Crossroads
The ACM Student Magazine

Casting in C++: Bringing Safety and Smartness to Your Programs

By **G. Bowden Wise**

The new C++ standard is full of powerful additions to the language: templates, run-time type identification (RTTI), namespaces, and exceptions to name a few. Rather than talk about one of these ``major'' extensions, I will discuss one of the minor extensions: the new C++ *casting operators*.

The C++ draft standard includes the following four casting operators:

- `static_cast`
- `const_cast`
- `dynamic_cast`, and
- `reinterpret_cast`.

These new operators are intended to remove some of the holes in the C type system introduced by the old C-style casts.

In this article we will learn about casting in general, discuss the problems with the old C-style cast, and take a look at the new C++ casting operators in detail.

Why Cast?

Casts are used to convert the type of an object, expression, function argument, or return value to that of another type. Some conversions are performed automatically by the compiler without intervention by the programmer. These conversions are called **implicit conversions**. The standard C++ conversions and user-defined conversions are performed implicitly by the compiler where needed. Other conversions must be explicitly specified by the programmer and are appropriately called **explicit conversions**.

Standard conversions are used for integral promotions (e.g., `enum` to `int`), integral conversions (e.g., `int` to `unsigned int`), floating point conversions (e.g., `float` to `double`), floating-integral conversions (e.g., `int` to `float`), arithmetic conversions (e.g., converting operands to the type of the widest operand before evaluation), pointer conversions (e.g., derived class pointer to base class pointer), reference conversions (e.g., derived class reference to base class reference), and pointer-to-member conversions.

You can provide a user-defined conversion from a class `X` to a class `Y` by providing a constructor for `Y` that takes an `X` as an argument:

```
Y(const X& x)
```

or by providing a class `Y` with a conversion operator:

```
operator X()
```

When a type is needed for an expression that cannot be obtained through an implicit conversion or when more than one standard conversion creates an ambiguous situation, the programmer must explicitly specify the target type of the conversion.

In C, an expression, `expr`, of type `S` can be cast to another type `T` in one of the following ways. By using an explicit cast:

```
(T) expr
```

or by using a functional form:

```
T(expr)
```

We will refer to either of these constructs as the **old C-style casts**.

The old C-style casts have several shortcomings. First, the syntax is the same for every casting operation. This means it is impossible for the compiler (or users) to tell the intended purpose of the cast. Is it a cast from a base class pointer to a derived class pointer? Does the cast remove the ``const-ness'' of the object? Or, is it a conversion of one type to a completely unrelated type? The truth is, it is impossible to tell from the syntax. As a result, this makes the cast harder to comprehend, not only by humans, but also by compilers which are unable to detect improper casts.

Another problem is that the C-style casts are hard to find. Parentheses with an identifier between them are used all over C++ programs. There is no easy way to ``grep'' a source file and get a list of all the casts being performed.

Perhaps the most serious problem with the old C-style cast is that it allows you to cast practically any type to any other type. Improper use of casts can lead to disastrous results. The old C-style casts have created a few holes in the C type system and have also been a souce of confusion for both programmers and compilers. Even in C++, the old C-style casts are retained for backwards compatibility. However, using the new C++ style casting operators will make your programs more readable, type-safe, less error-prone, and easier to maintain.

The New C++ Casting Operators

The new C++ casting operators are intended to provide a solution to the shortcomings of the old C-style casts by providing:

- *Improved syntax.* Casts have a clear, concise, although somewhat cumbersome syntax. This makes casts easier to understand, find, and maintain.
- *Improved semantics.* The intended meaning of a cast is no longer ambiguous. Knowing what the programmer intended the cast to do makes it possible for compilers to detect improper casting operations.
- *Type-safe conversions.* Allow some casts to be performed safely at run-time. This will enable programmers to check whether a particular cast is successful or not.

C++ introduces four new casting operators:

- `static_cast`, to convert one type to another type;
- `const_cast`, to cast away the ``const-ness'' or ``volatile-ness'' of a type;

- `dynamic_cast`, for *safe* navigation of an inheritance hierarchy; and
- `reinterpret_cast`, to perform type conversions on un-related types.

All of the casting operators have the same syntax and are used in a manner similar to templates. For example, to perform a `static_cast` of `ptr` to a type `T` we write:

```
T* t = static_cast<T> (ptr);
```

As we will soon see, `static_cast` is the most general and is intended as a replacement for most C-style casts. The other three forms are for specific circumstances to be discussed below.

The `static_cast` Operator

The `static_cast` operator takes the form

```
static_cast<T> (expr)
```

to convert the expression `expr` to type `T`. Such conversions rely on static (compile-time) type information.

Subject to certain restrictions, you may use `static_cast` to convert a base class pointer to a derived class pointer, perform arithmetic conversions, convert an `int` to an `enum`, convert a reference of type `X&` to another reference of type `Y&`, convert an object of type `X` to an object of type `Y`, and convert a pointer-to-member to another pointer-to-member within the same class hierarchy.

Internally, `static_casts` are used by the compiler to perform implicit type conversions such as the standard conversions and user-defined conversions. In general, a complete type can be converted to another type so long as some conversion sequence is provided by the language.

The downcast of a base class pointer `X*` to a derived class pointer `Y*` can be done statically only if the conversion is unambiguous and `X` is not a `virtual` base class. Consider this class hierarchy:

```
class BankAcct
      { /* ... */ }
class SavingsAcct : public BankAcct
      { /* ... */ }
```

Given a base class pointer, we can cast it to a derived class pointer:

```
void f (BankAcct* acct)
{
   SavingsAcct* d1 =
     static_cast<SavingsAcct*>(acct);
}
```

This is called a **downcast**. The `static_cast` operator allows you to perform safe downcasts for non-polymorphic classes.

Note that `static_cast` relies on static (compile-time) type information and does not perform any run-time type checking. This means that if `acct` does, in fact, *not* refer to an actual `SavingsAcct` the result of the cast is undefined. Borland C++ 4.5, seemingly incorrectly, still performs the conversion, however. Your compiler mileage may vary. If you want to use run-time type information during conversion of polymorphic class types, use `dynamic_cast`. It is not possible to perform a downcast from a `virtual` base class using a `static_cast`; you must use a `dynamic_cast`.

More generally, a `static_cast` may be used to perform the explicit inverse of the implicit standard conversions. A conversion from type `S` to `T` can only be done if the conversion from type `T` to `S` is an implicit conversion. Also, the ``const-ness'' of the original type, `S`, must be preserved. You cannot use `static_cast` to change ``const-ness''; use `const_cast` instead.

One of the more common uses of `static_cast` is to perform arithmetic conversions, such as from `int` to `double`. For example, to avoid the truncation in the following computation:

```
int   total = 500;
int   days  = 9;
```

```
double rate = total/days;
```

We can write:

```
double rate =
       static_cast<double>(total)/days;
```

A `static_cast` may also be used to convert an integral type to an enumeration. Consider:

```
enum fruit {apple=0,orange,banana};
int i  1 = 2;
fruit f1 = static_cast<fruit> (i1);
```

The conversion results in an enumeration with the same value as the integral type provided the integral value is within the range of the enumeration. The conversion of an integral value that is not within the range of the enumeration is undefined.

You may also use `static_cast` to convert any expression to a `void`, in which case the value of the expression is discarded.

One interesting side effect of the old C-style casts, was to gain access to a private base class of a derived class. Consider this hierarchy:

```
class Base
{
public:
   Base() : _data(999) {}
   int  Data() const {return _data;}
private:
   int _data;
};

class Derived : private Base
{
public:
   Derived () : Base() {}
};
```

```
Derived* d1 = new Derived;
```

Normally, you should not be able to access `Data()` through the pointer `d1`. However, using an old C-style cast, we can:

```
Base* b1 = (Base*) d1;
int i = b1->Data(); // works!
```

The good news is that if you attempt to use `static_cast`:

```
Base* b1 = static_cast<Base*>(d1);
```

the compiler will correctly report that `Base` is inaccessible because it is a `private` base class.

Another unfortunate hole created in the type system by the old C-style casts results with incomplete types. Consider:

```
class X; // incomplete
class Y; // incomplete
```

The old C-style casts, let us cast from one incomplete type to another! Here is an example:

```
void f(X* x)
{
   Y* y = (Y*) x;  // works!
}
```

Thankfully, this hole has also been plugged by `static_cast`:

```
void f(X* x)
{
   Y* y = static_cast<Y*> x;  // fails
}
```

The `const_cast` Operator

The `const_cast` operator takes the form

```
const_cast<T> (expr)
```

and is used to add or remove the ``const-ness'' or ``volatile-ness'' from a type.

Consider a function, `f`, which takes a non-`const` argument:

```
double f(double& d);
```

However, we wish to call `f` from another function `g`:

```
void g (const double& d)
{
  val = f(d);
}
```

Since `d` is `const` and should not be modified, the compiler will complain because `f` may potentially modify its value. To get around this dilemma, we can use a `const_cast`:

```
void g (const double& d)
{
   val = f(const_cast<double&>(d));
}
```

which strips away the ``const-ness'' of `d` before passing it to `f`.

Another scenario where `const_cast` is useful is inside `const` functions. Remember that when you make a member function `const`, you are telling your users (and the compiler) that calling this function will not change the value of the object. However, in some cases, we find that it is sometimes still necessary to change the value of some internal data members inside a function that is `const`. For example, consider class `B`:

```
class B
{
public:
   B()  {}
```

```
    ~B() {}
    void f() const;
  private:
    int  _count;
  };
```

Suppose that, `f()`, which is declared to be `const`, must modify `_count` whenever it is called:

```
  void B::f() const
  {
    _count += 1;
  }
```

The compiler will not allow `_count` to be changed because the function is `const`. Just how does the compiler perform this magic? It turns out that the type of the internal `this` pointer helps the compiler perform this check.

Every non-static member function of a class `C` has a `this` pointer. For non-`const` member functions of class `C`, `this` has type

```
  C * const
```

This means that `this` is a **constant pointer**. In other words, you cannot change what the pointer `this` points to, after all, that would be disastrous, wouldn't it? However, you can still change what ever `this` points to (i.e., you can change data members of class `C`).

For `const` member functions of class `C`, `this` has a type of

```
 const C * const
```

Not only is `this` a constant pointer but also *what is pointed to is constant*. So the data members of `C` may not be changed through the `this` pointer. This is how the compiler ensures that you do not modify data members inside `const` functions.

Examining the member function B::f again, the statement _count is actually interpreted as this->_count. But since this has type const B * const, it cannot be used to change the value of _count so the compiler reports an error.

We can, however, use const_cast to cast away the ``const-ness'' of this:

```
void B::f() const
{
   B* const localThis =
            const_cast<B* const>(this);
   localThis->_count += 1;
}
```

Actually, you should not be casting away the ``const-ness'' of this using const_cast. C++ now has the keyword mutable for those data members whose value may be changed by const functions. By declaring _count as:

```
mutable int _count;
```

We can use the original implementation of B::f without casting away the ``const-ness'' of this.

const_cast can also be used to strip away the ``volatile-ness'' of an object in a similar manner. You cannot use const_cast for any other types of casts, such as casting a base class pointer to a derived class pointer. If you do so, the compiler will report an error.

The dynamic_cast Operator

The dynamic_cast operator takes the form

```
dynamic_cast<T> (expr)
```

and can be used only for pointer or reference types to navigate a class hierarchy. The dynamic_cast operator can be used to cast from a derived class pointer to a base class pointer, cast a derived class pointer to another derived (sibling) class pointer, or cast a base class pointer to a derived class pointer. Each of these conversions may also be applied to references. In addition, any pointer may also be cast to a `void*`.

The `dynamic_cast` operator is actually part of C++'s run-time type information, or RTTI, sub-system. As such, it has been provided for use with **polymorphic** classes -- those classes which have at least one virtual function. Use `static_cast` to perform conversions between non-polymorphic classes.

All of the derived-to-base conversions are performed using the static (compile-time) type information. These conversions may, therefore, be performed on both non-polymorphic and polymorphic types. These conversions will produce the same result if they are converted using a `static_cast`. These conversions are fairly straightforward so we won't discuss them further.

Conversions down the hierarchy from base to derived, or across a class hierarchy, rely on run-time type information and can only be performed on polymorphic types. Such conversions can now be performed safely since `dynamic_cast` will indicate whether the conversion is successful. When performing a `dynamic_cast` on a pointer, a null pointer is returned when the cast is unsuccessful. When a reference is being cast, a `Bad_cast` exception is thrown.

Let's look at the power of run-time type conversions by revisiting the bank account hierarchy introduced above with `static_cast`. Recall that when `acct` does not actually point to a `SavingsAcct` object, the result of the `static_cast` is undefined. Since `BankAcct` has at least one virtual function, it is a polymorphic class. We can use a `dynamic_cast` instead to check that the cast was successful:

```
void f (BankAcct* acct)
{
   SavingsAcct* d1 =
      dynamic_cast<SavingsAcct*>(acct);
   if (d1)
   {
      // d1 is a savings account
   }
}
```

Let's expand our bank account hierarchy to include a few more types of accounts, such as a checking account and a money market account. Let's suppose we also want to extend the functionality so that we can credit the interest for all savings and money market accounts in our database. Suppose further that `BankAcct` is part of a vendor library; we are not able to add any new members functions to `BankAcct` since we do not have the source code.

Clearly, the best way to incorporate the needed functionality would be to add a virtual function, `creditInterest()` to the base class, `BankAcct`. But since we are not able to modify `BankAcct`, we are unable to do this. Instead, we can employ a `dynamic_cast` to help us.

We add the method `creditInterest()` to both `SavingsAcct` and `MMAcct` classes. The resulting class hierarchy looks like:

```
class BankAcct    { /* ... */ }
class SavingsAcct : public BankAcct
{
public:
  // ...
  void computeInterest();
}
class MMAcct : public BankAcct
{
public:
  // ...
  void computeInterest();
}
```

We can now compute interest for an array of `BankAcct*`s:

```
void DoInterest (BankAcct* a[],
                 int num_accts)
{
  for (int i = 0; i < num_accts; i++)
  {
    // Check for savings
    SavingsAcct* sa =
```

```
      dynamic_cast<SavingsAcct*>(accts[i]);
      if (sa)
      {
         sa->creditInterest();
      }

      MMAcct* mm =
      dynamic_cast<MMAcct*>(accts[i]);
      if (mm)
      {
         mm->creditInterest();
      }
    }
  }
```

A `dynamic_cast` will return a null pointer if the cast is not successful. So only if the pointer is of type `SavingsAcct*` or `MMAcct*` is interest credited. `dynamic_cast` allows you to perform *safe* type conversions and lets your programs take appropriate actions when such casts fail.

When a pointer is converted to a `void*`, the resulting object points to the most derived object in the class hierarchy. This enables the object to be seen as raw memory. Meyers [**3**] demonstrates how a cast to `void*` can be used to determine if a particular object is on the heap.

The `reinterpret_cast` Operator

The `reinterpret_cast` operator takes the form

```
  reinterpret_cast<T> (expr)
```

and is used to perform conversions between two unrelated types. The result of the conversion is usually implementation dependent and, therefore, not likely to be portable. You should use this type of cast only when absolutely necessary.

A `reinterpret_cast` can also be used to convert a pointer to an integral type. If the integral type is then converted back to the same pointer type, the result will be the same value as the original pointer.

Meyers [3] shows how `reinterpret_casts` can be used to cast between function pointer types.

Summary

The new C++ cast operators enable you to develop programs which are easier to maintain and understand, and perform some conversions safely. Casts should not be taken lightly. As you convert your old C-style casts to use the new C++ casting operators, ask yourself if a cast is really needed there. It may be that you are using a class hierarchy in a way not originally intended or that you may be able to do the same thing with virtual functions.

References

1
Ellis, M. A., and Stroustrup, B. ***The Annotated C++ Reference Manual***. Addison-Wesley, Reading, Mass., 1990.

2
Meyers, S. ***Effective C++: 50 Specific Ways to Improve Your Programs and Designs***. Addison-Wesley, Reading, Mass., 1992.

3
Meyers, S. ***More Effective C++: 35 New Ways to Improve Your Programs and Designs***. Addison-Wesley, Reading, Mass., 1996.

4
Stroustrup, B. ***The Design and Evolution of C++***. Addison-Wesley, New York, 1994.