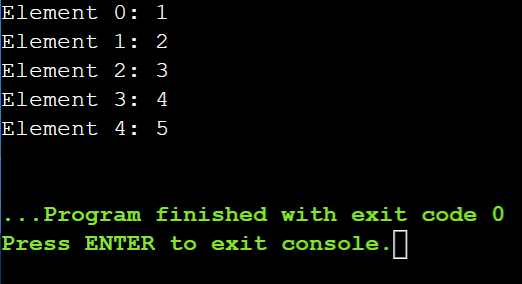
charPtr += sizeof(int);

}

return 0;

}

**Output:**



**Explanation of Unsafe Example:**

In this example, ptr is a pointer to an array of integers.

charPtr is created by casting ptr to char\*, effectively treating the memory as an array of characters rather than integers.

The loop iterates over the memory locations, interpreting them as characters (char), but printing them as integers**.**

**Safe Pointer Arithmetic:**

Safe pointer arithmetic involves using pointers within their intended types and ensuring bounds checking.

#include <iostream>

using namespace std;

int main() {

int arr[] = {1, 2, 3, 4, 5};

int \*ptr = arr;

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

cout << ptr[i] << " ";

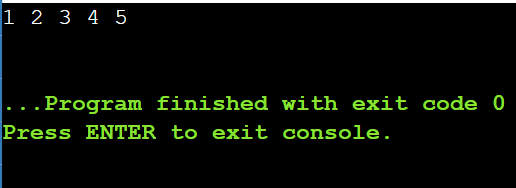
}

cout << endl;

return 0;

}

**Output:**



**Explanation of Safe Example:**

ptr is a pointer to an array of integers (int\*).

The loop iterates over the array using safe pointer arithmetic (ptr[i]), accessing each element of the array.

**Potential Consequences of Unsafe**

**Undefined Behavior:** This code demonstrates undefined behavior because it accesses memory in a manner inconsistent with its original type (int accessed as char). This violates strict aliasing rules and can lead to unpredictable results.

**Memory Corruption:** Incorrect casting and pointer arithmetic can corrupt memory if the program writes to memory locations not intended for the type it assumes**.**

**Portability Issues:** Reliance on such unsafe practices can lead to code that behaves differently on different platforms or compilers**.**

**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()){

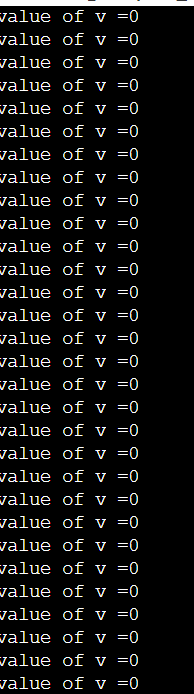
cout<<"value of v ="<<\*v<<endl;

}

return 0;

}

**Output:**



**List:**

#include <iostream>

#include<list>

#include<iterator>

using namespace std;

void showlist(list <int> g)

{

list<int>::iterator it;

for(it=g.begin();it != g.end(); ++it)

cout<<'\t'<<\*it;

cout<<'\n';

}

int main()

{

list<int>gqlist1,gqlist2;

for(int i=0;i<10;++i){

gqlist1.push\_back(i\*2);

gqlist1.push\_back(i\*3);

}

cout<<"\nList1 (gqlist1) is :";

showlist(gqlist1);

cout<<"\nList2 (gqlist2) is :";

showlist(gqlist2);

cout<<"\ngqlist1.front():"<<gqlist1.front();

cout<<"\ngqlist1.back():"<<gqlist1.back();

cout<<"\ngqlist1.pop\_front:";

gqlist1.pop\_front();

showlist(gqlist1);

cout<<"\ngqlist2.pop\_front:";

gqlist2.pop\_front();

showlist(gqlist2);

cout<<"\ngqlist1.reverse():";

gqlist1.reverse();

showlist(gqlist1);

cout<<"\ngqlist2.reverse():";

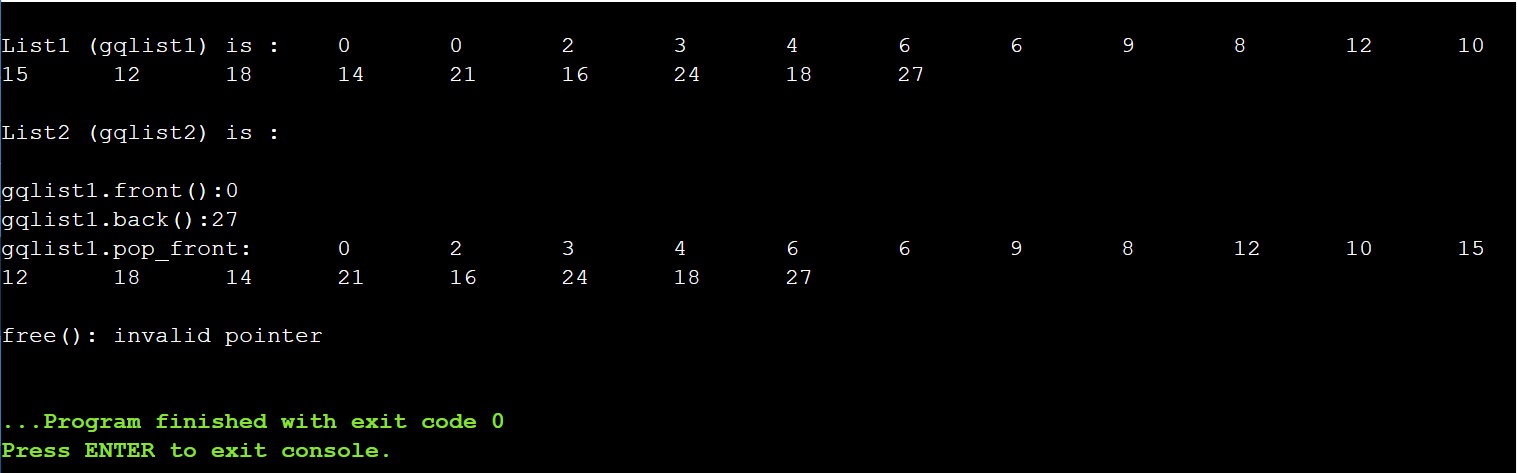
gqlist2.sort();

showlist(gqlist2);

return 0;

}

**Output:**



**Using a Vector-Array Style:**

#include <iostream>

#include <vector>

using namespace std;

void simple\_example() {

const int N = 10;

vector<int> ivec(N);

cout << "Enter " << N << " integers:" << endl;

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

cin >> ivec[i];

}

int ia[N];

for (int j = 0; j < N; ++j) {

ia[j] = ivec[j];

}

cout << "Array elements are: ";

for (int j = 0; j < N; ++j) {

cout << ia[j] << " ";

}

cout << endl;

}

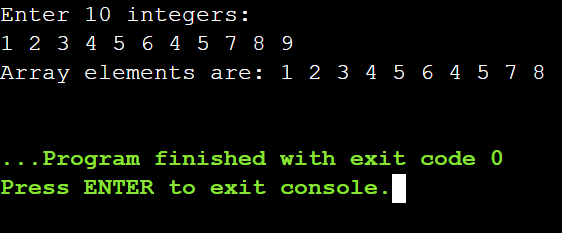
int main() {

simple\_example();

return 0;

}

**Output:**



**Class Exercise-1:**

**Write a program that read integers from the user, sorts them, and print the result.**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

vector<int> numbers;

int num;

cout << "Enter integers (type any non-integer to stop):" << endl;

while (cin >> num) {

numbers.push\_back(num);

}

sort(numbers.begin(), numbers.end());

cout << "Sorted integers: ";

for (int number : numbers) {

cout << number << " ";

}

cout << endl;

return 0;

}

**Output:**

