There's something sneaky and underhanded about old-style casts. Their syntax is such that they can often pass unnoticed in a section of code, but they can do terrible damage, like an unexpected sucker punch from a bully. Let's clarify what we mean by "old-style" cast. Obviously, the original C syntax consisting of a parenthesized type applied to an expression is an old-style cast:

```c
char *hopeItWorks = (char *)0x00ff0000; // old-style cast
```

C++ introduced another way of saying the same thing with the function-style cast syntax:

```c
typedef char *PChar;
hopeItWorks = PChar( 0x00ff0000 ); // function-style/old-style cast
```

A function-style cast may look more civilized than its dread forebear, but it's just as nasty; avoid both of them like the plague.

Honest programmers use the new cast operators because they allow you to say more precisely what you mean, and mean more accurately what you say. There are four of them, and each has a specific purpose.

The `const_cast` operator allows you to add or remove `const` and `volatile` type qualifiers in an expression's type:

```c
const Person *getEmployee() { ... }
//...
Person *anEmployee = const_cast<Person *>(getEmployee());
```

In this code, we've used `const_cast` to strip a `const` type qualifier from the return type of `getEmployee`. We could have used an old-style cast to achieve the same result,

```c
anEmployee = (Person *)getEmployee();
```

but the `const_cast` is superior for a couple of reasons. First, it's obvious and hideous. It sticks out of the code like a sore thumb, and that's a good thing, because casts in any form are dangerous. They should be painful to write, because you should use them only if you have to use them. They should be easy to find, because casts are the "usual suspects" one examines first whenever a bug appears. Second, a `const_cast` is less powerful than an old-style cast because it will affect only type qualifiers. This restriction is a good thing as well, because it allows us to say more precisely what our intent is. Using an old-style cast tells the compiler to shut up because you intend that the return type of `getEmployee` be converted to `Person *`. Use of a `const_cast` tells the compiler to shut up because you intend to strip a `const` from the return type of `getEmployee`. There is not a big difference in these two statements (although they're both pretty disrespectful, really) unless some maintenance occurs to the `getEmployee` function:
const Employee *getEmployee(); // big change!

The gag rule enforced by the old-style cast is still in effect; the improper conversion of const Employee * to Person * will not be flagged by the compiler, but the compiler will complain about the const_cast, because that drastic a conversion is beyond its capabilities. In short, the const_cast is preferred to the old-style cast because it's more hideous, harder to use, and less powerful.

The static_cast operator is used for casts that are relatively portable across platforms. Most commonly, it is used to cast "down" an inheritance hierarchy from a base class pointer or reference toward a derived class pointer or reference (see also Capability Queries [27, 93]):

Shape *sp = new Circle;
Circle *cp = static_cast<Circle *>(sp); // downcast

In this case, the static_cast will result in correct code, because sp actually does refer to a Circle object. However, if sp had pointed instead to some other type of Shape, we would have likely gotten a runtime error of some sort when we used cp. To reiterate, these new cast operators are safer than old-style casts, but they're not necessarily safe.

Note that a static_cast may not change type qualifiers the way a const_cast can. This implies that it is sometimes necessary to use a sequence of two new cast operators to achieve what a single old-style cast could do:

const Shape *getNextShape() { ... }
//...
Circle *cp =
    static_cast<Circle *>(const_cast<Shape *>(getNextShape()));

The standard doesn't guarantee much about the behavior of reinterpret_cast, but it generally does just what its name says; it looks at the bits of an object and lets you pretend they represent something else entirely:

hopeItWorks = // pretend int is pointer
    reinterpret_cast<char *>(0x00ff0000);
int *hopeless = // pretend char * is int *
    reinterpret_cast<int *>(hopeItWorks);

This sort of thing is occasionally necessary in low-level code, but it's not likely to be portable. Proceed with caution. Note the difference between reinterpret_cast and static_cast when down casting from a pointer to a base class to a pointer to a derived class. A reinterpret_cast will typically just pretend the base class pointer is a derived class pointer without changing its value whereas a static_cast (and an old style cast, for that matter) will perform the correct address manipulation (see Meaning of Pointer Comparison [28, 97]).

Speaking of casting within a hierarchy brings us to dynamic_cast. A dynamic_cast is used typically to perform a safe down cast from a pointer to a base to a derived class (but see Capability Queries [27, 93]). The difference from static_cast is that a dynamic_cast used as a down cast may be performed only on a polymorphic type (that is, the type of the expression being cast has to be a
pointer to class type with a virtual function), and the cast actually performs a runtime check to see that the cast is correct. This safety comes at a cost, though; a static_cast typically has no or minimal runtime cost whereas using a dynamic_cast implies significant runtime overhead.

    const Circle *cp =
        dynamic_cast<const Circle *>( getNextShape() );
    if( cp ) { ... }

If getNextShape returns a pointer to a Circle (or something publicly derived from Circle, that is, something that is-a Circle; see Polymorphism [2, 3]), the cast will succeed and cp will point to a Circle. Otherwise cp will be null. Note that we can combine the declaration and test in the same expression:

    if( const Circle *cp
        = dynamic_cast<const Circle *>(getNextShape()) ) { ... }

This is advantageous because it restricts the scope of the variable cp to the if-statement, so cp will just go out of scope when we no longer need it.

A less common use of dynamic_cast is to perform a down cast to a reference type:

    const Circle &rc = dynamic_cast<const Circle &>(*getNextShape());

The operation is similar to that of a dynamic_cast to a pointer type, but if the cast fails, the operator throws a std::bad_cast exception rather than simply returning a null pointer. (Remember, there are no null references; see References Are Aliases, Not Pointers [5, 13].) Idiomatically, a dynamic_cast to a pointer is asking a question ("Is this Shape pointer actually pointing to a Circle? If not, I can deal with it."), whereas a dynamic_cast to a reference is stating an invariant ("This Shape is supposed to be a Circle. If it's not, something is seriously wrong!").

As with the other new cast operators, use of dynamic_cast is occasionally necessary, but it is often overused because of its reputation for being a "safe" cast. See Factory Method [30, 103] for an example of such overuse.