Casting

quarta-feira, 27 de novembro de 2024 07:45

1. `static_cast`

   Used for well-defined and non-polymorphic type conversions (e.g., converting an `int` to a `float`, or a pointer of a base class to a derived class, assuming it is safe).

   cpp Copy code

   ```cpp
   int a = 42;
   float b = static_cast<float>(a); // Convert int to float
   ```

2. `dynamic_cast`

   Used for safe downcasting in polymorphic hierarchies (requires at least one `virtual` function in the base class). If the cast is not valid, it returns `nullptr` for pointers or throws an exception for references.

   Base → Derived

   cpp Copy code

   ```cpp
   class Base { virtual void foo() {} };
   class Derived : public Base {};
   Base* basePtr = new Derived;
   Derived* derivedPtr = dynamic_cast<Derived*>(basePtr); // Safe downcast
   ```

3. `const_cast`

   Used to remove or add `const` qualifiers from a variable. It is often used when interfacing with legacy code.

   cpp Copy code

   ```cpp
   const int x = 42;
   int& y = const_cast<int&>(x); // Remove const (use with caution)
   ```

4. `reinterpret_cast`
   Used for low-level, unsafe conversions (e.g., casting between unrelated pointer types). Use this cast sparingly.

   ```cpp
   int a = 42;
   void* ptr = reinterpret_cast<void*>(&a); // Convert int* to void*
   int* b = reinterpret_cast<int*>(ptr);    // Convert void* back to int*
   ```

Why Prefer C++-Style Casts Over C-Style?

C-style casts do not differentiate between the types of casting, making them harder to read and prone to errors:

```cpp
float b = (float)a; // C-style cast, less explicit
```

By using C++-style casts, you:

- **Increase safety**: Invalid casts can fail at runtime or even during compilation.
- **Improve readability**: The intent of the cast is clearer.
- **Reduce ambiguity**: Differentiate between types of conversions.

More like a function and has a
runtime cost (does extra work)

`dynamic_cast` is used for **runtime type checking and safe downcasting** in C++'s polymorphic class hierarchies. It ensures that a cast is valid before allowing it, avoiding undefined behavior.

Requirements for `dynamic_cast`

1. **Polymorphic Base Class**: The base class must have at least one `virtual` function (commonly a virtual destructor).
2. **Pointer or Reference**: Works with pointers or references, not direct objects.

When to Use `dynamic_cast`

- **Safe Downcasting**: To convert a pointer/reference of a base class to a derived class pointer/reference, ensuring the object is of the desired type.

- **Safe Downcasting**: To convert a pointer/reference of a base class to a derived class pointer/reference, ensuring the object is of the desired type.
- **Type Checking**: To verify the type of an object at runtime.

→ Move around base & derived class

→ Base → Derived → way to check if its valid

→ if fails → return nullptr

Yes, you can use `static_cast` to convert from a base class to a derived class without runtime type checking. However, **it is your responsibility to ensure the cast is valid**, as `static_cast` does not perform any runtime checks. Using it incorrectly may lead to **undefined behavior**.

(Global Scope)

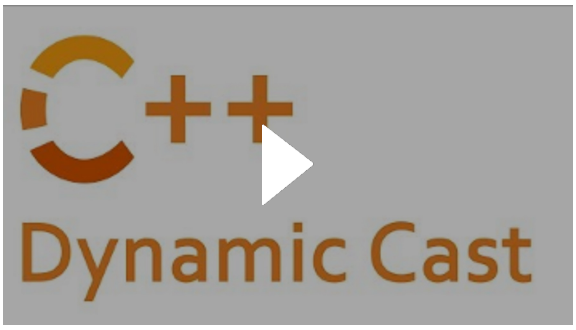
```cpp
class Entity
{
public:
    virtual void PrintName() {}
};

class Player : public Entity
{
};

class Enemy : public Entity
{
};

int main()
{
    Player* player = new Player();
    Entity* actuallyPlayer = player;
    Entity* actuallyEnemy = new Enemy();

    Player* p = dynamic_cast<Player*>(actuallyEnemy);
```

↳ fail → nullptr

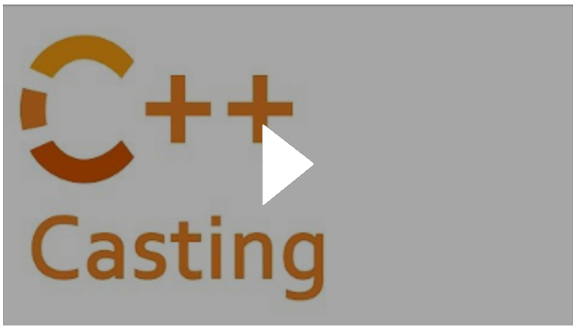
```cpp
    Player* p = dynamic_cast<Player*>(actuallyPlayer);
```

↳ works

[Dynamic Casting in C++](#)



[Casting in C++](#)



Differences Between `dynamic_cast` and `static_cast`

Feature	dynamic_cast	static_cast
Type Checking	Performs runtime type checking.	No runtime type checking.
Safety	Safe (returns `nullptr` or throws `std::bad_cast` if invalid).	Unsafe (undefined behavior if invalid).
Overhead	Slight runtime overhead.	No runtime overhead.
Use Case	Use when type safety is crucial.	Use when you're certain the cast is valid.

The performance difference between `static_cast` and `dynamic_cast` arises because of their fundamentally different mechanisms:

1. **Compile-Time (Static Cast)**:
   - `static_cast` is resolved **entirely at compile-time**.
   - The compiler generates the necessary code for type conversion without any runtime overhead.
   - **Performance**: Minimal; equivalent to direct pointer or reference assignment.
2. **Runtime (Dynamic Cast)**:
   - `dynamic_cast` involves **runtime type checking** to ensure the validity of the cast.
   - It queries the object's **type information** stored in the vtable (if the base class is polymorphic).
   - **Performance**: Slight overhead due to:
     - Accessing the type information (e.g., `std::type_info`).
     - Walking the inheritance hierarchy (especially for complex hierarchies).
   - The exact cost depends on the depth and complexity of the inheritance tree.