X3J16/93-0142, WG21/N0349

New Casts Revisited

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

By now many will have forgotten the details of the proposal for new casts #x3j16/92-0122, WG21/N0199 and #x3j16/93-23, WG21/N0231. More will not have followed the discussions since the proposal first was presented.

The purpose of the first part of this note is to give a brief presentation of the proposal as it has evolved since it was first presented.

The second part is the suggested additions to the working paper draft. Please note that this text is a reordering of the existing working paper text for casts rather than something original. ARM-style annotations are added to help comprehension.

Please note the suggested deprecation of the old cast syntax and the narrowing conversions (a deprecation is a suggestion for removal in the future, *not* a ban in this upcoming standard).

1 A New Cast Notation

Syntactically and semantically, casts are one of the most ugly features of C and C++. This has led to a continuous search for alternatives to using casts: function declarations, templates, and the relaxation of the overriding rules for virtual functions each remove the need for some cast usage. The `dynamic_cast` operator, on the other hand, provides a safer alternative to old-style casts for a specific usage. This led to a complementary approach to try to factor out the logically separate uses of casting and support them with operators similar to `dynamic_cast`:

```
dynamic_cast<T>(e)     // for run-time checked casts
static_cast<T>(e)      // for reasonably well-behaved casts
reinterpret_cast<T>(e) // for casts yielding values that must
                       // be cast back to be used safely
const_cast<T>(e)       // for casting away const
```

The current state of these ideas owes much to the efforts of the extensions working group. Dag Brück, Jerry Schwarz, and Andrew Koenig have made particularly constructive contributions.

To conserve space the discussion is almost completely restricted to the most difficult case: pointers. The treatment of arithmetic types, pointers to members, references, etc. is left as an exercise for the reader (see earlier papers and the draft manual text).

As ever, the moral of the story is: Avoid casts – of any sort – whenever possible.

2 The Problem

The C and C++ cast is a sledgehammer; `(T)expr` will – with very few exceptions – yield a value of type `T` based in some way on the value of `expr`. Maybe a simple reinterpretation of the bits of `expr` is involved, maybe an arithmetic narrowing or widening is involved, maybe some address arithmetic is done to navigate a class hierarchy, maybe the result is implementation dependent, maybe a `const` or `volatile` attribute is removed, etc. It is not possible for a reader to determine what the writer intended from an isolated cast expression. For example:

```
const Y* pc = new Y;
// ...
pv = (X*)pc;
```

Did the programmer intend to change the type from Y to X without changing the value? Cast away the const attribute ? Both? Is the intent to gain access to a private base class X of Y? The possibilities for confusion seems endless.

Further, an apparently innocent change to a declaration can quietly change the meaning of an expression dramatically. For example:

```
class X : public A, public B { /* ... */ };

void f(X* px)
{
        ((A*)px)->g();  // call A's g
        px->A::g();      // a more explicit, better, way
}
```

Change the definition of X so that A no longer is a base class and the meaning of the cast (A*)px changes completely without giving the compiler any chance to diagnose a problem.

Apart from the semantic problems with old-style casts, the notation is unfortunate. Because the notation is close to minimal and uses only parentheses – the most overused syntactic construct in C – casts are hard for humans to spot in a program and also hard to search for using simple tools such as grep. The cast syntax is also a major source of C++ parser complexity. For example:

```
int(*p);  // cast of *p to int, or
          // declaration of pointer to int called p
          // (using redundant parentheses)?
```

It is a declaration only because there is a rule saying so; int(*p) is syntactically ambiguous.

To sum up, old-style casts:

[1] are a problem for understanding: they provide a single notation for several weakly related operations;
[2] are error prone: almost every type combination has some legal interpretation;
[3] are hard to spot in code and hard to search for with simple tools;
[4] complicate the C and C++ grammar.

To have any chance of success the new cast operators must be able to perform any operation that the old casts can do. Otherwise, a reason for the permanent retention of old-style casts would have been provided. I have found only one exception: old-style casts can cast from a derived class to its private base class. There is no reason for this operation; it is dangerous and useless. There is no mechanism for granting oneself access to the complete private representation of an object – and none is needed. The fact that an old-style cast can be used to gain access to the part of a representation that is a private base is an unfortunate historical accident. For example:

```
class B { public: int b; };

class D : private B {
private:
        int m;
        // ...
};

void f(D* pd) // f() is not a member or a friend of D
{
        B* pb1 = (B*)pd;  // gain access to D.b. Yuck!
        B* pb2 = static_cast<B*>(pd); // error: can't access private. Fine!
}
```

Except by manipulating pd as a pointer to raw memory there is no way for f() to get to D::m. Thus the new cast operators close a loophole in the access rules and provides a greater degree of consistency.

Except for this restriction the new cast operators simply represents a classification of the old cast's functionality. The definition of the new casts is simply the statements from the definition of the old cast sorted under new headings.

3 The static_cast Operator

The static_cast<T>(e) notation is meant to replace (T)e for conversions such as long to int and Base* to Derived* that are not always safe but frequent and well-defined (or at least well-defined on a given implementation) in cases where the user does not want a run-time check. For example:

```
class B { /* ... */ };

class D : public B { /* ... */ };

void f(B* pb, D* pd)
{
        D* pd2 = static_cast<D*>(pb); // what we used
                                      // to call (D*)pb.

        B* pb2 = static_cast<B*>(pd); // safe conversion
        // ...
}
```

In contrast to dynamic_cast, no run-time check is required for the static_cast<D*>(pb) conversion. The object pointed to by pb might not point to a D in which case uses of *pd2 are undefined and probably disastrous.

In contrast to the old-style cast, pointer and reference types must be complete; that is, converting to or from a pointer to a type for which the declaration hasn't been seen using static_cast is an error. For example:

```
class X;  // X is an incomplete type
class Y;  // Y is an incomplete type

void f(X* px)
{
        Y* p = (Y*)px;  // allowed, dangerous
        p = static_cast<Y*>(px);  // error: X and Y undefined
}
```

This eliminate a source of errors cause by the old-style cast (Y*)px being legal. If you need to cast incomplete types use reinterpret_cast to make clear that you are not trying to do hierarchy navigation.

One way of thinking of static_cast for built-in types is that if an conversion from T to S may be performed implicitly, then static_cast can perform the inverse conversion from S to T. Except that static_cast respects ‘‘constness’’ and privacy, it can do S->T provided T->S can be done implicitly. This implies that in most cases the result of static_cast can be used without further casting. In this it differs from reinterpret_cast.

In addition, conversions that may be performed implicitly such as standard conversions and user-defined conversions are invoked by static_cast.
The effect of both dynamic_cast and static_cast on pointers to classes is navigation in a class hierarchy. However, static_cast relies exclusively on static information (and can therefore be fooled). Consider:

```
class B { /* ... */ };

class D : public B { /* ... */ };

void f(B* pb)
{
        D* pd1 = dynamic_cast<D*>(pb);
        D* pd2 = static_cast<D*>(pb);
}
```

If pb really points to a D then pd1 and pd2 get the same value. Similarly, if pb==0 both pd1 and pd2 become 0. However, if pb points to a B (only) then dynamic_cast will know enough to return 0 whereas `static_cast` must rely on the programmer's assertion that `pb` points to a `D` and return a pointer to the supposed `D` object. Worse, consider:

```
class D1 : public D { /* ... */ };
class D2 : public B { /* ... */ };
class X : public D1, public D2 { /* ... */ };

void g()
{
        D2* pd2 = new X;
        f(pd2);
}
```

Here, `g()` will call `f()` with a `B` that is not a sub-object of a `D`. `Dynamic_cast` will correctly find the sibling sub-object of type `D` whereas `static_cast` will return a pointer to some inappropriate part of the `X`.

In addition, `static_cast` can cast from any pointer to object type to a `void*` and from `void*` to any pointer to object type. The `void*` to `T*` conversion cannot be checked but is well-defined.

4 The `reinterpret_cast` operator

The `reinterpret_cast<T>(e)` notation is meant to replace `(T)e` for conversions such as `char*` to `int*` and `Some_class*` to `Unrelated_class*` that are inherently unsafe and often implementation dependent. Basically, `reinterpret_cast<T>(e);` returns a value of `T` that is a crude re-interpretation of the value of `e`. For example:

```
class S;
class T;

void f(int* pi, char* pc, S* ps, T* pt, int i)
{
        pi = reinterpret_cast<int*>(pi);
        ps = reinterpret_cast<int*>(pt);
        pt = reinterpret_cast<int*>(pc);
        i  = reinterpret_cast<int*>(pc);
        pt = reinterpret_cast<int*>(i);
}
```

The `reinterpret_cast` operator allows any pointer to be converted into any other pointer type and also any integral type to be converted into any pointer type and vise versa. Essentially all of these conversions are unsafe and/or implementation dependent. Unless the desired conversion is inherently low-level and unsafe, the programmer should use one of the other casts.

Note that `reinterpret_cast` does *not* do class hierarchy navigation. For example:

```
class A { /* ... */ };
class B { /* ... */ };
class D : public A, public B { /* ... */ };

void f(B* pb)
{
        D* pd1 = reinterpret_cast<D*>(pb);
        D* pd2 = static_cast<D*>(pb);
}
```

Here, `pd1` and `pd2` will typically get different values. In a call

```
f(new D);
```

`pd1` will point to the start of the `D` object passed whereas `pd2` will point to the start of the `B` sub-object.

`Reinterpret_cast<T>(arg)` is almost as bad as `(T)arg`. However, `reinterpret_cast` is more visible, never performs class hierarchy navigation, and does not cast away `const` or `volatile`. `Reinterpret_cast` is an operation for performing low-level and usually implementation dependent conversions – only.

Unlike `static_cast`, the result of a `reinterpret_cast` usually can't be used for anything except being cast back to its original type. Other uses cause errors. This is why pointer to function and pointer to member conversions are `reinterpret_cast`s rather than `static_cast`s.

5 The `const_cast` Operator

The thorniest issue in finding a replacement for old-style casts has been to find an acceptable way of treating `const`. The ideal is to ensure that constness is never quietly removed. For this reason `dynamic_cast`, `static_cast`, and `reinterpret_cast` was conceived as respecting constness; that is, they can't be used to ''cast away `const`.''

The `const_cast<T>(e)` notation is meant to replace `(T)e` for conversions used to gain access to data specified `const` or `volatile`. For example:

```
extern "C" char* strchr(char*, char);

const char* strchr(const char* p, char c)
{
        return strchr(const_cast<char*>(p), char c);
}
```

In `const_cast<T>(e)`, the type argument `T` must be identical to the type of the argument `e` except for `const` and `volatile` modifiers. The result is of type `T`.

6 Problems

The new cast operators would serve well for the examples described so far. The long names and the cast syntax puts off some people. That may be all to the better because one of the purposes on the new cast operators is to remind people that casting is a hazardous business and to emphasize that there are different kinds of danger involved in the use of the different operators. In my experience, the strongest dislike is expressed by people using C++ mostly as a dialect of C and by people who hasn't yet made serious use of templates. This dislike for the template-like notation of these operators wear off as people gain experience with templates.

There are, however, subtleties in the type system that makes it hard to make some conventional uses of pointers to functions convenient without allowing implicit violation of ''constness.'' Consider:

```
const char ch = 'c';
void thump(char* p) { *p = 'x'; }
```

If I can get `thump()` called with a pointer to `ch` then I have violated constness. If I manage to do so without using `const_cast` then the idea that the new cast operators (except `const_cast`) respect constness is in serious trouble.

```
typedef void (*Pcc)(const char*); // can take ch as argument
typedef void (*Pc)(char*);

Pcc p;

void f()
{
  thump(&ch); // error: argument mismatch
  p = &thump; // error: argument mismatch
  p = (Pcc)&thump; // constness (too) easily removed

  p = reinterpret_cast<Pcc>(&thump); // error

  p = reinterpret_cast<Pcc>(const_cast<Pc>(&thump)); // ok

  (*p)(&ch); // violate const
}
```

This example demonstrates that if `reinterpret_cast` could cast away constness for argument types then the system would not be const-safe. It also demonstrates that you can violate constness by a cast that *adds* an explicit `const` to an argument type. This comes to a surprise to most programmers.

Because the violation is so non-obvious it is very hard for programmers to find when the type system doesn't give a clear warning – as it doesn't now where the old-style cast simply does all conversions. I have talked to programmers who had spent days tracking down exactly this kind of bug. They were adamant that something guaranteeing that a `const` didn't quietly change its value was needed.

On the other hand, most programmers will not find the possible consequences of casting away ''cost-ness'' for argument types obvious. Many will undoubtedly assume that the compiler is wrong or that the language is stupid and revert to the old-style cast rather than finding the correct solution using both `const_cast` and `reinterpret_cast`.

Unfortunately, the problem is even worse. A more typical sequence of operations would be:

```
typedef void (*Pv)(...);
Pv q = (Pv)&thump;
Pcc p = (Pcc)q;
```

Here constness and argument types both disappear, and later reappears in a wrong form. We must either require

```
Pv q = reinterpret_cast<Pv>(const_cast<Pc>(&thump))
Pcc p = reinterpret_cast<Pcc>(q);
```

or accept the ellipsis as a fundamental hole in the type system that violates constness as well as any other type rule – yet is necessary for C compatibility.

A similar argument can be constructed using `void*`:

```
const char** pcc = &ch;
void* pv = pcc; // no cast needed:
                // pcc isn't a const, it only points to one
char** pc = (char**)pv;

void f()
{
  **pc = 'x'; // Zap!
}
```

Unless we require `const_cast` for the conversion to `void*` we have another quiet violation of const-ness. However, having `void*` unsafe might be acceptable because everybody knows – or at least ought to know – that casts *from* `void*` are inherently tricky. Similarly, people ought to consider `...` a sign that unusual care is needed.

Such examples become interesting when you start building classes that can contain a variety of pointer types (for example, to minimize generated code). For example:

```
template<class T> class link {
  const void* p;
  link* next;
public:
  void push(T* t) {
        p = t;  // implicit conversion
                // constness may be added
        // ...
  }
  T* pop() {
        // ...
        return reinterpret_cast<T*>(const_cast<void*>(p));
  }
};
```

Here, a pointer to a `T` is stored in a `const` as it must be because `T` might be const. To retrieve the pointer we must restore its type to `T*`. This cannot be done simply by `static_cast<T*>(p)` because `p` is a const pointer and `T` might not be. However, by first removing constness by `const_cast<void*>(p)` we get a pointer we can cast to `T*` whether `T*` is a const pointer or not.

Unfortunately, this solution requires the use of `void*`. There is no way of saying ''the type that is just like T except that it has no/every part const.'' Also, C++ does not have generic pointer to function and pointer to member types to match void*. Even if it have it would not be possible to write a container template that handled pointers to objects, pointers to functions, and pointers to members. This problem exists independently of what syntax is used for casting, though.

7 The Constructor Call Notation

C++ supports the constructor notation T(v) as a synonym for to the old-style cast notation (T)v. A better solution would be to re-define T(v) as a synonym for valid object construction for types with constructors and as in initializations such as

```
T val = v;
```

for types without constructors. This would require a transition because – like the suggested deprecation of (T)v – this will break existing code. For example:

```
extern int* pi;
typedef char* Pchar;
char* pc = Pchar(pi); // error: conversion of unrelated pointer types
                      // equivalent to ‘‘char* pc = pi;’’
                      // (warning only, for now)
char* pc = static_cast<Pchar>(pi); // ok

typedef B* PB;
typedef D* PD;

void f(PB pb, PD pd)
{
        PB(pd);  // ok: equivalent to ‘‘PB t; ... (t=pd,t)’’
        PD(pb);  // error: equivalent to ‘‘PD t; ... (t=pb,t)’’
                 // (warning only, for now)
}
```

In addition, narrowing implicit conversions, such as long to char should be deprecated.

8 Using the New Casts

Can the new casts be used without understanding the subtleties presented here? Can code using old-style casts be converted to use the new style casts without programmers getting bogged down in language law? For the new casts to be useful the answer to both questions must be 'yes.'

The simple strategy is to use static_cast in all cases and see what the compiler says.

If the compiler doesn't like static_cast in some case then that case is worth examining. If the problem is a const violation one must look to see if the result of the cast does lead to an actual violation. If the problem is incomplete types, pointer to functions, or casting between unrelated pointer types one must try to determine that the resulting pointer is actually cast back again before use. If the problem is something like a pointer to int conversion one ought to to think harder about what should be going on, but if one doesn't have time to think reinterpret_cast does exactly what an old-style cast would do.

In all cases, it would be better if the cast could be eliminated.

9 Implementation

Syntax analysis for the new casts is trivial.

Code generation for the new casts is easy because every C++ compiler already has the code for performing every action the new casts can specify. The union of their actions is identical to those of old-style casts except for the accessibility check for base classes. Every compiler has code for such a check somewhere also.

In addition to finding the one action for each combination of two types the compiler has to select actions based on two types and the kind of cast requested.

The only compatibility problem is the three new keywords static_cast, reinterpret_cast, and const_cast. These were chosen partially because they are unlikely to be identifiers in existing programs.

The new casts have been implemented at least once.

10 Conclusions

So what can we do about old-style casts? We could not ban them; that would be an ill-advised introduction of a major incompatibility. Instead, the new cast operators provide the individual programmer and individual organizations with a way to avoid the insecurities of old-style casts where type safety is more important than compatibility. That done, old-style casts could be deprecated, giving hope that they could be eliminated some time in the next century. The new casts can also be supported by compiler warnings for the use of old-style casts and narrowing implicit conversions.

The main argument for the new cast operators is that there is currently no alternative in C++ for programmers who care about the safety of the casts that they must write. Not all casts can be eliminated, yet, there is no mechanism that clearly distinguishes the logically different conversion operations, respects the access rules, and respect constness. Thus, programmers are forced to use the inherently error-prone old-style casts.

This problem is likely to increase in seriousness over the years as the general quality of code improves and tools assuming type safety (including const safety) gets into widespread use – in C++ and in other languages. The new cast operators – or something long that line – can provide an evolution path to a safer, yet no less efficient, style of programming.