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

http://stackoverflow.com/questions/332030/when-should-static-cast-dynamic-cast-const-cast-and-reinterpret-cast-be-used

What are the proper uses of:

- **static_cast**
- **dynamic_cast**
- **const_cast**
- **reinterpret_cast**
- C-style cast **(type)value**
- Function-style cast **type(value)**

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

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 its behavior is undefined to static_cast down a hierarchy to a type that isn't actually the type of the object.

---

const_cast can be used to remove or add *const* to a variable; no other C++ cast is capable of removing *const* (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 is almost 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 the same class occurs multiple times 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.
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 functionality. It's not some miraculous silver-bullet "type-checking cast" in general.

---

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 used primarily for particularly weird conversions and bit manipulations, like turning a raw data stream into actual data, or storing data in the low bits of an aligned pointer.

C casts are casts using (type)object or type(object). A C-style cast is 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.

How do you explain the differences among static_cast, reinterpret_cast, const_cast, and dynamic_cast to a new C++ programmer?

https://www.quora.com/How-do-you-explain-the-differences-among-static_cast-reinterpret_cast-const_cast-and-dynamic_cast-to-a-new-C++-programmer

Why are they really needed?

Brian Bi, software engineer

Good question! Actually, this is the key to understanding why C++ has four different casts.

In C there is only a single cast, but it performs many different conversions:

1. Between two arithmetic types
2. Between a pointer type and an integer type
3. Between two pointer types
4. Between a cv-qualified and cv-unqualified type (const or volatile)
5. A combination of (4) and either (1), (2), or (3)

C++ adds the following features that really change the game:

- Inheritance
- Templates

Why are these important? Well, inheritance changes the game because now there are three different things we might mean by a pointer cast. Say class D derives from class B. What should (D*)pb do, where pb has type B*? Here are three possibilities:

1. Return a pointer to the **same byte of memory** (exactly the same address), but just change the type of the pointer. (This is the same as what all pointer casts do in C, as explained above.)
2. **Check** whether the B* really points to a B that is *part of* a D object. If so, return a pointer to the D object. If not, fail (maybe by returning a null pointer or throwing an exception.)
3. **Assume** that the B* points to a B that is part of a D object; don't bother performing a check. Adjust the address of the pointer if necessary so that it will point to the D object.

It is conceivable that in C++ we might want to perform any of these three conversions. Therefore C++ gives us three different casts:

- **reinterpret_cast**<D*> for function (1),
- **dynamic_cast**<D*> for function (2), and

- **static_cast**<D*> for function (3).

Now, I also mentioned the fact that C++ has templates. The reason why this is relevant is that if you use C-style casts in C++, and either the source or target or both types are template parameters or depend on template parameters, then in general *the kind of casting involved might depend on the template parameters*. For example, say we write a function like this:

```
template <class T>
unsigned char* alias(T& x) {
    return (unsigned char*)(&x);
}
```

So you can pass in any lvalue and you get a pointer to its first byte. All right! But hang on, what happens if I pass in a const lvalue? Then the C-style cast silently casts away the constness, and the function returns an **unsigned char*** which we can then use to modify the original object, causing undefined behavior. So, you see, the C-style cast to pointer sometimes just changes the type of the object pointed to, but sometimes it just removes const (like if we pass in a **const unsigned char** lvalue) and sometimes it does both (like if we pass in a **const int** lvalue). Better to write

```
template <class T>
unsigned char* safe_alias(T& x) {
    return reinterpret_cast<unsigned char*>(&x);
}
```

That way, if someone passes in a const lvalue, they'll get a compilation error, telling them---whoops! You almost casted away constness, which is super dangerous! Glad we caught that in time, eh?

And that's why we need **const_cast** as a separate cast from the other three casts: its function is entirely distinct from the other three casts' functions, you rarely need to cast away constness, and you almost always want the compiler to prevent you from accidentally casting away constness. If every **static_cast**, **dynamic_cast**, and **reinterpret_cast** had the power to cast away constness too, using them would become a lot more dangerous---*especially* when templates are involved! Now if someone really wants to get a **char*** to a **const int** object, they can call, *e.g.,* safe_alias(**const_cast**<**int**&>(x)). Now it's explicit that constness is being cast away.

The fact that C++ has templates also forces us to explicitly perform implicit conversions sometimes! Consider the following in C:

```
int* pi;
void* pv = pi; // OK; implicit conversion
void* pv2 = (void*)pi; // OK but unnecessary
void f(void* pv);
f(pi); // OK; implicit conversion
f((void*)pi); // OK but unnecessary
```

But in C++ we can have something like this:

```
template <class T> void f(T* p);
// ...
int* pi;
f(pi); // OK; calls f<int>
f(static_cast<void*>(pi)); // OK; calls f<void>
```

Even though **int*** can be converted to **void*** *without a cast*, we might still need a cast anyway in order to force the argument to have the correct type! (That is, if we want to call f<void> and not f<int>.) In C this doesn't happen because you can't overload functions, let alone write function templates. Boost provides an implicit_cast function template specifically designed to explicitly perform implicit conversions. Many people feel that this should become a part of standard C++. But for now all we have is **static_cast**.

So to summarize approximately:

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

The reason why there are four different casts in C++ is so that you can write a cast and be explicit about what kind of conversion you intend to perform; the compiler will never incorrectly "guess" what you meant; and other people reading your code will be able to tell what kind of conversions it does.