**static\_cast**

It can be used for any normal conversion between types, conversions that rely on static (compile-time) type information. This includes any casts between numeric types (for instance : from short to int or from int to float), casts of pointers and references up the hierarchy (upcasting).

**static\_cast** performs no run-time checks and hence no runtime overhead.

1. **int** a = 5, b = 2;
2. **double** result = **static\_cast**<double>(a) / b;

**dynamic\_cast**

It can only be used with pointers and references to objects. It's almost exclusively used for handling polymorphism. It makes sure that the result of the type conversion is valid and**complete** object of the requested class.

1. **class** Base { };
2. **class** Derived : **public** Base { };
4. Base a, \*ptr\_a;
5. Derived b, \*ptr\_b;
7. ptr\_a = **dynamic\_cast**<Base \*>(&b); *// Fine*
8. ptr\_b = **dynamic\_cast**<Derived \*>(&a); *// Fail*

The first **dynamic\_cast** statement will work because we cast from derived class to base. The second **dynamic\_cast** statement will produce a compilation error because base class to derived conversion is not allowed with **dynamic\_cast** **unless the base class is polymorphic (***a polymorphic type has at least one virtual function, declared or inherited***)**.

If a class is polymorphic then **dynamic\_cast** will perform a special check during runtime.

1. **class** Base { **virtual** **void** dummy() {} }; *// polymorphic class*
2. **class** Derived : **public** Base { **int** a; }; *// so is this*
4. Base \*ptr\_a = **new** Derived{};
5. Base \*ptr\_b = **new** Base{};
7. Derived \*ptr\_c = **nullptr**;
8. Derived \*ptr\_d = **nullptr**;
10. ptr\_c = **dynamic\_cast**<Derived \*>(ptr\_a); *// Fine*
11. ptr\_d = **dynamic\_cast**<Derived \*>(ptr\_b); *// ptr\_d will be NULL*
13. *// Check if downcasting succeeded*
14. **if** (ptr\_c != **nullptr**) {
15. *// ptr\_a actually points to a Derived object*
16. }
18. **if** (ptr\_d != **nullptr**) {
19. *// ptr\_b actually points to a Derived object*
20. }

When the **dynamic\_cast** is performed, ptr\_a is pointing to a full object of class Derived, but the pointer ptr\_b points to an object of class Base. This object is an incomplete object of class Derived; thus this cast will result in a null pointer !

*Important:*

1. A null pointer is returned to indicate a failure while dealing with a pointer.
2. When a reference type conversion fails then there will be an exception thrown.

*NOTE:* **dynamic\_cast have a significant runtime overhead.**

**reinterpret\_cast**

It is used for casts that are not type-safe. It converts between types by reinterpreting the underlying bit pattern.

Unlike static\_cast, but like const\_cast, the reinterpret\_cast expression does not compile to any CPU instructions. It is purely a compiler directive which instructs the compiler to treat the sequence of bits (object representation) of expression as if it had the new\_type (that you are casting to).

In short, you tell the compiler, believe me I know what am I doing.

* Between pointers and pointers (even of unrelated classes). All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.
* Between integers and pointers.
* Between function-pointers and function-pointer

However it **can't** modify const qualification.

1. **class** A { */\* ... \*/* };
2. **class** B { */\* ... \*/* };
3. A \*a = **new** A{};
4. B \*b = **reinterpret\_cast**<B\*>(a); *// Fine*
6. **const** **char**\* message = "hello";
7. **int**\* data = **reinterpret\_cast**<**int**\*>(message); *// Fail*

**const\_cast**

The only way to cast away the constness of an object is to use **const\_cast**. But this is not the end of story,

*Important:*

**const\_cast** can be safely used to remove constness from a reference or a pointer used to access a **non-const object**.

1. **int** a = 5; *// NOTE: non-const object*
2. **const** **int**\* pA = &a;
3. \*pA = 10; *// compiler error, pA is a pointer to const int*
4. **int**\* pX = **const\_cast**<**int**\*>(pA); *// cast away constness*
5. \*pX = 10 *// fine and a is now 10*

But if your **actual object is declared const**, that constness can never be removed. Modifying a const object through a non-const access path results in undefined behavior.

1. **const** **int** a = 5; *// NOTE: const object*
2. **const** **int**\* pA = &a;
3. \*pA = 10; *// compiler error, pA is a pointer to const int*
4. **int**\* pX = **const\_cast**<**int**\*>(pA); *// cast away constness*
5. \*pX = 10 *// Free ticket to a long journey of UNDEFINED BEHAVIOR*

The **const\_cast** can not be used to cast to other data-types, as it is possible with the other casts.

1. **int**\* a = **nullptr**;
2. **const** **char**\* ptr = "Hello";
3. a = **const\_cast**<**int**\*>(ptr); *// Fail*
4. a = **reinterpret\_cast**<**int**\*>(ptr); *// Fail, reinterpret\_cast can't cast away const qualifiers*
5. a = **reinterpret\_cast**<**int**\*>(**const\_cast**<**char**\*>(ptr)); *// Fine, as long as you kno*