Explicit type conversions

09 June 2021 07:07

1. In C++, there are 5 different types of casts: C-style casts, static_cast, const_cast, dynamic_cast, and reinterpret_cast.

2. Types of casts in C++
   - `static_cast`
   - `const_cast`
   - `reinterpret_cast`
   - `dynamic_cast`
   - `std::move  -> static_cast<std::remove_reference<decltype(arg)>::type&&>(arg) - conv obj to temporary r-value`
   - `std::forward<T> -> Used in templated functions`
     `rvalue arguments to be passed on as rvalues, and lvalues to be passed on as lvalues, a scheme called "perfect forwarding" - Refer to templates chapter where arg is rvalue reference - can accept l-value ref and r-val reference`
     https://isocpp.org/blog/2018/02/quick-q-whats-the-difference-between-stdmove-and-stdforward#:~:text=std%3A%3Amove%20takes%20an,a%20temporary%20(an%20rvalue).&text=This%20allows%20rvalue%20arguments%20to,by%20lvalue%5Cn%22%3B%20%7D

   - `gsl::narrow_cast<type>` applies a `static_cast`
   - `gsl::narrow<type>` applies a `static_cast`

3. C++ cast outputs
   - an lvalue if new_type is an lvalue reference type
   - An lvalue if new_type is an rvalue reference to function type;
   - an xvalue if new_type is an rvalue reference to object type;
   - a prvalue otherwise.

4. C-style cast
   - Although a C-style cast appears to be a single cast, it can actually perform a variety of different conversions depending on context.

   This can include a
   - static cast
   - const cast or
   - a reinterpret cast

   Not explicit & confusing

5. static_cast - performed compile-time?

   `static_cast < type-name > (expression)`
   It's valid only if type_name can be converted implicitly to the same type that expression has, or vise versa. Otherwise, the type cast is an error. Eg since upcast can be done implicitly, downcast is allowed

   - Also performs what implicit conversions cannot -- pointers/value types
   - Has compile-time checks such as type checks
     Doesn't allow
     - Conversions between different user-defined objects
     - Conversions between pointers of different types
     - Pointer -> value type or vice-versa
     - Cast-away of const/**volatile** i.e. const to non-const
     - Downcast to virtually inherited derived class pointers - due to memory format?

     Allows
     - Conversions between fundamental types -> Int -> char is narrowing conversion.. To allow that use static_cast
     - One pointer to void* or viceversa
     - Upcast or downcast of object pointers
       - Downcast - safely downcast only when it can see class definitions entirely and inheritance is polymorphic
     - Non-const -> const
     - Scoped enum to integral type
     - Lvalue to xvalue -> std::move

   `a = static_cast<T>(p) // Convert type of p into a T if a T can be converted into p's type.`

6. const_cast
   Cast-away const or volatile property of a pointer or reference

7. reinterpret_cast - compile-time doesn't resolve to instruction
   It is purely a compile-time directive which instructs the compiler to treat **expression** as if it had the type **new_type**.
   Basically Reinterpreting underlying bit pattern.

   1. Cast any pointer or integral type to any other pointer or integral type of same size..
   2. Any value of type std::nullptr_t, including nullptr can be converted to any integral type as if it were (void*)0
   3. T1* Objp can be converted to cv T2* Objp.. Eq to
      `static_cast<cv T2*>(static_cast<cv void*>(expression))`

4. An lvalue expression of type `T1` can be converted to reference to another type `T2`. The result is an lvalue or xvalue referring to the same object as the original lvalue, but with a different type.
5. The null pointer value of any pointer type can be converted to any other pointer type, resulting in the null pointer value of that type
6. Conversion of func ptr of one function to another function type..

If ptr types r of different size compiler gives warning.

8. The rule of the thumb for reinterpret_cast:

Never use reinterpret_cast or C-Style casting, if you need to cast pointers, cast them via void*, and only if absolutely necessary use reinterpret_cast - that means, if you really have to reinterpret the data.

9. dynamic_cast<type* or type&>(var) - Polymorphic cast

C++ allows implicit conversion of derived class POINTER OR reference to base class pointer/ref -> UPCASTING

DYNAMIC-CAST helps with SAFE downcasting i.e. base pointer -> derived pointer only if they are polymorphic..
It converts between pointers/references in the same class hierarchy with run-time checks

```
Base *b{ &derived };
Derived *d{ dynamic_cast<Derived*>(b) }; // use dynamic cast to convert Base pointer into Derived pointer
```

Above works because b is actually pointing to a Derived object. Below example fails and dynamic_cast returns nullptr

```
Base *b{ &base };
Derived *d{ dynamic_cast<Derived*>(b) }; // Fail - runtime check of dynamic_cast
```

Always ensure your dynamic casts actually succeeded by checking for a null pointer result.

10. Dynamic_cast failure cases

Also note that there are several cases where downcasting using dynamic_cast will not work:

- With protected or private inheritance.
- In order to work with dynamic_cast, your classes must be polymorphic type i.e. must include at least one virtual method.
- In certain cases involving virtual base classes (see this page for an example of some of these cases, and how to resolve them).

static_cast always allows downcasting. Responsibility of programmer to check ---> use static_cast for upcasting!! -- all implicit upcasts are static_cast actually

11. Dynamic_cast typecasting references will return std::bad_cast during incorrect conversions instead of nullptr

12. A warning about dynamic_cast and RTTI

dynamic_cast operator uses the runtime type information generated from polymorphic classes.

Run-time type information (RTTI) is a feature of C++ that exposes information about an object's data type at runtime. This capability is leveraged by dynamic_cast. Because RTTI has a pretty significant space performance cost, some compilers allow you to turn RTTI off as an optimization. Needless to say, if you do this, dynamic_cast won't function correctly

13. Use auto or {..} initialization to avoiding narrowing

14. gsl::narrow to check narrowing

gsl::narrow cast throws an exception if a narrowing conversion happens

```
i = narrow<int>(d);        // OK: throws narrowing_error
```

15. gsl::narrow_cast - basically static_cast to force narrowing

https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-conversions-and-casts
Narrowing -> Means a conversion or assignment etc that results in loss of precision

Explicitly state u want narrowing/ ask compiler to check for narrowing
    The gsl::narrow_cast performs the narrow cast using static_cast

```
i = narrow_cast<int>(d);   // OK (you asked for it): narrowing: i becomes 7
```

16. F

I

17. F

I

18. F

19. F

I

20. F

I

21. F

I

22. F

I

23. F
    I
24. F
    I
25. D
    I
26. F
    I
27. F
    I
28. F
    I
29. F
30. F
    I
31. F
    I
32. F
    I
33. F
    I
34. F
    I
35. F
    I
36. D
    I
37. F
    I
38. F
    I
39. F
    I
40. F
41. F
    I
42. F
    I
43. F
    I
44. F
    I
45. F
    I
46. F
    I
47. D
    I
48. F
    I
49. F
    I
50. F
    I
51. F
52. F
    I
53. F
    I
54. F
    I
55. F
    I

56. F
I