Item 27: Minimize casting.

The rules of C++ are designed to guarantee that type errors are impossible. In theory, if your program compiles cleanly, it's not trying to perform any unsafe or nonsensical operations on any objects. This is a valuable guarantee. You don't want to forgo it lightly.

Unfortunately, casts subvert the type system. That can lead to all kinds of trouble, some easy to recognize, some extraordinarily subtle. If you're coming to C++ from C, Java, or C#, take note, because casting in those languages is more necessary and less dangerous than in C++. But C++ is not C. It's not Java. It's not C#. In this language, casting is a feature you want to approach with great respect.

Let's begin with a review of casting syntax, because there are usually three different ways to write the same cast. C-style casts look like this:

```
(T) expression                          // cast expression to be of type T
```

Function-style casts use this syntax:

```
T(expression)                           // cast expression to be of type T
```

There is no difference in meaning between these forms; it's purely a matter of where you put the parentheses. I call these two forms *old-style casts*.

C++ also offers four new cast forms (often called *new-style* or *C++-style casts*):

```
const_cast<T>(expression)

dynamic_cast<T>(expression)

reinterpret_cast<T>(expression)

static_cast<T>(expression)
```

Each serves a distinct purpose:

- `const_cast` is typically used to cast away the constness of objects. It is the only C++-style cast that can do this.
- `dynamic_cast` is primarily used to perform "safe downcasting," i.e., to determine whether an object is of a particular type in an inheritance hierarchy. It is the only cast that cannot be performed using the old-style syntax. It is also the only cast that may have a significant runtime cost. (I'll provide details on this a bit later.)
- `reinterpret_cast` is intended for low-level casts that yield implementation-dependent (i.e., unportable) results, e.g., casting a pointer to an `int`. Such casts should be rare outside low-level code. I use it only once in this book, and that's only when discussing how you might write a debugging allocator for raw memory (see Item 50).
- `static_cast` can be used to force implicit conversions (e.g., non-`const` object to `const` object (as in Item 3), `int` to `double`, etc.). It can also be used to perform the reverse of many such conversions (e.g., `void*`

pointers to typed pointers, pointer-to-base to pointer-to-derived), though it cannot cast from `const` to non-`const` objects. (Only `const_cast` can do that.)

The old-style casts continue to be legal, but the new forms are preferable. First, they're much easier to identify in code (both for humans and for tools like `grep`), thus simplifying the process of finding places in the code where the type system is being subverted. Second, the more narrowly specified purpose of each cast makes it possible for compilers to diagnose usage errors. For example, if you try to cast away constness using a new-style cast other than `const_cast`, your code won't compile.

About the only time I use an old-style cast is when I want to call an `explicit` constructor to pass an object to a function. For example:

```
class Widget {
public:
  explicit Widget(int size);
  ...
};

void doSomeWork(const Widget& w);

doSomeWork(Widget(15));                    // create Widget from int
                                           // with function-style cast

doSomeWork(static_cast<Widget>(15));       // create Widget from int
                                           // with C++-style cast
```

Somehow, deliberate object creation doesn't "feel" like a cast, so I'd probably use the function-style cast instead of the `static_cast` in this case. Then again, code that leads to a core dump usually feels pretty reasonable when you write it, so perhaps you'd best ignore feelings and use new-style casts all the time.

Many programmers believe that casts do nothing but tell compilers to treat one type as another, but this is mistaken. Type conversions of any kind (either explicit via casts or implicit by compilers) often lead to code that is executed at runtime. For example, in this code fragment,

```
int x, y;
...
double d = static_cast<double>(x)/y;          // divide x by y, but use
                                              // floating point division
```

the cast of the `int x` to a `double` almost certainly generates code, because on most architectures, the underlying representation for an `int` is different from that for a `double`. That's perhaps not so surprising, but this example may

widen your eyes a bit:

```
class Base { ... };

class Derived: public Base { ... };

Derived d;

Base *pb = &d;                          // implicitly convert Derived* ⇒ Base*
```

Here we're just creating a base class pointer to a derived class object, but sometimes, the two pointer values will not be the same. When that's the case, an offset is applied *at runtime* to the `Derived*` pointer to get the correct `Base*` pointer value.

This last example demonstrates that a single object (e.g., an object of type `Derived`) might have more than one address (e.g., its address when pointed to by a `Base*` pointer and its address when pointed to by a `Derived*` pointer). That can't happen in C. It can't happen in Java. It can't happen in C#. It *does* happen in C++. In fact, when multiple inheritance is in use, it happens virtually all the time, but it can happen under single inheritance, too. Among other things, that means you should generally avoid making assumptions about how things are laid out in C++, and you should certainly not perform casts based on such assumptions. For example, casting object addresses to `char*` pointers and then using pointer arithmetic on them almost always yields undefined behavior.

But note that I said that an offset is "sometimes" required. The way objects are laid out and the way their addresses are calculated varies from compiler to compiler. That means that just because your "I know how things are laid out" casts work on one platform doesn't mean they'll work on others. The world is filled with woeful programmers who've learned this lesson the hard way.

An interesting thing about casts is that it's easy to write something that looks right (and might be right in other languages) but is wrong. Many application frameworks, for example, require that virtual member function implementations in derived classes call their base class counterparts first. Suppose we have a `Window` base class and a `SpecialWindow` derived class, both of which define the virtual function `onResize`. Further suppose that `SpecialWindow`'s `onResize` is expected to invoke `Window`'s `onResize` first. Here's a way to implement this that looks like it does the right thing, but doesn't:

```
class Window {                                  // base class
public:
  virtual void onResize() { ... }               // base onResize impl
  ...
};

class SpecialWindow: public Window {            // derived class
public:
  virtual void onResize() {                     // derived onResize impl;
    static_cast<Window>(*this).onResize();      // cast *this to Window,
```

```
                                                // then call its onResize;
                                                // this doesn't work!

    ...                                         // do SpecialWindow-
  }                                             // specific stuff

  ...

};
```

I've highlighted the cast in the code. (It's a new-style cast, but using an old-style cast wouldn't change anything.) As you would expect, the code casts `*this` to a `Window`. The resulting call to `onResize` therefore invokes `Window::onResize`. What you might not expect is that it does not invoke that function on the current object! Instead, the cast creates a new, temporary *copy* of the base class part of `*this`, then invokes `onResize` on the copy! The above code doesn't call `Window::onResize` on the current object and then perform the `SpecialWindow`-specific actions on that object — it calls `Window::onResize` on a *copy of the base class part* of the current object before performing `SpecialWindow`-specific actions on the current object. If `Window::onResize` modifies the current object (hardly a remote possibility, since `onResize` is a non-`const` member function), the current object won't be modified. Instead, a *copy* of that object will be modified. If `SpecialWindow::onResize` modifies the current object, however, the current object *will* be modified, leading to the prospect that the code will leave the current object in an invalid state, one where base class modifications have not been made, but derived class ones have been.

The solution is to eliminate the cast, replacing it with what you really want to say. You don't want to trick compilers into treating `*this` as a base class object; you want to call the base class version of `onResize` on the current object. So say that:

```
class SpecialWindow: public Window {
public:
  virtual void onResize() {
    Window::onResize();                     // call Window::onResize
    ...                                     // on *this
  }
  ...

};
```

This example also demonstrates that if you find yourself wanting to cast, it's a sign that you could be approaching things the wrong way. This is especially the case if your want is for `dynamic_cast`.

Before delving into the design implications of `dynamic_cast`, it's worth observing that many implementations of `dynamic_cast` can be quite slow. For example, at least one common implementation is based in part on string comparisons of class names. If you're performing a `dynamic_cast` on an object in a single-inheritance hierarchy four levels deep, each `dynamic_cast` under such an implementation could cost you up to four calls to `strcmp` to compare class names. A deeper hierarchy or one using multiple inheritance would be more expensive. There are reasons that some implementations work this way (they have to do with support for dynamic linking). Nonetheless, in addition to being leery of casts in general, you should be especially leery of `dynamic_cast`s in performance-sensitive code.

The need for `dynamic_cast` generally arises because you want to perform derived class operations on what you believe to be a derived class object, but you have only a pointer- or reference-to-base through which to manipulate the object. There are two general ways to avoid this problem.

First, use containers that store pointers (often smart pointers — see Item 13) to derived class objects directly, thus eliminating the need to manipulate such objects through base class interfaces. For example, if, in our `Window`/`SpecialWindow` hierarchy, only `SpecialWindow`s support blinking, instead of doing this:

```
class Window { ... };


class SpecialWindow: public Window {

public:

  void blink();

  ...

};

typedef                                          // see Item 13 for info

  std::vector<std::tr1::shared_ptr<Window> > VPW;  // on tr1::shared_ptr


VPW winPtrs;


...


for (VPW::iterator iter = winPtrs.begin();       // undesirable code:

     iter != winPtrs.end();                      // uses dynamic_cast

     ++iter) {

  if (SpecialWindow *psw = dynamic_cast<SpecialWindow*>(iter->get()))

     psw->blink();

}
```

try to do this instead:

```
typedef std::vector<std::tr1::shared_ptr<SpecialWindow> > VPSW;

VPSW winPtrs;

...

for (VPSW::iterator iter = winPtrs.begin();          // better code: uses
     iter != winPtrs.end();                          // no dynamic_cast
     ++iter)
  (*iter)->blink();
```

Of course, this approach won't allow you to store pointers to all possible `Window` derivatives in the same container. To work with different window types, you might need multiple type-safe containers.

An alternative that will let you manipulate all possible `Window` derivatives through a base class interface is to provide virtual functions in the base class that let you do what you need. For example, though only `SpecialWindow`s can blink, maybe it makes sense to declare the function in the base class, offering a default implementation that does nothing:

```
class Window {
public:
  virtual void blink() {}                          // default impl is no-op;
  ...                                              // see Item 34 for why
};                                                 // a default impl may be
                                                   // a bad idea

class SpecialWindow: public Window {
public:
  virtual void blink() { ... };                    // in this class, blink
  ...                                              // does something
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;                                       // container holds
```

```
                                                  // (ptrs to) all possible
...                                               // Window types

for (VPW::iterator iter = winPtrs.begin();
     iter != winPtrs.end();
     ++iter)                                      // note lack of
  (*iter)->blink();                               // dynamic_cast
```

Neither of these approaches — using type-safe containers or moving virtual functions up the hierarchy — is universally applicable, but in many cases, they provide a viable alternative to `dynamic_cast`ing. When they do, you should embrace them.

One thing you definitely want to avoid is designs that involve cascading `dynamic_cast`s, i.e., anything that looks like this:

```
class Window { ... };

...                                   // derived classes are defined here

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin(); iter != winPtrs.end(); ++iter)
{
  if (SpecialWindow1 *psw1 =
       dynamic_cast<SpecialWindow1*>(iter->get())) { ... }

  else if (SpecialWindow2 *psw2 =
            dynamic_cast<SpecialWindow2*>(iter->get())) { ... }

  else if (SpecialWindow3 *psw3 =
            dynamic_cast<SpecialWindow3*>(iter->get())) { ... }
```

```
  ...
}
```

Such C++ generates code that's big and slow, plus it's brittle, because every time the `Window` class hierarchy changes, all such code has to be examined to see if it needs to be updated. (For example, if a new derived class gets added, a new conditional branch probably needs to be added to the above cascade.) Code that looks like this should almost always be replaced with something based on virtual function calls.

Good C++ uses very few casts, but it's generally not practical to get rid of all of them. The cast from `int` to `double` on page 118, for example, is a reasonable use of a cast, though it's not strictly necessary. (The code could be rewritten to declare a new variable of type `double` that's initialized with `x`'s value.) Like most suspicious constructs, casts should be isolated as much as possible, typically hidden inside functions whose interfaces shield callers from the grubby work being done inside.

Things to Remember

- Avoid casts whenever practical, especially `dynamic_cast`s in performance-sensitive code. If a design requires casting, try to develop a cast-free alternative.
- When casting is necessary, try to hide it inside a function. Clients can then call the function instead of putting casts in their own code.
- Prefer C++-style casts to old-style casts. They are easier to see, and they are more specific about what they do.