When should static_cast, dynamic_cast, const_cast, and reinterpret_cast be used?

Asked 16 years ago Modified 11 months ago Viewed 805k times

3162

What are the proper uses of:

- `static_cast`
- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`
- `(type)value` (C-style cast)
- `type(value)` (function-style cast)

How does one decide which to use in which specific cases?

c++ pointers casting c++-faq

Share

Improve this question

Follow

edited Jul 4, 2022 at 21:41

Mateen Ulhaq
27.1k 21 117 152

asked Dec 1, 2008 at 20:11

James Eichele
119k 41 182 214

46 Maybe a good reference here:[How do you explain the differences among static_cast, reinterpret_cast, const_cast, and dynamic_cast to a new C++ programmer?]. – Nan Xiao Jan 8, 2016 at 3:49

4 For some useful concrete examples of using different kind of casts, you can check the first answer on a similar question in [this other topic]. – TeaMonkie Feb 24, 2017 at 10:41

5 You can find really good answers for your question above. But I would like to put one more point here, @e.James "There is nothing that these new c++ cast operators can do and c style cast cannot. These are added more or less for the better code readability." – BreakBadSP Oct 9, 2018 at 7:51

4 @BreakBadSP The new casts are *not* only for better code readability. They are there to make it harder to do do dangerous things, like casting away const or casting pointers instead of their values. static_cast has much less possibilities to do something dangerous than a c style cast! – FourtyTwo Jan 13, 2020 at 12:50

12 Answers

Sorted by: Highest score (default)

3103

`static_cast`

`static_cast` is the first cast you should attempt to use. 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 (or implicit ones). In many cases, explicitly stating `static_cast` isn't necessary, but it's important to note that the `T(something)` syntax is equivalent to `(T)something` and should be avoided (more on that later). A `T(something, something_else)` is safe, however, and guaranteed to call the constructor.

`static_cast` can also cast through inheritance hierarchies. It is unnecessary when casting upwards (towards a base class), but when casting downwards it can be used as long as it doesn't cast through `virtual` inheritance. It does not do checking, however, and it is undefined behavior to `static_cast` down a hierarchy to a type that isn't actually the type of the object.

`const_cast`

`const_cast` can be used to remove or add `const` to a variable; no other C++ cast is capable of removing it (not even `reinterpret_cast`). It is important to note that modifying a formerly `const` value is only undefined if the original variable is `const`; if you use it to take the `const` off a reference to something that wasn't declared with `const`, it is safe. This can be useful when overloading member functions based on `const`, for instance. It can also be used to add `const` to an object, such as to call a member function overload.

`const_cast` also works similarly on `volatile`, though that's less common.

`dynamic_cast`

`dynamic_cast` is exclusively used for handling polymorphism. You can cast a pointer or reference to any polymorphic type to any other class type (a polymorphic type has at least one virtual function, declared or inherited). You can use it for more than just casting downwards – you can cast sideways or even up another chain. The `dynamic_cast` will seek out the desired object and return it if possible. If it can't, it will return `nullptr` in the case of a pointer, or throw `std::bad_cast` in the case of a reference.

`dynamic_cast` has some limitations, though. It doesn't work if there are multiple objects of the same type in the inheritance hierarchy (the so-called 'dreaded diamond') and you aren't using `virtual` inheritance. It also can only go through public inheritance - it will always fail to travel through `protected` or `private` inheritance. This is rarely an issue, however, as such forms of inheritance are rare.

`reinterpret_cast`

`reinterpret_cast` is the most dangerous cast, and should be used very sparingly. It turns one type directly into another — such as casting the value from one pointer to another, or storing a pointer in an `int`, or all sorts of other nasty things. Largely, the only guarantee you get with `reinterpret_cast` is that normally if you cast the result back to the original type, you will get the exact same value (but ***not*** if the intermediate type is smaller than the original type). There are a number of conversions that **`reinterpret_cast`** cannot do, too. It's often abused for particularly weird conversions and bit manipulations, like turning a raw data stream into actual data, or storing data in the low bits of a pointer to aligned data. For those cases, see `std::bit_cast`.

C-Style Cast and Function-Style Cast

C-style cast and function-style cast are casts using `(type)object` or `type(object)`, respectively, and are functionally equivalent. They are defined as the first of the following which succeeds:

- `const_cast`
- `static_cast` (though ignoring access restrictions)
- `static_cast` (see above), then `const_cast`
- `reinterpret_cast`
- `reinterpret_cast`, then `const_cast`

It can therefore be used as a replacement for other casts in some instances, but can be extremely dangerous because of the ability to devolve into a `reinterpret_cast`, and the latter should be preferred when explicit casting is needed, unless you are sure `static_cast` will succeed or `reinterpret_cast` will fail. Even then, consider the longer, more explicit option.

C-style casts also ignore access control when performing a `static_cast`, which means that they have the ability to perform an operation that no other cast can. This is mostly a kludge, though, and in my mind is just another reason to avoid C-style casts.

`std::bit_cast` [C++20]

`std::bit_cast` copies the bits and bytes of the source object (its representation) directly into a new object of the target type. It's a standards-compliant way to do type punning. If you find yourself writing `*reinterpret_cast<SomeType*>(&x)`, you probably should use `std::bit_cast<SomeType>(x)` instead.

`std::bit_cast` is declared in `<bit>`. The objects must be the same size and be trivially copyable. If you can't yet use C++20, use `memcpy` to copy the source value into a variable of the desired type.

Share Improve this answer Follow

edited Jan 12 at 16:36

community wiki
26 revs, 20 users 59%
coppro

---

23 dynamic_cast is only for polymorphic types. you only need to use it when you're casting to a derived class. static_cast is certainly the first option unless you specifically need dynamic_cast's functinoality. It's not some miraculous silver-

bullet "type-checking cast" in general.
– Stack Overflow is garbage Dec 1, 2008 at 21:20

3 Great answer! One quick remark: static_cast might be necessary to cast up the hierarchy in case you have a Derived*& to cast into Base*&, since double pointers/references don't automagically cast up the hierarchy. I came across such (frankly, not common) situation two minutes ago. ;-) – bartgol Apr 8, 2013 at 15:16

10 *"no other C++ cast is capable of removing `const` (not even `reinterpret_cast`)"... really? What about `reinterpret_cast<int *>(reinterpret_cast<uintptr_t>(static_cast<int const *>(0)))` ? – user541686 Jan 15, 2015 at 11:33

59 I think an important detail missing above is that dynamic_cast has a run-time performance penalty compared to static or reinterpret_cast. This is important, e.g. in real-time software. – jfritz42 Jul 30, 2015 at 0:55

11 May be worth mentioning that `reinterpret_cast` is often the weapon of choice when dealing with an API's set of opaque data types – Class Skeleton Aug 4, 2015 at 13:09

432

- Use `dynamic_cast` for converting pointers/references within an inheritance hierarchy.
- Use `static_cast` for ordinary type conversions.
- Use `reinterpret_cast` for low-level reinterpreting of bit patterns. Use with extreme caution.
- Use `const_cast` for casting away `const/volatile`. Avoid this unless you are stuck using a const-incorrect API.

Share Improve this answer

Follow

edited Aug 2, 2021 at 12:24

Mateen Ulhaq

27.1k ● 21 ● 117 ● 152

answered Dec 1, 2008 at 20:22

Fred Larson

62k ● 18 ● 116 ● 177

13 Be careful with dynamic_cast. It relies on RTTI and this will not work as expected across shared libraries boundaries. Simply because you build executable and shared library independly in there is no standardized way to sync RTTI across different builds. For this reason in Qt library there exists qobject_cast<> which uses the QObject type info for checking types. – user3150128 Oct 23, 2018 at 8:25

1 You cannot use `dynamic_cast` for converting pointers/references within an inheritance hierarchy which does not use virtual polymorphism. For those hierarchies you must use `static_cast`. – JMC Mar 16, 2023 at 12:59

241

(A lot of theoretical and conceptual explanation has been given above)

Below are some of the **practical examples** when I used **static_cast**, **dynamic_cast**, **const_cast**, **reinterpret_cast**.

(Also referes this to understand the explaination : http://www.cplusplus.com/doc/tutorial/typecasting/)

static_cast :

```
OnEventData(void* pData)

{
  ......

  //  pData is a void* pData,

  //  EventData is a structure e.g.
  //  typedef struct _EventData {
  //  std::string id;
  //  std:: string remote_id;
  //  } EventData;

  // On Some Situation a void pointer *pData
  // has been static_casted as
  // EventData* pointer

  EventData *evtdata = static_cast<EventData*>(pData);
  .....
}
```

dynamic_cast :

```
void DebugLog::OnMessage(Message *msg)
{
    static DebugMsgData *debug;
    static XYZMsgData *xyz;

    if(debug = dynamic_cast<DebugMsgData*>(msg->pdata)
        // debug message
    }
    else if(xyz = dynamic_cast<XYZMsgData*>(msg->pdata
        // xyz message
    }
    else/* if( ... )*/{
        // ...
    }
}
```

const_cast :

```
// *Passwd declared as a const

const unsigned char *Passwd


// on some situation it require to remove its constnes

const_cast<unsigned char*>(Passwd)
```

reinterpret_cast :

```
typedef unsigned short uint16;

// Read Bytes returns that 2 bytes got read.

bool ByteBuffer::ReadUInt16(uint16& val) {
  return ReadBytes(reinterpret_cast<char*>(&val), 2);
}
```

Share Improve this answer

Follow

edited Jul 11, 2015 at 0:22

RamblingMad
5,518 • 2 • 26 • 52

answered Jan 21, 2014 at 4:53

Sumit Arora
5,211 • 8 • 41 • 60

43 The theory of some of the other answers are good, but still confusing, seeing these examples after reading the other answers really makes them all make sense. That is without the examples, I was still unsure, but with them, I am now sure about what the other answers mean. – Solx Apr 29, 2014 at 14:41

1 About the last usage of reinterpret_cast: isn't this the same as using `static_cast<char*>(&val)` ? – Lorenzo Belli

May 27, 2016 at 11:55

6 @LorenzoBelli Of course not. Did you try it? The latter is not valid C++ and blocks compilation. `static_cast` only works between types with defined conversions, visible relation by inheritance, or to/from `void *`. For everything else, there are other casts. `reinterpret cast` to any `char *` type is permitted to allow reading the representation of any object - and one of the only cases where that keyword is useful, not a rampant generator of implementation-/undefined behaviour. But this isn't considered a 'normal' conversion, so isn't allowed by the (usually) very conservative `static_cast`. – underscore_d Jul 16, 2016 at 22:53

2 reinterpret_cast is pretty common when you are working with system software such as databases. Most cases you write your own page manager which has no idea about what is the data type stored in the page and just returns a void pointer. Its up to the higher levels to do a reinterpret cast and infer it as whatever they want. – Sohaib May 17, 2017 at 11:46

1 The first example is dangerous, in that it assumes good behavior on the part of the caller (to always pass a pointer to an actual `EventData` object and nothing else). Unfortunately I don't think there's any practical way to type-check a void pointer in any meaningful way. Ideally the argument would be strongly-typed. Just some observations; not a critique of the answer. – Brian A. Henning Jan 29, 2019 at 15:29

149

It might help if you know little bit of internals...

static_cast

- C++ compiler already knows how to convert between scaler types such as `float` to `int`. Use `static_cast` for them.

- When you ask compiler to convert from type `A` to `B`, `static_cast` calls `B`'s constructor passing `A` as param. Alternatively, `A` could have a conversion operator (i.e. `A::operator B()`). If `B` doesn't have such constructor, or `A` doesn't have a conversion operator, then you get compile time error.
- Cast from `A*` to `B*` always succeeds if A and B are in inheritance hierarchy (or void) otherwise you get compile error.
- **Gotcha**: If you cast base pointer to derived pointer but if actual object is not really derived type then you *don't* get error. You get bad pointer and very likely a segfault at runtime. Same goes for `A&` to `B&`.
- **Gotcha**: Cast from Derived to Base or viceversa creates *new* copy! For people coming from C#/Java, this can be a huge surprise because the result is basically a chopped off object created from Derived.

dynamic_cast

- dynamic_cast uses runtime type information to figure out if cast is valid. For example, `(Base*)` to `(Derived*)` may fail if pointer is not actually of derived type.
- This means, dynamic_cast is very expensive compared to static_cast!
- For `A*` to `B*`, if cast is invalid then dynamic_cast will return nullptr.

- For `A&` to `B&` if cast is invalid then dynamic_cast will throw bad_cast exception.
- Unlike other casts, there is runtime overhead.

const_cast

- While static_cast can do non-const to const it can't go other way around. The const_cast can do both ways.
- One example where this comes handy is iterating through some container like `set<T>` which only returns its elements as const to make sure you don't change its key. However if your intent is to modify object's non-key members then it should be ok. You can use const_cast to remove constness.
- Another example is when you want to implement `T& SomeClass::foo()` as well as `const T& SomeClass::foo() const`. To avoid code duplication, you can apply const_cast to return value of one function from another.

reinterpret_cast

- This basically says that take these bytes at this memory location and think of it as given object.
- For example, you can load 4 bytes of `float` to 4 bytes of `int` to see how bits in `float` looks like.
- Obviously, if data is not correct for the type, you may get segfault.

- There is no runtime overhead for this cast.

Share Improve this answer

Follow

edited Dec 22, 2020 at 12:41

Andrew Truckle

19k ● 17 ● 82 ● 213

answered Dec 11, 2016 at 2:05

Shital Shah

68.4k ● 20 ● 256 ● 198

I added the conversion operator information, but there are a few other things that should be fixed as well and I don't feel that comfortable updating this too much. Items are: 1. `If you cast base pointer to derived pointer but if actual object is not really derived type then you don't get error. You get bad pointer and segfault at runtime.` You get UB which may result in a segfault at runtime if you're lucky. 2. Dynamic casts can also be used in cross casting. 3. Const casts can result in UB in some cases. Using `mutable` may be a better choice to implement logical constness. – Adrian Nov 12, 2018 at 20:33

---

1 @Adrian you are correct in all count. The answer is written for folks at more or less beginner level and I didn't wanted to overwhelm them with all other complications that comes with `mutable`, cross casting etc. – Shital Shah Nov 16, 2018 at 11:20

@Shital Shah "Cast from Derived to Base or viceversa creates new copy! For people coming from C#/Java, this can be a huge surprise because the result is basically a chopped off object created from Derived." Could you please show a simple example code to make it easier to understand? Thanks. – John Oct 27, 2021 at 1:11

---

2 What about std::bit_cast ? – frakod Jul 19, 2022 at 4:49

25

`static_cast` vs `dynamic_cast` vs `reinterpret_cast` internals view on a downcast/upcast

In this answer, I want to compare these three mechanisms on a concrete upcast/downcast example and analyze what happens to the underlying pointers/memory/assembly to give a concrete understanding of how they compare.

I believe that this will give a good intuition on how those casts are different:

- `static_cast`: does one address offset at runtime (low runtime impact) and no safety checks that a downcast is correct.
- `dyanamic_cast`: does the same address offset at runtime like `static_cast`, but also and an expensive safety check that a downcast is correct using RTTI.

  This safety check allows you to query if a base class pointer is of a given type at runtime by checking a return of `nullptr` which indicates an invalid downcast.

  Therefore, if your code is not able to check for that `nullptr` and take a valid non-abort action, you should just use `static_cast` instead of dynamic cast.

  If an abort is the only action your code can take, maybe you only want to enable the `dynamic_cast` in debug builds (`-NDEBUG`), and use `static_cast` otherwise, e.g. as done here, to not slow down your fast runs.

- `reinterpret_cast`: does nothing at runtime, not even the address offset. The pointer must point exactly to the correct type, not even a base class works. You generally don't want this unless raw byte streams are involved.

Consider the following code example:

main.cpp

```
#include <iostream>

struct B1 {
    B1(int int_in_b1) : int_in_b1(int_in_b1) {}
    virtual ~B1() {}
    void f0() {}
    virtual int f1() { return 1; }
    int int_in_b1;
};

struct B2 {
    B2(int int_in_b2) : int_in_b2(int_in_b2) {}
    virtual ~B2() {}
    virtual int f2() { return 2; }
    int int_in_b2;
};

struct D : public B1, public B2 {
    D(int int_in_b1, int int_in_b2, int int_in_d)
        : B1(int_in_b1), B2(int_in_b2), int_in_d(int_i
    void d() {}
    int f2() { return 3; }
    int int_in_d;
};
```

```
int main() {
    B2 *b2s[2];
    B2 b2{11};
    D *dp;
    D d{1, 2, 3};

    // The memory layout must support the virtual meth
    b2s[0] = &b2;
    // An upcast is an implicit static_cast<>().
    b2s[1] = &d;
    std::cout << "&d           " << &d           << st
    std::cout << "b2s[0]       " << b2s[0]       << st
    std::cout << "b2s[1]       " << b2s[1]       << st
    std::cout << "b2s[0]->f2() " << b2s[0]->f2() << st
    std::cout << "b2s[1]->f2() " << b2s[1]->f2() << st

    // Now for some downcasts.

    // Cannot be done implicitly
    // error: invalid conversion from ‘B2*’ to ‘D*’ [-
    // dp = (b2s[0]);

    // Undefined behaviour to an unrelated memory addr
not D.
    dp = static_cast<D*>(b2s[0]);
    std::cout << "static_cast<D*>(b2s[0])            "
std::endl;
    std::cout << "static_cast<D*>(b2s[0])->int_in_d  "
std::endl;

    // OK
    dp = static_cast<D*>(b2s[1]);
    std::cout << "static_cast<D*>(b2s[1])            "
std::endl;
    std::cout << "static_cast<D*>(b2s[1])->int_in_d  "
std::endl;

    // Segfault because dp is nullptr.
    dp = dynamic_cast<D*>(b2s[0]);
    std::cout << "dynamic_cast<D*>(b2s[0])           "
std::endl;
    //std::cout << "dynamic_cast<D*>(b2s[0])->int_in_d
std::endl;
```

```
    // OK
    dp = dynamic_cast<D*>(b2s[1]);
    std::cout << "dynamic_cast<D*>(b2s[1])           "
std::endl;
    std::cout << "dynamic_cast<D*>(b2s[1])->int_in_d "
std::endl;

    // Undefined behaviour to an unrelated memory addr
    // did not calculate the offset to get from B2* to
    dp = reinterpret_cast<D*>(b2s[1]);
    std::cout << "reinterpret_cast<D*>(b2s[1])
std::endl;
    std::cout << "reinterpret_cast<D*>(b2s[1])->int_in
std::endl;
}
```

Compile, run and disassemble with:

```
g++ -ggdb3 -O0 -std=c++11 -Wall -Wextra -pedantic -o m
setarch `uname -m` -R ./main.out
gdb -batch -ex "disassemble/rs main" main.out
```

where `setarch` is used to disable ASLR to make it easier to compare runs.

Possible output:

```
&d             0x7fffffffc930
b2s[0]         0x7fffffffc920
b2s[1]         0x7fffffffc940
b2s[0]->f2() 2
b2s[1]->f2() 3
static_cast<D*>(b2s[0])            0x7fffffffc910
static_cast<D*>(b2s[0])->int_in_d  1
static_cast<D*>(b2s[1])            0x7fffffffc930
static_cast<D*>(b2s[1])->int_in_d  3
dynamic_cast<D*>(b2s[0])           0
dynamic_cast<D*>(b2s[1])           0x7fffffffc930
```

```
dynamic_cast<D*>(b2s[1])->int_in_d 3
reinterpret_cast<D*>(b2s[1])           0x7fffffffc940
reinterpret_cast<D*>(b2s[1])->int_in_d 32767
```

Now, as mentioned at: https://en.wikipedia.org/wiki/Virtual_method_table in order to support the virtual method calls efficiently, supposing that the memory data structures of B1 is of form:

```
B1:
  +0: pointer to virtual method table of B1
  +4: value of int_in_b1
```

and `B2` is of form:

```
B2:
  +0: pointer to virtual method table of B2
  +4: value of int_in_b2
```

then memory data structure of `D` has to look something like:

```
D:
  +0: pointer to virtual method table of D (for B1)
  +4: value of int_in_b1
  +8: pointer to virtual method table of D (for B2)
 +12: value of int_in_b2
 +16: value of int_in_d
```

The key fact is that the memory data structure of `D` contains inside it memory structure identical to that of `B1` and `B2`, i.e.:

- +0 looks exactly like a B1, with the B1 vtable for D followed by `int_in_b1`
- +8 looks exactly like a B2, with the B2 vtable for D followed by `int_in_b2`

or at a higher level:

```
D:
   +0: B1
   +8: B2
  +16: <fields of D itsef>
```

Therefore we reach the critical conclusion:

> an upcast or downcast only needs to shift the pointer value by a value known at compile time

This way, when `D` gets passed to the base type array, the type cast actually calculates that offset and points something that looks exactly like a valid `B2` in memory, except that this one has the vtable for `D` instead of `B2`, and therefore all virtual calls work transparently.

E.g.:

```
b2s[1] = &d;
```

simply needs to get the address of `d` + 8 to reach the corresponding B2-like data structure.

Now, we can finally get back to type casting and the analysis of our concrete example.

From the stdout output we see:

```
&d             0x7fffffffc930
b2s[1]         0x7fffffffc940
```

Therefore, the implicit `static_cast` done there did correctly calculate the offset from the full `D` data structure at 0x7fffffffc930 to the `B2` like one which is at 0x7fffffffc940. We also infer that what lies between 0x7fffffffc930 and 0x7fffffffc940 is likely be the `B1` data and vtable.

Then, on the downcast sections, it is now easy to understand how the invalid ones fail and why:

- `static_cast<D*>(b2s[0]) 0x7fffffffc910`: the compiler just went up 0x10 at compile time bytes to try and go from a `B2` to the containing `D`

  But because `b2s[0]` was not a `D`, it now points to an undefined memory region.

  The disassembly is:

  ```
  49          dp = static_cast<D*>(b2s[0]);
     0x0000000000000fc8 <+414>:   48 8b 45 d0     mo
     0x0000000000000fcc <+418>:   48 85 c0        te
     0x0000000000000fcf <+421>:   74 0a   je     0xf
     0x0000000000000fd1 <+423>:   48 8b 45 d0     mo
     0x0000000000000fd5 <+427>:   48 83 e8 10     su
     0x0000000000000fd9 <+431>:   eb 05   jmp    0xf
  ```

```
   0x0000000000000fdb <+433>:   b8 00 00 00 00   mo
   0x0000000000000fe0 <+438>:   48 89 45 98      mo
```

so we see that GCC does:

- check if pointer is NULL, and if yes return NULL
- otherwise, subtract 0x10 from it to reach the `D` which does not exist

- `dynamic_cast<D*>(b2s[0])                0`: C++ actually found that the cast was invalid and returned `nullptr`!

  There is no way this can be done at compile time, and we will confirm that from the disassembly:

```
59          dp = dynamic_cast<D*>(b2s[0]);
   0x00000000000010ec <+706>:   48 8b 45 d0      mo
   0x00000000000010f0 <+710>:   48 85 c0         te
   0x00000000000010f3 <+713>:   74 1d   je     0x1
   0x00000000000010f5 <+715>:   b9 10 00 00 00   mo
   0x00000000000010fa <+720>:   48 8d 15 f7 0b 20
0x200bf7(%rip),%rdx        # 0x201cf8 <_ZTI1D>
   0x0000000000001101 <+727>:   48 8d 35 28 0c 20
0x200c28(%rip),%rsi        # 0x201d30 <_ZTI2B2>
   0x0000000000001108 <+734>:   48 89 c7         mo
   0x000000000000110b <+737>:   e8 c0 fb ff ff   ca
<__dynamic_cast@plt>
   0x0000000000001110 <+742>:   eb 05   jmp    0x1
   0x0000000000001112 <+744>:   b8 00 00 00 00   mo
   0x0000000000001117 <+749>:   48 89 45 98      mo
```

  First there is a NULL check, and it returns NULL if th einput is NULL.

  Otherwise, it sets up some arguments in the RDX, RSI and RDI and calls `__dynamic_cast`.

I don't have the patience to analyze this further now, but as others said, the only way for this to work is for `__dynamic_cast` to access some extra RTTI in-memory data structures that represent the class hierarchy.

It must therefore start from the `B2` entry for that table, then walk this class hierarchy until it finds that the vtable for a `D` typecast from `b2s[0]`.

This is why dynamic cast is potentially expensive! Here is [an example where a one liner patch converting a `dynamic_cast` to a `static_cast` in a complex project reduced runtime by 33%!](#).

- `reinterpret_cast<D*>(b2s[1]) 0x7fffffffc940` this one just believes us blindly: we said there is a `D` at address `b2s[1]`, and the compiler does no offset calculations.

  But this is wrong, because D is actually at 0x7fffffffc930, what is at 0x7fffffffc940 is the B2-like structure inside D! So trash gets accessed.

  We can confirm this from the horrendous `-O0` assembly that just moves the value around:

```
70          dp = reinterpret_cast<D*>(b2s[1]);
   0x00000000000011fa <+976>:   48 8b 45 d8     mo
   0x00000000000011fe <+980>:   48 89 45 98     mo
```

Related questions:

- [How is dynamic_cast implemented](#)

- Downcasting using the 'static_cast' in C++

Tested on Ubuntu 18.04 amd64, GCC 7.4.0.

Share Improve this answer

Follow

edited Jan 8 at 14:33

Weijun Zhou

4,896 • 2 • 21 • 41

answered Feb 26, 2020 at 12:44

Ciro Santilli OurBigBook.com

380k • 117 • 1.3k • 1.1k

18

Does this answer your question?

I have never used `reinterpret_cast`, and wonder whether running into a case that needs it isn't a smell of bad design. In the code base I work on `dynamic_cast` is used a lot. The difference with `static_cast` is that a `dynamic_cast` does runtime checking which may (safer) or may not (more overhead) be what you want (see msdn).

Share Improve this answer

Follow

edited Apr 4, 2012 at 16:31

talnicolas

14k • 7 • 37 • 56

answered Dec 1, 2008 at 20:20

andreas buykx

12.9k • 11 • 64 • 76

---

5 I have used reintrepret_cast for one purpose -- getting the bits out of a double (same size as long long on my platform).

– Joshua Nov 14, 2009 at 22:33

3 reinterpret_cast is needed e.g. for working with COM objects. CoCreateInstance() has output parameter of type void** (the last parameter), in which you will pass your pointer declared as e.g. "INetFwPolicy2* pNetFwPolicy2". To do that, you need to write something like reinterpret_cast<void**>(&pNetFwPolicy2) . – Serge Rogatch May 31, 2015 at 13:44

2 Perhaps there is a different approach, but I use `reinterpret_cast` to extract pieces of data out of an array. For instance if I have a `char*` containing a big buffer full of packed binary data that I need to move through and get individual primitives of varying types. Something like this: `template<class ValType> unsigned int readValFromAddress(char* addr, ValType& val) { /*On platforms other than x86(_64) this could do unaligned reads, which could be bad*/ val = (*(reinterpret_cast<ValType*>(addr))); return sizeof(ValType); }` – James Matta Aug 23, 2018 at 19:34

1 Personally I have only ever seen `reinterpret_cast` used for one reason. I've seen raw object data stored to a "blob" datatype in a database, then when the data is retrieved from the database, `reinterpret_cast` is used to turn this raw data into the object. – ImaginaryHuman072889 Oct 7, 2019 at 11:29

2 Not encountering `reinterpret_cast` comments tells a lot. If your `C++` is using another `C` library, **then** you'll see a lot of `reinterpret_cast` . – daparic Aug 14, 2020 at 3:54

18

In addition to the other answers so far, here is unobvious example where `static_cast` is not sufficient so that `reinterpret_cast` is needed. Suppose there is a function which in an output parameter returns pointers to objects of different classes (which do not share a common base class). A real example of such function is `CoCreateInstance()` (see the last parameter, which is in fact `void**`). Suppose you request particular class of object from this function, so you know in advance the type for the pointer (which you often do for COM objects). In this case you cannot cast pointer to your pointer into `void**` with `static_cast`: you need `reinterpret_cast<void**>(&yourPointer)`.

In code:

```
#include <windows.h>
#include <netfw.h>
.....
INetFwPolicy2* pNetFwPolicy2 = nullptr;
HRESULT hr = CoCreateInstance(__uuidof(NetFwPolicy2),
    CLSCTX_INPROC_SERVER, __uuidof(INetFwPolicy2),
    //static_cast<void**>(&pNetFwPolicy2) would give a
    reinterpret_cast<void**>(&pNetFwPolicy2) );
```

However, `static_cast` works for simple pointers (not pointers to pointers), so the above code can be rewritten to avoid `reinterpret_cast` (at a price of an extra variable) in the following way:

```
#include <windows.h>
#include <netfw.h>
.....
INetFwPolicy2* pNetFwPolicy2 = nullptr;
void* tmp = nullptr;
HRESULT hr = CoCreateInstance(__uuidof(NetFwPolicy2),
    CLSCTX_INPROC_SERVER, __uuidof(INetFwPolicy2),
```

```
    &tmp );
pNetFwPolicy2 = static_cast<INetFwPolicy2*>(tmp);
```

Share Improve this answer

Follow

edited Feb 22, 2016 at 10:30

Lii

12.1k ● 9 ● 68 ● 89

answered May 31, 2015 at 14:16

Serge Rogatch

14.9k ● 9 ● 95 ● 178

1 Wouldn't it work something like `&static_cast<void*>(pNetFwPolicy2)` instead of `static_cast<void**>(&pNetFwPolicy2)`? – jp48 Aug 14, 2019 at 17:26

@jp48 No, because `static_cast<void*>(pNetFwPolicy2)` is an rvalue so you can't take its address. – Donald Duck Jul 30 at 14:11

16

While other answers nicely described all differences between C++ casts, I would like to add a short note why you should not use C-style casts `(Type) var` and `Type(var)`.

For C++ beginners C-style casts look like being the superset operation over C++ casts (static_cast<>(), dynamic_cast<>(), const_cast<>(), reinterpret_cast<>()) and someone could prefer them over the C++ casts. In fact C-style cast is the superset and shorter to write.

The main problem of C-style casts is that they hide developer real intention of the cast. The C-style casts can do virtually all types of casting from normally safe casts done by static_cast<>() and dynamic_cast<>() to potentially dangerous casts like const_cast<>(), where const modifier can be removed so the const variables can be modified and reinterpret_cast<>() that can even reinterpret integer values to pointers.

Here is the sample.

```
int a=rand(); // Random number.

int* pa1=reinterpret_cast<int*>(a); // OK. Here develo
wanted to do this potentially dangerous operation.

int* pa2=static_cast<int*>(a); // Compiler error.
int* pa3=dynamic_cast<int*>(a); // Compiler error.

int* pa4=(int*) a; // OK. C-style cast can do such cas
was intentional or developer just did some typo.

*pa4=5; // Program crashes.
```

The main reason why C++ casts were added to the language was to allow a developer to clarify his intentions - why he is going to do that cast. By using C-style casts which are perfectly valid in C++ you are making your code less readable and more error prone especially for other developers who didn't create your code. So to make your code more readable and explicit you should always prefer C++ casts over C-style casts.

Here is a short quote from Bjarne Stroustrup's (the author of C++) book The C++ Programming Language 4th edition - page 302.

> This C-style cast is far more dangerous than the named conversion operators because the notation is harder to spot in a large program and the kind of conversion intended by the programmer is not explicit.

Share Improve this answer

Follow

answered Aug 22, 2018 at 11:18

Timmy_A

1,232 ● 13 ● 10

1 Upvoting due to referencing Stroustrup's quote. Hard to find these days especially that we often instead heard it from *very smart* people instead of the man himself. – daparic Aug 14, 2020 at 4:02

1 Upvoting because it's informational. But, I still definitely prefer C-style casts in my C++ code. I'm doing some more study on it now by reading answers here and looking at CplusPlus.com here: cplusplus.com/doc/tutorial/typecasting. It is yet to be determined whether or not I will come out converted and begin using C++-style casts in C++. I rather hate the verbosity and type-safetyness-over-readability aspect of C++. – Gabriel Staples Mar 24, 2023 at 4:00

6

To understand, let's consider below code snippet:

```
struct Foo{};
struct Bar{};

int main(int argc, char** argv)
{
    Foo* f = new Foo;
```

```
    Bar* b1 = f;                                  // (1)
    Bar* b2 = static_cast<Bar*>(f);               // (2)
    Bar* b3 = dynamic_cast<Bar*>(f);              // (3)
    Bar* b4 = reinterpret_cast<Bar*>(f);          // (4)
    Bar* b5 = const_cast<Bar*>(f);                // (5)

    return 0;
}
```

Only line (4) compiles without error. Only **reinterpret_cast** can be used to convert a pointer to an object to a pointer to an any unrelated object type.

One this to be noted is: The **dynamic_cast** would fail at run-time, however on most compilers it will also fail to compile because there are no virtual functions in the struct of the pointer being casted, meaning **dynamic_cast** will work with only polymorphic class pointers.

When to use C++ cast:

- Use **static_cast** as the equivalent of a C-style cast that does value conversion, or when we need to explicitly up-cast a pointer from a class to its superclass.
- Use **const_cast** to remove the const qualifier.
- Use **reinterpret_cast** to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if we know what we are doing and we understand the aliasing issues.

Share Improve this answer edited Dec 21, 2018 at 3:22

Follow

answered Dec 21, 2018 at 2:53

pkthapa

1,071 ● 2 ● 17 ● 28

The provided snippet is a bad example. While I agree that, indeed, it compiles. The *When* listing is vaguely correct but mostly filled with opinions insufficient to fathom the required granularity. – daparic Aug 14, 2020 at 4:13

2

Let's see the difference of `reinterpret_cast` and `static_cast` in an example:

```
#include <iostream>
using namespace std;

class A
{
    int a;
};

class B
{
    int b;
};

class C : public A, public B
{
    int c;
};

int main()
{
    {
        B b;
        cout << &b << endl;
```

```
        cout << static_cast<C *>(&b) << endl;      //
        cout << reinterpret_cast<C *>(&b) << endl; //
    }
    cout << endl;
    {
        C c;
        cout << &c << endl;
        cout << static_cast<B *>(&c) << endl;      //
        cout << reinterpret_cast<B *>(&c) << endl; //
    }
    cout << endl;
    {
        A a;
        cout << &a << endl;
        cout << static_cast<C *>(&a) << endl;
        cout << reinterpret_cast<C *>(&a) << endl;
    }
    cout << endl;
    {
        C c;
        cout << &c << endl;
        cout << static_cast<A *>(&c) << endl;
        cout << reinterpret_cast<A *>(&c) << endl;
    }
    return 0;
}
```

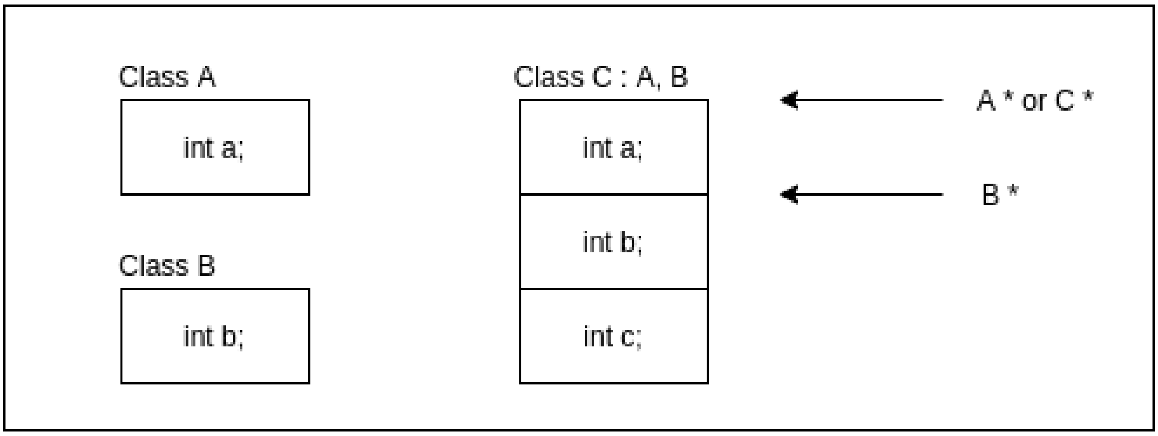
Produces the output:

```
0x7ffcede34f0c
0x7ffcede34f08 // 1
0x7ffcede34f0c // 2

0x7ffcede34f0c
0x7ffcede34f10 // 3
0x7ffcede34f0c // 4

0x7ffcede34f0c
0x7ffcede34f0c
0x7ffcede34f0c

0x7ffcede34f0c
```

```
0x7ffcede34f0c
0x7ffcede34f0c
```

Notice that output `1` and `2` are different, as well as `3` and `4`. Why is that? One of them is `static_cast` and the other is `reinterpret_cast` to the same type of the same input in both cases.

The situation can be visualized in the following figure:



`C` contains a `B` but the starting address of `B` is not the same as `C`. `static_cast` correctly calculates the address of `B` within `C`. However `reinterpret_cast` returns the same address we give as input, which is not correct for this case: there is no `B` at that address.

However, both casts return the same results when converting between `A` and `C` pointers because they happen to start at the same location which by the way is not anyway guaranteed by the standard.

Share Improve this answer Follow

edited Aug 28, 2022 at 12:22

answered Aug 28, 2022 at 11:55

Özgür Murat Sağdıçoğlu
3,396 ● 2 ● 16 ● 28

2

I think we need a more beginner-friendly explanation, and after just studying the topic myself, I think the best I've found is here: https://www.tutorialspoint.com/When-should-static-cast-dynamic-cast-const-cast-and-reinterpret-cast-be-used-in-Cplusplus [1]

> ## When should `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast` be used in C++?
>
> `const_cast`
> can be used to remove or add const to a variable. This can be useful if it is necessary to add/remove constness from a variable.
>
> `static_cast`
> This is used for the normal/ordinary type conversion. This is also the cast responsible for implicit type co[nv]ersion and can also be called explicitly. You should use it in cases like converting float to int, char to int, etc.
>
> `dynamic_cast`
> This cast is used for handling polymorphism. You only need to use it when you're casting to a derived class. This is exclusively to be used in

> inheritence when you cast from base class to derived class.

- My own added words: it allows safe downcasting to convert from a ptr to a base class (which was created by taking the address of a derived class inheriting from that base class) to a ptr to a derived class, ensuring that the ptr actually points to a full, complete, derived class object (and returning a `nullptr` if not). There's a great code example in the "`dynamic_cast`" section here: https://cplusplus.com/doc/tutorial/typecasting/. (And I discuss that code [in my answer here](#)).

> `reinterpret_cast`
> This is the trickiest to use. It is used for reinterpreting bit patterns and is extremely low level. It's used primarily for things like turning a raw data bit stream into actual data or storing data in the low bits of an aligned pointer.

Then, for a deeper dive, read these:

1. Quora Answer by @Brian Bi: https://qr.ae/prz8xL - very good, thoughtful, well-written, and thorough answer. Here is the summary from the end [order rearranged to be the same as the order above]:

> - `const_cast` only changes cv-qualification; all other casts cannot cast away constness.
> - `static_cast` performs implicit conversions, the reverses of implicit standard conversions, and (possibly unsafe) base to derived conversions.
> - `dynamic_cast` casts up and down class hierarchies only, always checking that the conversion requested is valid.
> - `reinterpret_cast` converts one pointer to another without changing the address, or converts between pointers and their numerical (integer) values.

2. [The main community wiki answer here]()
3. [Easy to read; written for everybody; very informative] https://cplusplus.com/doc/tutorial/typecasting/ - this article also contains **code examples of each type of cast!**
4. [pedantic, language lawyer, hard to read, but more thorough] CppReference wiki:
    1. https://en.cppreference.com/w/cpp/language/const_cast
    2. https://en.cppreference.com/w/cpp/language/static_cast

3. https://en.cppreference.com/w/cpp/language/dynamic_cast
4. https://en.cppreference.com/w/cpp/language/reinterpret_cast

---

5.
6. https://en.cppreference.com/w/cpp/language/explicit_cast
7. https://en.cppreference.com/w/cpp/language/implicit_cast

[1]Note: TutorialsPoint is notorious for plagiarism and not citing sources. I think they actually took wording from the main community wiki answer here without citing it. However, I do like the extreme conciseness and simplicity of their article, making it easy to *begin to grasp* for a beginner, or *quick to review* for someone who needs a refresher during or just before an interview or test.

Share Improve this answer Follow

edited Jun 7, 2023 at 15:03

answered Mar 24, 2023 at 22:51

Gabriel Staples
51.5k ● 30 ● 273 ● 354

1

Nice feature of `reinterpret_cast`, not mentioned in the other answers, is that it allows us to create a sort of `void*` pointer for function types. Normally, for object types one uses `static_cast` to retrieve the original type of a pointer stored in `void*`:

```
int i = 13;
void *p = &i;
auto *pi = static_cast<int*>(p);
```

For functions, we must use `reinterpret_cast` twice:

```
#include<iostream>

using any_fcn_ptr_t = void(*)();


void print(int i)
{
   std::cout << i <<std::endl;
}

int main()
{
  //Create type-erased pointer to function:
  auto any_ptr = reinterpret_cast<any_fcn_ptr_t>(&prin

  //Retrieve the original pointer:
  auto ptr = reinterpret_cast< void(*)(int) >(any_ptr)

  ptr(7);
}
```

With `reinterpret_cast` we can even get a similar sort-of-void* pointer for pointers to member functions.

As with plain `void*` and `static_cast`, C++ guarantees that `ptr` points to `print` function (as long as we pass the correct type to `reinterpret_cast`).

Share Improve this answer

Follow

answered Apr 16, 2022 at 21:36

Adrian

1,127 ● 7 ● 15

The problem with this is that `any_ptr();` compiles but is undefined behavior. – Donald Duck Jul 30 at 14:26

@Donald Duck Sure, but you are not supposed to call any_ptr. It's exactly like with void* pointer for object types -- it can only be dereferenced safely after converting back to original pointer type. – Adrian Jul 31 at 21:32