DAY-11:

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

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

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

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>

#include <string>

int main() {

auto sum = [](int a, int b) {

return a + b;

};

std::cout << "Sum of 3 and 5: " << sum(3, 5) << std::endl;

int x = 5;

auto square = [x]() {

return x \* x;

};

std::cout << "Square of " << x << " is: " << square() << std::endl;

std::string suffix = " World!";

auto add\_prefix = [&suffix](std::string str) {

return "Hello" + suffix;

};

std::cout << add\_prefix("John") << std::endl;

int num = 10;

bool flag = true;

auto conditional\_operation = [num, flag]() {

if (flag) {

return num \* 2;

} else {

return num / 2;

}

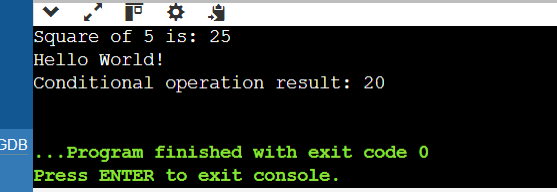
};

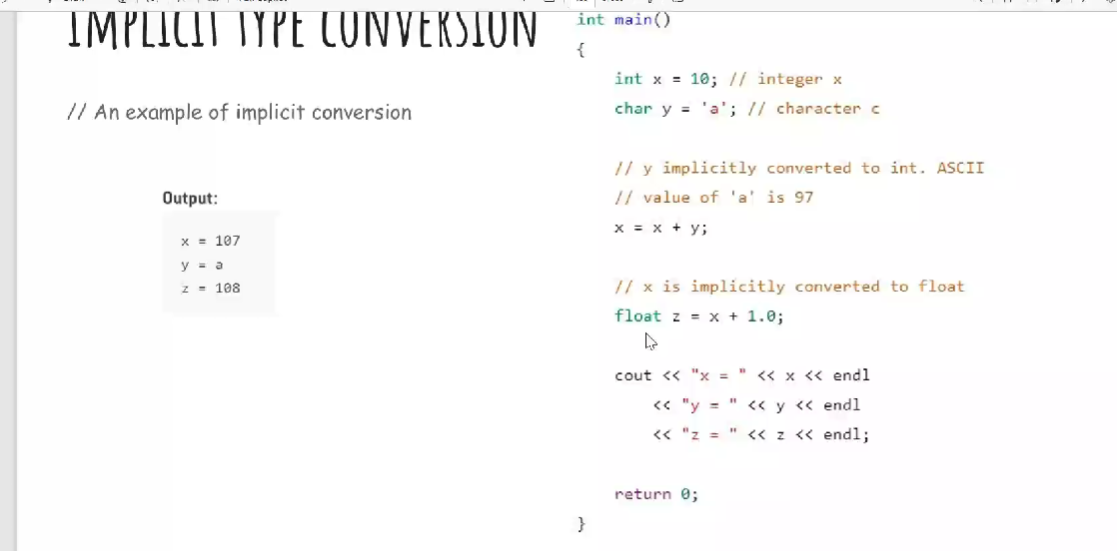
std::cout << "Conditional operation result: " << conditional\_operation() << std::endl;

return 0;

}

Output:





#include<iostream>

using namespace std;

int main()

{

int x =10;

char y='a';

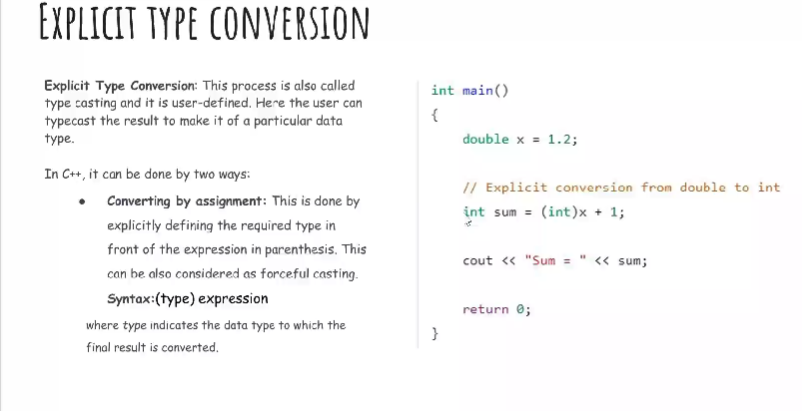
x=x+y;

float z=x+1.0;

cout<<"x =" <<x<<endl<<"y = "<<y<<endl<<"z = "<<z<<endl;

return 0;

}



#include<iostream>

using namespace std;

int main()

{

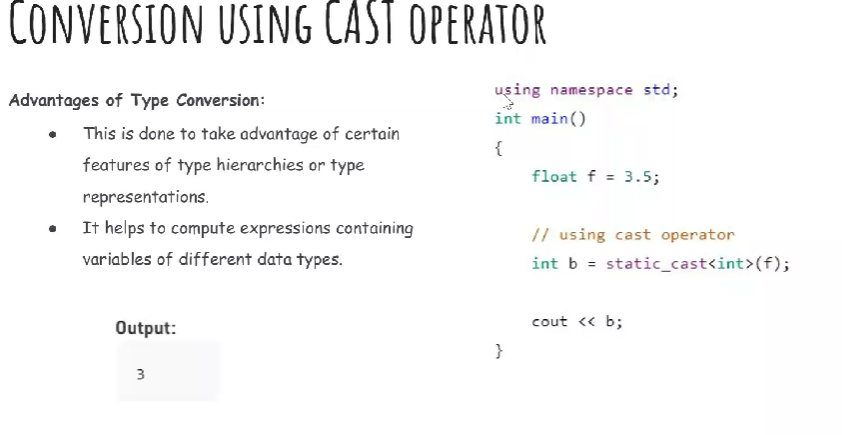
double x=1.2;

int sum=(int)x+1;

cout<<"sum="<<sum;

return 0;

}



#include<iostream>

using namespace std;

int main()

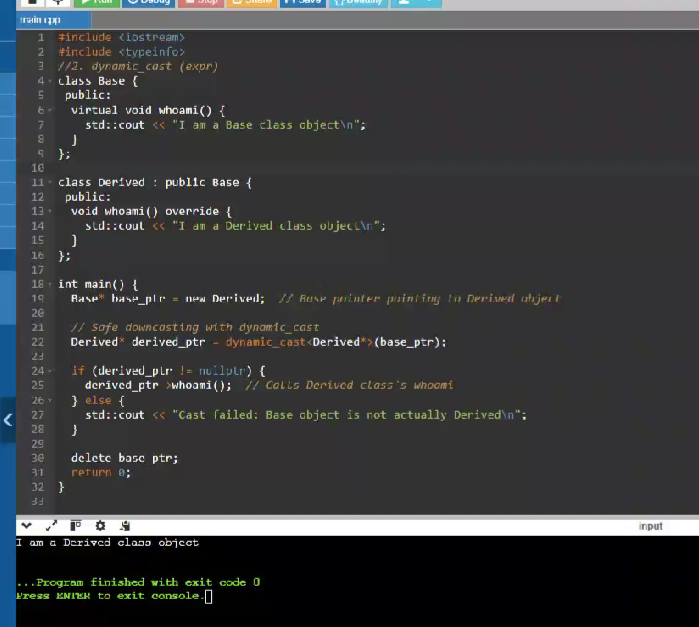
{

float f=3.5;

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

cout<< b;

}



#include<iostream>

#include<typeinfo>

using namespace std;

class Base{

public:

virtual void whoami()

{

cout<<"I am 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;

}

1)

#include <iostream>

int main() {

int value=10;

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

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

return 0;

}

Output:

4013e-44

2) dynamic\_cast (expr):

#include <iostream>

#include<typeinfo>

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=3.14159;

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

derived\_ptr->whoami();

}

else{

cout<<"warning: base object night not be of actually 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 faile: base object is not actually derived\n";

}

delete actual\_derived\_ptr;

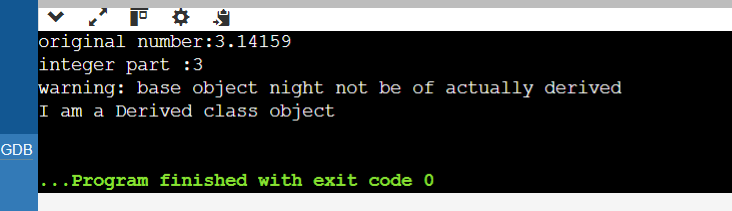
int value=10;

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

return 0;

}

Output:



1)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=12;

float b=3.14;

float result=a/b;

cout<<"result of division a/b:"<<result;

return 0;

}

Output:

result of division a/b:3.82166

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>

using namespace std;

int main(){

int x=256;

char y;

y=static\_cast<char>(x);

cout<<"value of y (char):"<<static\_cast<int>(y)<<endl;

return 0;

}

Output:  
Value of y (char): 0

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>

using namespace std;

int main(){

double d=123.456;

int i =static\_cast<int>(d);

cout<<"value of i:"<<i;

return 0;

}

Output:

value of i:123

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>

using namespace std;

int main(){

int num=5;

int \*ptr=&num;

float\*fptr=reinterpret\_cast<float\*>(ptr);

cout<<"value at fptr:"<<\*fptr;

return 0;

}

Output:

value at fptr:7.00649e-45

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>

using namespace std;

int main()

{

int num=5;

float fval=3.14;

int\* intptr=&num;

float\* floatptr=&fval;

floatptr=reinterpret\_cast<float\*>(intptr);

cout<<"value at floatptr"<<\*floatptr;

return 0;

}

Output:

value at floatptr7.00649e-45

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;

float& refF = reinterpret\_cast<float&>(refX);

std::cout << "x: " << x << std::endl;

std::cout << "refX: " << refX << std::endl;

std::cout << "refF: " << refF << std::endl;

refF = 3.14;

std::cout << "x after modification through refF: " << x << std::endl;

return 0;

}

Output:

x: 10

refX: 10

refF: 1.4013e-44

x after modification through refF: 1078523331

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>

using namespace std;

int main() {

int x = 10;

float f = 3.14;

int& refX = x;

cout << "x: " << x <<endl;

cout << "refX: " << refX <<endl;

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

return 0;

}

Output:

x: 10

refX: 10

f: 3.14

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 calculateArea(int width, int height) {

return width \* height;

}

double calculateArea(double width, double height) {

return width \* height;

}

int main() {

int widthInt = 5, heightInt = 3;

double widthDouble = 5.5, heightDouble = 3.5;

int areaInt = calculateArea(widthInt, heightInt);

std::cout << "Area (int): " << areaInt << std::endl;

double areaDouble = calculateArea(widthDouble, heightDouble);

std::cout << "Area (double): " << areaDouble << std::endl;

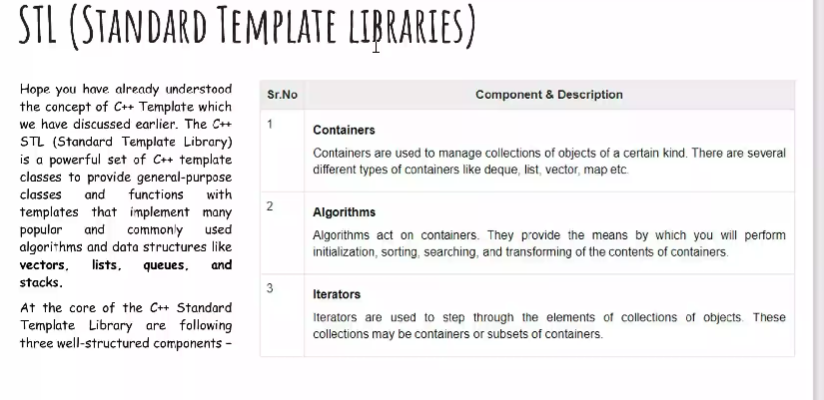
return 0;

}

Output:

Area (int): 15

Area (double): 19.25



LIST CONTINUED:

#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);

gqlist2.push\_front(i\*3);

}

cout<<"\nList 1(gqlist1) is :";

showlist(gqlist1);

cout<<"\nList 2(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\_back():";

gqlist2.pop\_back();

showlist(gqlist2);

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

gqlist1.reverse();

showlist(gqlist1);

cout<<"\ngqlist2.sort();";

gqlist1.sort();

showlist(gqlist2);

return 0;

}

