<https://www.geeksforgeeks.org/catching-base-and-derived-classes-as-exceptions-in-cpp-and-java/>

The destructors are called in reverse order of constructors. Also, after the try block, the destructors are called only for completely constructed objects.

When an object is created inside a try block, destructor for the object is called before control is transferred to catch block.

The statement 'throw;' is used to re-throw an exception. This is useful when a function can handles some part of the exception handling and then delegates the remaining part to the caller. A catch block cleans up resources of its function, and then rethrows the exception for handling elsewhere.

It is compiler error to put catch all block before any other catch. The catch(...) must be the last catch block.

CASTING IN C+++:::\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

A cast is a special operator that forces one data type to be converted into another

<https://www.javatpoint.com/upcasting-and-downcasting-in-cpp>

## **Upcasting**

It is the process to create the derived class's pointer or reference from the base class's pointer or reference, and the process is called Upcasting. It means the upcasting used to convert the reference or pointer of the derived class to a base class. Upcasting is safe casting as compare to downcasting. It allows the public inheritance that implicitly cast the reference from one class to another without an explicit typecast. By default, the upcasting create is-a relationship between the base and derived classes.

Base \*ptr = &derived\_obj;

## **Downcasting**

The Downcasting is an opposite process to the upcasting, which converts the base class's pointer or reference to the derived class's pointer or reference. It manually cast the base class's object to the derived class's object, so we must specify the explicit typecast. The downcasting does not follow the **is- a** relation in most of the cases. It is not safe as upcasting. Furthermore, the derived class can add new functionality such as; new data members and class member's functions that use these data members. Still, these functionalities could not apply to the base class.

Derived \*d\_ptr = &b\_obj;

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Parent
4. {
5. **public**:
6. **void** base()
7. {
8. cout << " It is the function of the Parent class "<< endl;
9. }
10. };
12. **class** Child : **public** Parent
13. {
14. **public**:
15. **void** derive()
16. {
17. cout << " it is the function of the Child class " <<endl;
18. }
19. };
21. **int** main ()
22. {
23. Parent pobj; // create Parent's object
24. Child \*cobj; // create Child's object
26. // explicit type cast is required in downcasting
27. cobj = (Child \*) &pobj;
28. cobj -> derive();
30. **return** 0;
31. }

**Output**

It is the function of the Child class

Upcasting and downcasting in c++

#include <iostream>

**using** **namespace** std;

**class** Parent {

**private**:

**int** id;

**public**:

**void** showid ()

        {

            cout << " I am in the Parent class " << endl;

            }

};

**class** Myson : **public** Parent {

**public**:

**void** disp ()

        {

            cout << " I am in the Myson class " << endl;

        }

};

**int** main ( **int** argc, **char** \* argv[])

{

    // create object of the Parent class

    Parent par\_obj;

    // create object of the Myson class

    Myson my\_obj;

    // upcast - here upcasting can be done implicitly

    Parent \*ptr1 = &my\_obj; // base class's reference the derive class's object

    // downcast - here typecasting is done explicitly

    Myson \*ptr2 = (Myson \*) &par\_obj;

    // Upcasting is safe:

    ptr1->showid();

    ptr2->showid();

    // downcasting is unsafe:

    ptr2->disp();

    getchar();

**return** 0;

}  **Output**

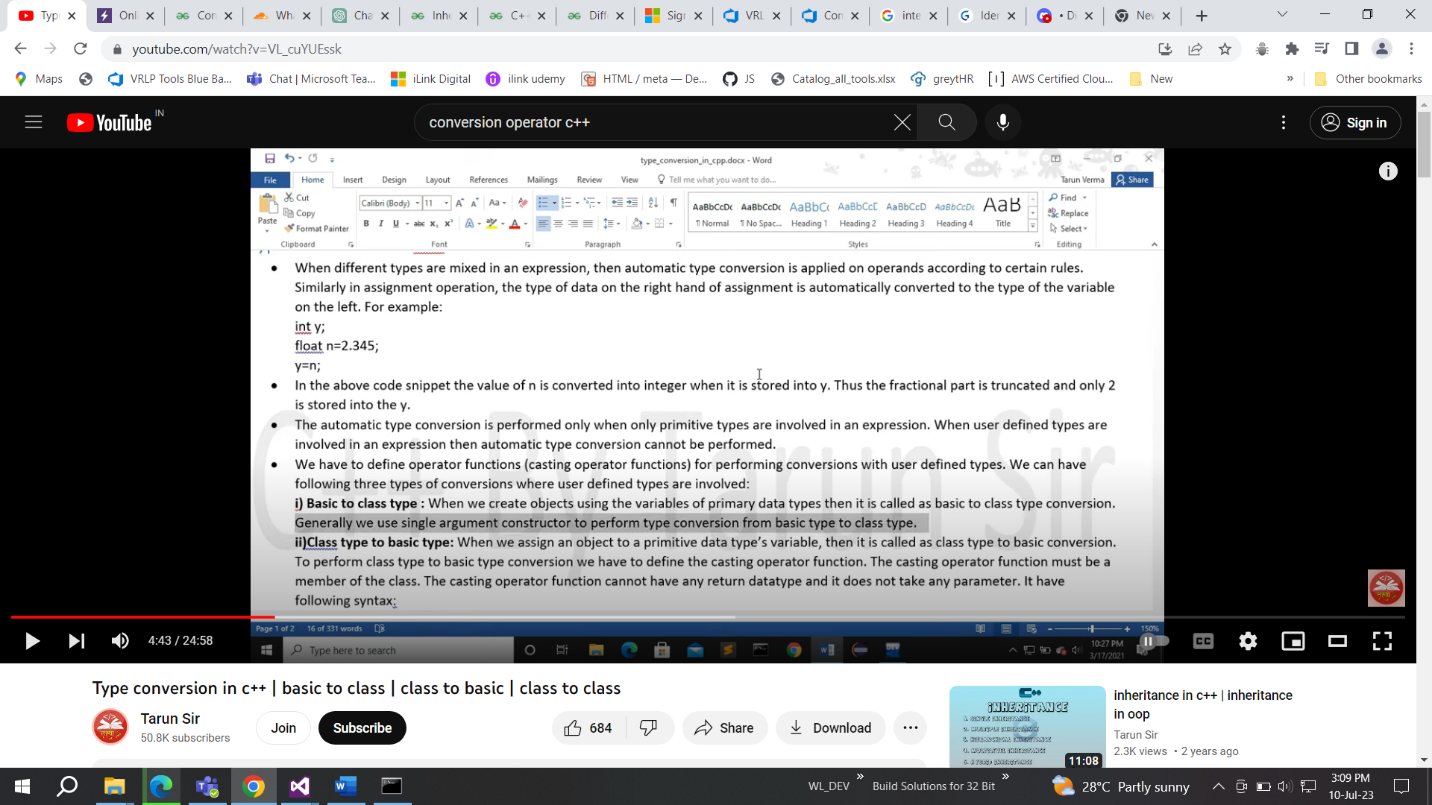
Iam in the Parent class

I am in the Parent class

I am in the Myson class

TYPE CONVERSION IN C++

1. Basic to class:



Eg:

#include <iostream>

using namespace std;

class number

{

private:

int n;

public:

number(int x):n(x) {}

void print()

{

cout<<"the value of x is:"<<n<<endl;

}

};

int main()

{

number n1 = 34;

n1.print();

}

2. Class to basic:

When we want to assign an object to any proimitive data type we need a conversion operator IN base class:

#include <iostream>

using namespace std;

class number

{

private:

int n;

public:

number(int x):n(x) {}

operator double()

{

return n;

}

void print()

{

cout<<"the value of x is:"<<n<<endl;

}

};

int main()

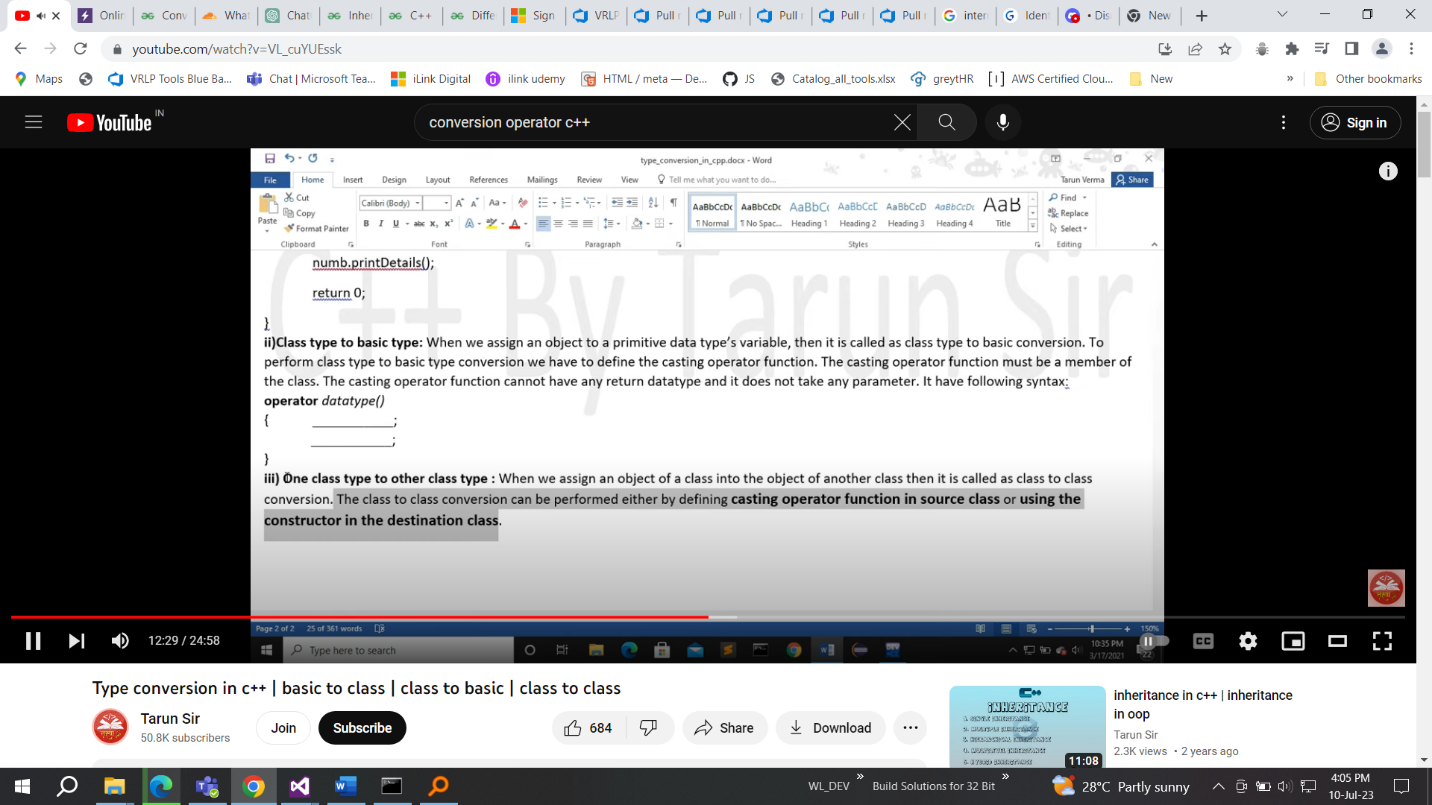
{

number n1 = 34;

int y = n1;

cout<<y;

}

3) class to class

Conversion operator:

Sometimes we need to convert some concret typeobjects to some other types objects or some primitive data types . to make this conversion we use

conversion operator.

#include <iostream>

using namespace std;

class triangle;

class rectangle

{

private:

int base, height;

float area;

public:

rectangle(int a, int b) : base(a), height(b) {}

void print()

{

area = base \* height;

cout << "area of rectangle is: " << area << endl;

}

operator traingle(); // Declaration of the conversion operator

};

class traingle

{

private:

int base, height;

float area;

public:

traingle(int a, int b) : base(a), height(b) {}

void print()

{

area = 0.5 \* base \* height;

cout << "area of triangle is: " << area << endl;

}

};

// Definition of the conversion operator

rectangle::operator traingle()

{

traingle temp(base, height);

return temp;

}

int main()

{

rectangle r(3, 5);

traingle t = r; // Conversion from rectangle to triangle

r.print(); // Output: area of rectangle is: 15

t.print(); // Output: area of triangle is: 7.5

cout << "Hello World";

return 0;

}

A conversion constructor is a single-parameter constructor that is declared without the function specifier explicitly. The compiler uses conversion constructors to convert objects from the type of the first parameter to the type of the conversion constructor’s class.

#include <iostream>

using namespace std;

class traingle

{

private:

int base, height;

float area;

public:

traingle(int a, int b) : base(a), height(b) {}

void print()

{

area = 0.5 \* base \* height;

cout << "area of triangle is: " << area << endl;

}

int getbase()

{

return base;

}

int getheight()

{

return height;

}

};

class rectangle

{

private:

int base, height;

float area;

public:

void print()

{

area = base \* height;

cout << "area of rectangle is: " << area << endl;

}

rectangle(traingle t)

{

base = t.getbase();

height = t.getheight();

}

};

int main()

{

traingle t(3,4);

rectangle r = t; // Conversion from rectangle to triangle

r.print(); // Output: area of rectangle is: 15

t.print(); // Output: area of triangle is: 7.5

cout << "Hello World";

return 0;

}

--------------------------------------------------------------------------------------------------

<https://www.geeksforgeeks.org/use-of-explicit-keyword-in-cpp/>

<https://www.geeksforgeeks.org/g-fact-35/>

explicit keyword in c++

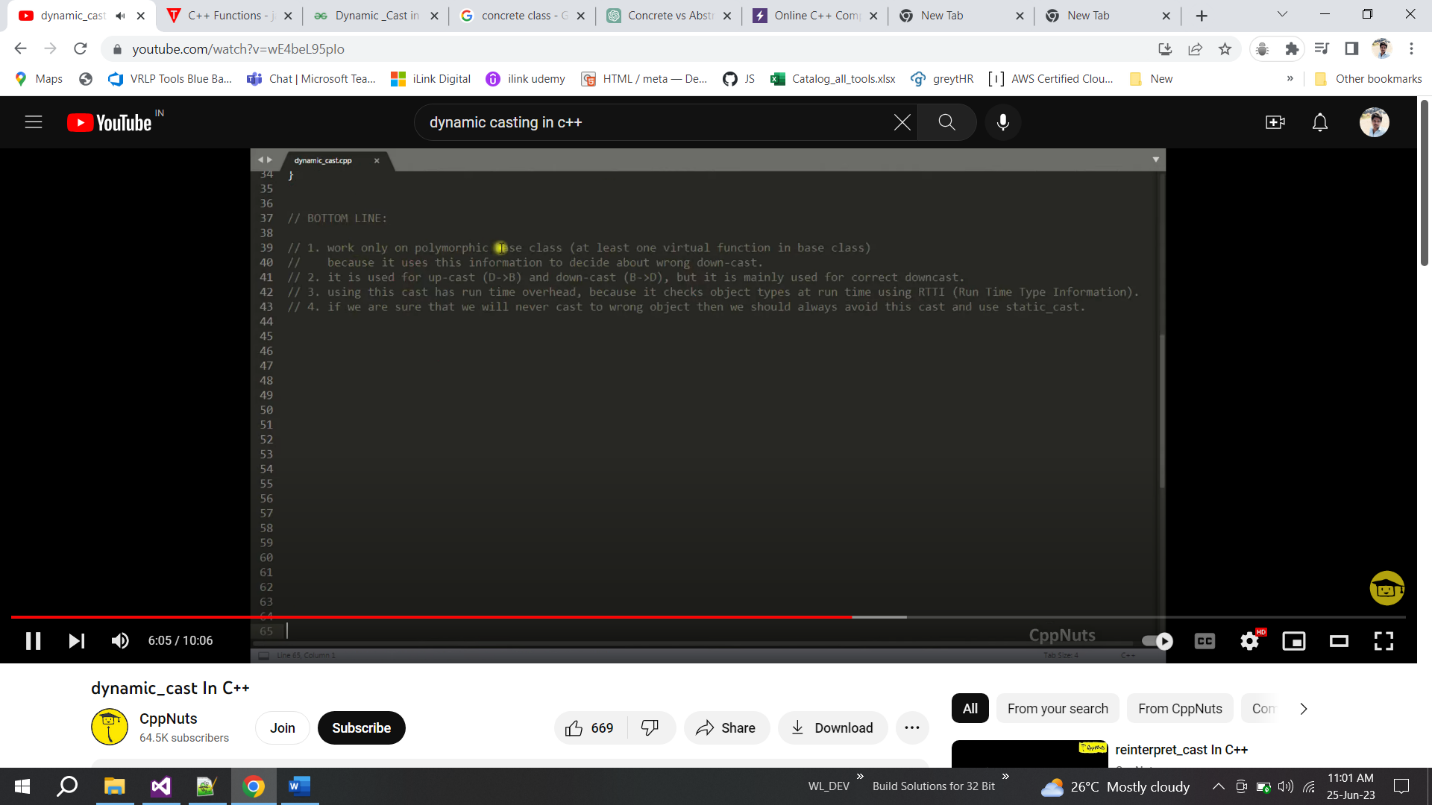
CASTING::->

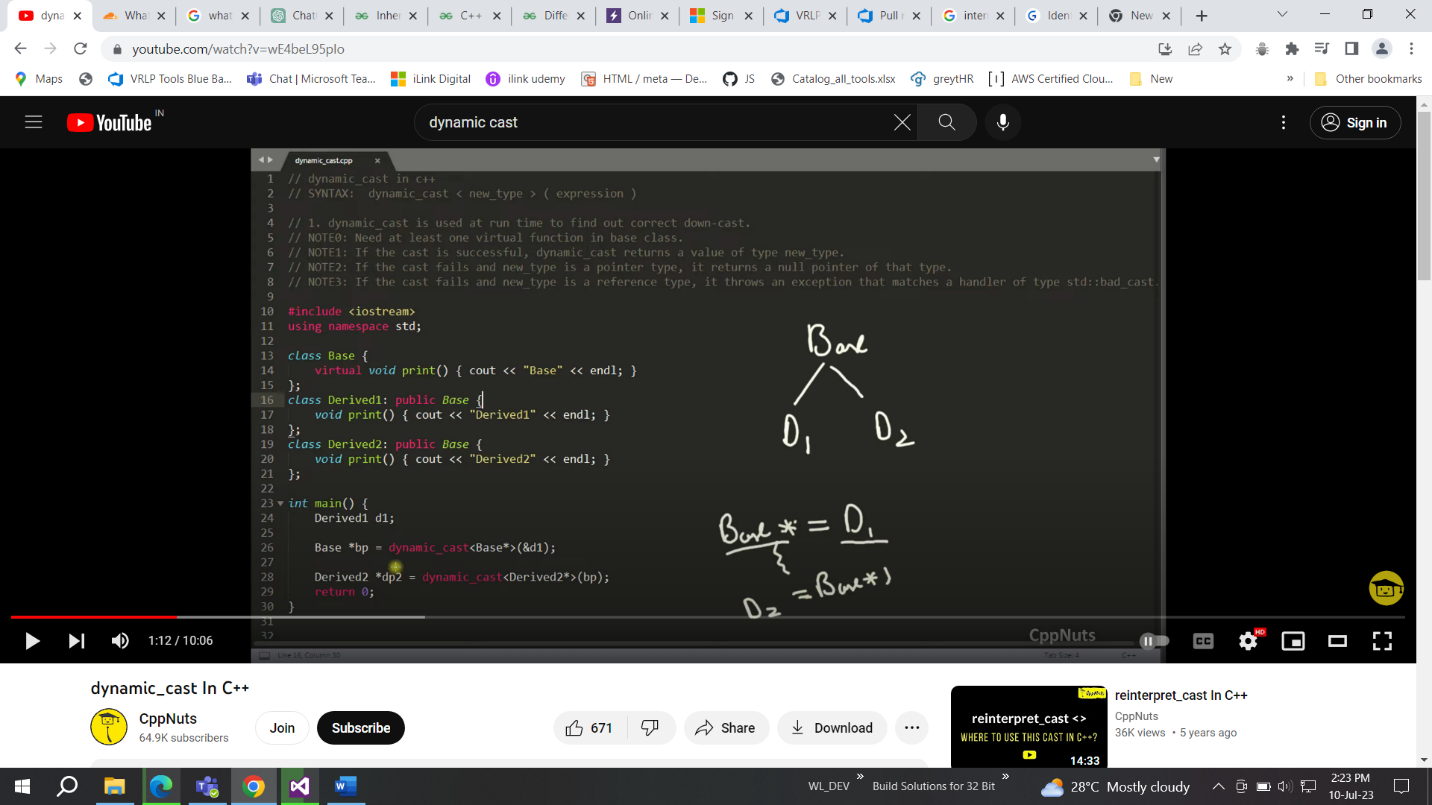
A Cast operator is a **unary operator** which forces one data type to be converted into another data type.

<https://www.geeksforgeeks.org/static_cast-in-cpp/>

**Dynamic Cast:**A cast is an operator that converts data from one type to another type. In C++, dynamic casting is mainly used for safe downcasting at run time. To work on **dynamic\_cast** there must be one virtual function in the base class. A **dynamic\_cast** works only polymorphic base class because it uses this information to decide safe downcasting.

Dynamic cast is used at run time to find correct down cast:





Static cast is a compile time cast and dynamic cast is a run time cast

STATIc CAST:

This is the simplest type of cast that can be used. It is a **compile-time cast**. It does things like implicit conversions between types (such as int to float, or pointer to void\*), and it can also call explicit conversion functions.

#include <iostream>

**using** **namespace** std;

// Driver code

**int** main()

{

**float** f = 3.5;

    // Implicit type case

    // float to int

**int** a = f;

    cout << "The Value of a: " << a;

    // using static\_cast for float to int

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

    cout << "\nThe Value of b: " << b;

}

### static\_cast for Inheritance in C++

#include <iostream>

**using** **namespace** std;

**class** Base

{};

**class** Derived : **public** Base

{};

// Driver code

**int** main()

{

  Derived d1;

  // Implicit cast allowed

  Base\* b1 = (Base\*)(&d1);

  // upcasting using static\_cast

  Base\* b2 = **static\_cast**<Base\*>(&d1);

**return** 0;

}

In the above example, we inherited the base class as public. What happens when we inherit it as private?

It gives error:

### static\_cast to Cast ‘to and from’ Void Pointer

// C++ program to demonstrate

// static\_cast to cast 'to and

// from' the void pointer

#include <iostream>

using namespace std;

// Driver code

int main()

{

int i = 10;

void\* v = static\_cast<void\*>(&i);

int\* ip = static\_cast<int\*>(v);

cout << \*ip;

return 0;

}

CONST\_CAST:

const\_cast is used to cast away the constness of variables. Following are some interesting facts about const\_cast.

**2)** const\_cast can be used to pass const data to a function that doesn’t receive const. For example, in the following program fun() receives a normal pointer, but a pointer to a const can be passed with the help of const\_cast.

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** fun(**int**\* ptr)  {  **return** (\*ptr + 10);  }    **int** main(**void**)  {  **const** **int** val = 10;  **const** **int** \*ptr = &val;  **int** \*ptr1 = **const\_cast** <**int** \*>(ptr);      cout << fun(ptr1);  **return** 0;  } |

 const\_cast is considered safer than simple type casting. It’safer in the sense that the casting won’t happen if the type of cast is not same as original object. For example, the following program fails in compilation because ‘int \*’ is being typecasted to ‘char \*’

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main(**void**)  {  **int** a1 = 40;  **const** **int**\* b1 = &a1;  **char**\* c1 = **const\_cast** <**char** \*> (b1); // compiler error      \*c1 = 'A';  **return** 0;  } |

REINTERPRET\_CAST:

* It is used to convert a pointer of some data type into a pointer of another data type, even if the data types before and after conversion are different.
* It does not check if the pointer type and data pointed by the pointer is same or not.

In C++, **reinterpret\_cast** is a type of casting operator that is used for low-level casting between pointer types and data types. It is a powerful and potentially dangerous cast that should be used with caution. The main use cases for **reinterpret\_cast** include:

1. **Pointer Type Conversion:**
   * **reinterpret\_cast** can be used to convert pointers between unrelated types. This cast is typically used when you need to interpret the binary representation of an object as if it were of a different type.

int intValue = 42;

double\* doublePtr = reinterpret\_cast<double\*>(&intValue);

**Pointer to Void Pointer:**

* **reinterpret\_cast** can be used to convert a pointer to any type to a **void** pointer (**void\***). This is commonly used in scenarios where you want a generic pointer type.

int intValue = 42;

void\* voidPtr = reinterpret\_cast<void\*>(&intValue);

**Integer to Pointer and Vice Versa:**

* **reinterpret\_cast** can be used to convert between integer types and pointers. This is often used in low-level programming or scenarios where you need to store a pointer as an integer or vice versa.

int intValue = 42;

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

* reinterpret\_cast is a very special and dangerous type of casting operator. And is suggested to use it using proper data type i.e., (pointer data type should be same as original data type).
* It can typecast any pointer to any other data type.
* \*It is used when we want to work with bits.
* If we use this type of cast then it becomes a non-portable product. So, it is suggested not to use this concept unless required.
* It is only used to typecast any pointer to its original type.
* Boolean value will be converted into integer value i.e., 0 for false and 1 for true.

// CPP code to illustrate using structure

#include <bits/stdc++.h>

using namespace std;

// creating structure mystruct

struct mystruct {

int x;

int y;

char c;

bool b;

};

int main()

{

mystruct s;

// Assigning values

s.x = 5;

s.y = 10;

s.c = 'a';

s.b = true;

// data type must be same during casting

// as that of original

// converting the pointer of 's' to,

// pointer of int type in 'p'.

int\* p = reinterpret\_cast<int\*>(&s);

cout << sizeof(s) << endl;

// printing the value currently pointed by \*p

cout << \*p << endl;

// incrementing the pointer by 1

p++;

// printing the next integer value

cout << \*p << endl;

p++;

// we are casting back char \* pointed

// by p using char \*ch.

char\* ch = reinterpret\_cast<char\*>(p);

// printing the character value

// pointed by (\*ch)

cout << \*ch << endl;

ch++;

/\* since, (\*ch) now points to boolean value,

so it is required to access the value using

same type conversion.so, we have used

data type of \*n to be bool. \*/

bool\* n = reinterpret\_cast<bool\*>(ch);

cout << \*n << endl;

// we can also use this line of code to

// print the value pointed by (\*ch).

cout << \*(reinterpret\_cast<bool\*>(ch));

return 0;

}

**Why downcasting is needed?**

Downcasting is the process of casting a pointer or reference from a base class type to a derived class type. It is needed in C++ in scenarios where you have a base class pointer or reference that actually points to an object of a derived class, and you want to access the specific features or members of that derived class.

Here are some common situations where downcasting is needed and why it is done in C++:

1. **Polymorphism:**
   * Downcasting is often associated with polymorphism, where you have a base class with virtual functions, and derived classes that provide specific implementations.
   * When you use polymorphism, you might have a collection of objects through pointers or references to the base class. Downcasting allows you to access the derived class-specific functionality.
2. **Access to Derived Class Features:**
   * When you have a pointer or reference to a base class, you can only access the members and functions of the base class.
   * Downcasting allows you to access the additional features or behavior introduced by the derived class.
3. **Event Handling and Callbacks:**
   * In systems with event handling or callback mechanisms (e.g., GUI frameworks), base class pointers are often used to manage a collection of objects.
   * Downcasting is then used to identify the actual derived class type of an object when a specific event occurs.

#include <iostream>

class Animal {

public:

virtual void makeSound() const {

std::cout << "Generic animal sound\n";

}

};

class Dog : public Animal {

public:

void makeSound() const override {

std::cout << "Bark\n";

}

void wagTail() const {

std::cout << "Wagging tail\n";

}

};

int main() {

Animal\* animalPtr = new Dog;

// Attempting downcast

Dog\* dogPtr = dynamic\_cast<Dog\*>(animalPtr);

if (dogPtr) {

// Downcast successful, access Dog-specific features

dogPtr->makeSound(); // Access overridden function in Dog

dogPtr->wagTail(); // Access Dog-specific function

} else {

std::cout << "Downcast failed\n";

}

delete animalPtr;

return 0;

}

Why upcasting is needed?

Upcasting in C++ is the process of casting a pointer or reference from a derived class type to a base class type. It is needed and done for several reasons, mainly related to the concepts of polymorphism, flexibility, and abstraction. Here are some key reasons why upcasting is used in C++:

1. **Polymorphism:**
   * Upcasting is closely related to polymorphism, where a base class pointer or reference can be used to point to objects of its derived classes.
   * It allows treating objects of different derived classes as instances of the common base class, facilitating a uniform interface.
2. **Common Interface:**
   * Upcasting enables working with objects through a common interface (base class) even if they are instances of different derived classes.
   * This common interface allows you to use a consistent set of functions or methods provided by the base class.

**Flexibility and Extensibility:**

* Upcasting makes the code more flexible and extensible by allowing you to add new derived classes without affecting existing code that interacts with the base class.

#include <iostream>

class Shape {

public:

virtual void draw() const {

std::cout << "Drawing a generic shape\n";

}

};

class Circle : public Shape {

public:

void draw() const override {

std::cout << "Drawing a circle\n";

}

};

class Square : public Shape {

public:

void draw() const override {

std::cout << "Drawing a square\n";

}

};

int main() {

Circle circle;

Square square;

Shape\* shapePtr = nullptr;

// Upcasting: base class pointer to derived class object

shapePtr = &circle;

shapePtr->draw(); // Calls draw() from Circle

shapePtr = &square;

shapePtr->draw(); // Calls draw() from Square

return 0;

}