DAY 06

C++ casts

Implicit cast & explicit cast :

- Cast is a way to explicitly convert a value from one data type to another data type. There are two types of casts in C++:

1. **Implicit Cast:** Implicit casts are performed by the compiler automatically. This happens when the compiler can safely convert one data type to another without losing information. For example, converting an int to a float is an implicit cast because the float data type can represent all the values of the int data type without losing information.

```
int x = 10;
float y = x;            // Implicit cast from int to float
```

2. **Explicit Cast:** Explicit casts are performed by the programmer by using the casting operators. These operators tell the compiler to convert a value from one data type to another data type. There are four types of explicit casts in C++:

- **static_cast:** This is used for non-polymorphic conversions between types such as converting an int to a float or a pointer to a derived class to a pointer to its base class.
- **dynamic_cast:** This is used for polymorphic conversions between types, such as converting a pointer to a base class to a pointer to a derived class.
- **const_cast:** This is used to remove the constness of an object or pointer.
- **reinterpret_cast:** This is used to convert a pointer to one type to a pointer to another type, even if the resulting pointer does not point to a valid object of the destination type.

```
double d = 3.14159;
int i = static_cast<int>(d);        // Explicit cast from double to int
```

The difference between c_style ans static cast :

- C-style casts and static_cast are two different ways to perform type casting in C++.
- **C-style casts** are a general-purpose type casting operator that allows you to perform implicit conversions between types. It can cast any type to any other type. The syntax for C-style casting is to enclose the type you want to cast to within parentheses and place it before the value you want to cast

```
int i = 10;
double d = (double) i;
```

- C-style casting can be dangerous because it can perform unintended conversions, and it's difficult to know exactly what is happening when it's used.

- **Static_cast** is a type of explicit type casting that allows you to convert one type to another in a safer and more predictable way. The syntax for static_cast is similar to C-style casting, but it requires that you specify the type of conversion you want to perform

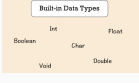
```
int i = 10;
double d = static_cast<double>(i);
```

Static_cast :

- Static_cast is a type of type casting operator that is used to convert one data type to another data type in a static or compile-time manner.
- The syntax of static_cast is as follows: **static_cast<new_type>(expression) .**

Some common cases where static_cast can be useful :

- Converting between compatible data types: static_cast can be used to convert between built-in data types :




```
int i = 10;
float f = static_cast<float>(i);        // convert i to float
```

- Converting pointers to void to pointers to other types: static_cast can be used to convert a pointer to void to a pointer to any other data type.

```
int i = 10;
void* v = &i;
int* p = static_cast<int*>(v);        // convert void* to int*
```

Dynamic_cast :

- Dynamic_cast is a type of casting operator that allows you to safely convert a pointer or reference to a base class to a pointer or reference of a derived class.
- It is called "dynamic" because the type of the object pointed to or referred to is determined at runtime, rather than at compile-time. This is in contrast to static_cast, which performs the conversion based solely on the types involved, and is therefore determined at compile-time .
- The syntax for dynamic_cast is as follows: **dynamic_cast<DerivedType*>(base_pointer);**
- Here, DerivedType is the type of the derived class that you want to convert the pointer or reference to, and base_pointer is the pointer or reference to the base class object that you want to convert.
- dynamic_cast returns a null pointer if the conversion is not possible (i.e., if the object pointed to by base_pointer is not actually an object of type DerivedType or a subclass of DerivedType). Otherwise, it returns a pointer or reference to the derived class object.

Some common cases where dynamic cast can be useful :

- **Upcasting and Downcasting:** Upcasting is when you cast a pointer or reference to a derived class object to a pointer or reference to its base class. This is always safe and can be done using a regular C++ cast. Downcasting is when you cast a pointer or reference to a base class object to a pointer or reference to a derived class. This can be unsafe, because the object may not actually be of the derived class type. In this case, you can use dynamic_cast to perform the conversion safely.

- **Checking Object Type:** You can use dynamic_cast to check whether a given object is of a certain type. If the object is not of the specified type, dynamic_cast will return a null pointer. This is useful when you want to perform a certain operation on objects of a specific type, and need to verify the type of the object first.

```
class Base {
public:
    virtual void print() {
        std::cout << "This is the Base class.\n";
    }
};

class Derived : public Base {
public:
    void print() override {
        std::cout << "This is the Derived class.\n";
    }
};
```

```
int main() {
    // Upcasting
    Derived d;
    Base* b = &d; // Upcast the Derived object to a Base pointer
    b->print(); // Calls Derived::print()

    // Downcasting
    Base* b2 = new Derived();
    Derived* d2 = dynamic_cast<Derived*>(b2); // Downcast the Base pointer to a Derived pointer
    if (d2) {
        d2->print(); // Calls Derived::print()
    } else {
        std::cout << "Dynamic cast failed.\n";
    }

    delete b2;
    return 0;
}
```

Reinterpret_cast :

- reinterpret_cast is a type of casting operator in C++ that allows you to convert a pointer or reference of one type to another type without performing any conversion checking or adding any additional overhead. It is the most powerful and dangerous of all C++ casting operators and should be used with great caution.

- Here is an example of using reinterpret_cast to convert a pointer to an integer:

```
int* ptr = new int(42);
uintptr_t addr = reinterpret_cast<uintptr_t>(ptr);
```

- In this example, reinterpret_cast is used to convert the pointer ptr to an unsigned integer type uintptr_t. The resulting value addr will contain the memory address of the int object that ptr points to.

Some common cases where reinterpret can be useful :

- Casting between unrelated pointer types: reinterpret_cast can be used to cast a pointer of one type to a pointer of another type that is not related by inheritance. This is useful in situations where you need to reinterpret the bits of the object as a different type.
- Type punning: reinterpret_cast can be used to reinterpret the bits of an object as a different type. This is often used in low-level programming to implement features like serialization, deserialization, and byte-swapping.
- Converting between pointers and integers: reinterpret_cast can be used to convert a pointer to an integer type, or vice versa. This can be useful in situations where you need to store a pointer as an integer, or retrieve a pointer from an integer.

Const_cast :

- const_cast is a type of C++ cast that allows a program to temporarily remove the constness of an object or pointer. This means that a program can cast a const object or pointer to a non-const object or pointer, allowing the program to modify the object or pointer.
- For example, suppose you have a const variable const int x = 5;. If you try to modify this variable directly, the compiler will produce an error because it is const. However, you can use const_cast to remove the constness temporarily and modify the variable:

```
const int x = 5;
int* y = const_cast<int*>(&x);
*y = 10;
```