x3j16/92-0122, WG21/N0199

An Alternative to Old-style Casts

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is a proposal for a set of conversion operators that can be used instead of traditional casts. The purpose of the new operators is to cleanly integrate conversions based on run-time type information into C++ and to provide safer and more easily checked alternative conversion operators.

1 Introduction

This proposal it the direct outcome of work on run-time type information and in particular the issue of using run-time type information to provide a checked cast from a base to a derived class. Along the way, general issues of conversions, notation for conversions, and the safety of conversions became key to several discussions at standards meeting, conferences, mail reflectors, and elsewhere. This is a proposal for four conversion operators

```
dynamic_cast<T>(e)       // for run-time checked casts
static_cast<T>(e)        // for reasonably well-behaved casts
reinterpret_cast<T>(e)   // for horrible casts
const_cast<T>(e)         // for casting away const and volatile
```

A appendix discusses the need for "casting away `const`."

Issues of type safety, when it is necessary to break it, and how that can be done with the least chance of accident dominated much of the discussion. Issues of access control, constness, and abstraction in general influenced this design. As ever, it was considered essential that no concern of efficiency or basic ability to perform a task, however low-level, should force people to use C instead of the proposed C++ features. This design is believed to be a complete alternative to old-style casts.

I do not, however, propose to ban old-style casts; that would be an ill-advised introduction of a major incompatibility. Instead, the proposed facilities provide the individual programmer with a way to avoid the insecurities of old-style casts where type safety is more important than compatibility.

This discussion assumes that the static type of an operand is correct. If someone already has "lied," say, so that a pointer of type `B*` doesn't really point to a `B` or something derived from `B` and isn't `0` either, then then the result of conversions of that pointer are undefined.

When the term sub-object is used "sub-object representing a base class" is meant. There is no proposal to define casts from members to their enclosing object.

2 Problems with Old-style Casts

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

Did the programmer intend to change the type from `X` to `Y` without changing the value? Cast away the

`const` attribute ? Both? Is the intent to gain access to a private base class `Y` of `X`? 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
}
```

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 used 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 one the sources of C++ parser complexity.

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.

The aim of this proposal is to eliminate these problems by providing alternatives to the various uses of old-style casts.

3 The `dynamic_cast` Operator

The

```
dynamic_cast<D*>pb;
```

operator is meant to replace

```
(D*)pb
```

for casts from a base to a derived class. `Dynamic_cast` examines the object referred to by its argument and if that object is of the desired result type a valid pointer is returned; otherwise the result is `0`. That is, `dynamic_cast` is a new variation of the "checked cast"

```
(?D*)pb
```

suggested in the paper on Run-time Type Identification [1]. A more extensive argument for the introduction of `dynamic_cast` into C++ can be found in that paper. For example:

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

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

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

        B* pb2 = dynamic_cast<B*>(pd); // safe conversion, no
                                       // run-time check needed
        // ...
}
```

The calls

```
f(new B,0);
f(0,0);
```

would both cause pd2 and pb2 to be initialized with 0. On the other hand, the calls

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

would cause pd2 and pb2 to be initialized with a pointer to the appropriate D objects.

Operands

For dynamic_cast<T>(arg), T must be a pointer to object type or a reference to object type. The (static) type of arg must be a polymorphic pointer or reference to type; that is a pointer or reference to a type with at least one virtual function. In addition, dynamic_cast can cast from any polymorphic pointer to an accessible base class or void*. In the latter case, the void* will point to the complete object that the pointer referred to a sub-object of. For example:

```
B* pb = new D;
void* pv = dynamic_cast<void*>(pb);
// pv points to the start of the D object
```

Note that dynamic_cast<void*> differs from the old-style cast (void*) - and from the static_cast<void*> operator defined below - by "finding" the start of the object.

Cast to Pointer Types

Consider

```
dynamic_cast<T*>(p)
```

The result is of type T*. The argument, p, must be of pointer type or the constant 0. The result of dynamic_cast<T*>(0) is 0.

Let us call p's (static) type X*. If X* is an incomplete type dynamic_cast<T*>(p) is an error. The reason for this is that to do a dynamic_cast we need enough information about the argument's type to be able to retrieve type information from the argument for use in a run-time type check. If T is an accessible base class of X, the conversion is safe and will be done without a run-time check.

If T is not a an accessible base class of X then X must be a polymorphic type. In this case, T need not be a complete type. The types T and X are compared at run time and if *p is a subobject of a T or a sub-object of a type that has a unique public base of type T then a pointer to that T is returned.

Basically, dynamic_cast is navigation in a class hierarchy. Given a pointer to some sub-object identified by its class the result will be to a base, derived, or sibling class to the class of the original pointer (or that class itself) and the resulting pointer will point to the appropriate sub-object of the original object.

Access Protection

The dynamic_cast operator differs from old-style casts, and from the proposal for (?T*)p, by respecting (simplified) access rules. For example:

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

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

void f(D* pd)
{
        B* pb = dynamic_cast<B*>(pd); // error: access violation
}
```

That is, errors that can be detected by the compiler are reported at compile time. Errors that cannot are indicated at run-time. For example:

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

void f(B* pb)
{
        A* pa = dynamic_cast<A*>(pb); // use run-time type information
}

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

void g()
{
        f(new DD);
}
```

Here pa becomes 0 because the base A is inaccessible to DD.

Consider,

```
void f(B* pb)
{
        A* pa = dynamic_cast<A*>(pb);
}
```

If *pb is a sub-object of an A then a pointer to that A is the result. Otherwise, let X be the (dynamic) type of the object pointed to by pb. If A is a unique public base of X then a pointer to the A in X is the result. Note that private and protected bases are not considered when looking for A in X. This simplified (and slightly restrictive) rule mirrors the behavior of exception handling and can be checked without considering the (static) context of the cast.

The reason to respect access rules is to avoid accidental violation of abstractions. For example:

```
class SS : public Set, private Slist {
        slink* current_element;
        // ...
};
```

Here the representation of an SS is an Slist and a current_element. Presumably every operation on an SS must ensure that the Slist and the current_element corresponds. Consider:

```
void f(set* pset)
{
    Slist* pslist1 = dynamic_cast<Slist*>(pset);  // 0: Slist is private
    SS* pss = dynamic_cast<SS*>(pset);            // ok
    // ...
}
```

In the first cast (to to private sibling class Slist) it is too dangerous to return a pointer to the sibling sub-object. The programmer doesn't actually know the type of the object enclosing the Set and Slist siblings and therefore can't make an informed judgement whether breaking the protection is warranted or not. Anyway, that would still leave the private member current_element inaccessible. We considered providing an operation access_cast<Slist*>(pss) to allow the user to do what can be done with an old-style cast (Slist*)pss, but found such an operation neither necessary nor sufficient for the purpose of gaining access to a representation.

The fundamental rule for access is that it is always granted by the class and never unilaterally grabbed by other code. This implies that a function such as f() above can gain access to SS's representation only if SS has made arrangements to allow it. For example:

```
class SS : public Set, private Slist {
        Slink* current;

        // ...
public:
        // ...
        Slist* get_list();
        Slink*& current();
};

void f(set* ps)
{
        SS* pss = dynamic_cast<SS*>(pss);
        if (pss) {
                Slist* psl = pss->get_list();
                // ...
        }
        else {
                // ...
        }
}
```

It is with some trepidation that I suggest banning casting to a private base because that leaves one thing that can be done by an old-style cast but not by the new cast operators. However, I consider the ability to cast to a private base class (for example, `(slist*)pss`) a misfeature that is neither necessary nor sufficient to gain access to a private representation in general. It also violates the fundamental rule about granting of access. Thus casting to a private base class should not be provided by a replacement for old-style casts. Note that there has been no demand for an operation for grabbing access to private data in general.

Cast to Reference Types

A dynamic cast to a reference type

```
dynamic_cast<T&>(obj)
```

is similar to a pointer cast except that `obj` must be a reference or an object and failure to convert is indicated by throwing a `Bad_cast` exception rather than returning `0`.

Cast to Other Types

`Dynamic_cast` can only be be used for pointers and references to polymorphic types. `Dynamic_cast` cannot be defined for user-defined types and cannot be redefined for pointers or references by the user.

Why not? The idea is to reserve `dynamic_cast` for inquiries about the actual type of polymorphic objects. Other conversions can be performed by `static_cast` (§4) which is a closer equivalent to the old-style cast as used for well-behaved conversions between non-pointer types. Allowing `dynamic_cast` to be used for other conversions that depends on the value of an object and may fail (at run time), such as a `long` to `int` would be logical, but not necessary because such operations can be defined without new conversion operators.

Uses of Dynamic Casts

Dynamic casts have two main uses:

[1] Casting from a base to a derived class (downcasting).

[2] Casting from a base to a public sibling class (cross-hierarchy casting).

Both can be seen as gaining access to other valid interfaces for an object that we already have a handle on. For example:

```
void f(A* pa)
{
        X* px = dynamic_cast<X*>(pa);
        B* pb = dynamic_cast<B*>(pa);
        // ...
}
```

would yield non-zero px and pb if pa pointed to a class like Y:

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

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

Comparison with Old-style Casts

Dynamic casts provide a mechanism that isn't currently directly supported in C++. To determine whether an object is of a given derived class requires scaffolding in the base class of some form; for example, a virtual function [2] or a type field. Once a pointer to a derived class is found, old-style casts allow us to break the access protection by casting to a private base class. This is not desirable and in any case an old-style cast is an insufficient mechanism for gaining access because it gives access to private bases only and not in general to private members.

A fundamental advantage of `dynamic_cast` over old style casts is that it does a specific operation so that compile time errors can be issued for misuses, such as attempting a `dynamic_cast` on an object of a type that does not support it, and so that the meaning of a `dynamic_cast` is evident from the source code.

It is clearly harder to type `dynamic_cast<D*>(p)` than to type `(D*)p`. This is deliberate. The aim is partly to make casting somewhat unattractive and - more importantly - to make uses of casts highly visible in code and to make it easy to search for with simple tools, such as Unix's `grep`. Ideally, the length of the cast operators would reflect the ugliness of their semantics. Casts are occasionally necessary, but the ideal must be to avoid them.

4 The `static_cast` Operator

The

```
static_cast<T>e;
```

operator 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); // this is 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.

Operands

For `static_cast<T>(arg)`, `T` must be a pointer, reference, arithmetic type, or enumerations. Pointer to function and pointer to member conversions are allowed following the rules for old-style casts†. The argument, `arg`, must come from the same set of types and match `T` as described below. All 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;
class Y;

Y* f(X* px)
{
        return static_cast<Y*>(px);     // error: incomplete types
}
```

In addition, user-defined conversions are invoked by `static_cast`.

Cast to Pointer Type

`Static_cast` can convert from a derived class to a unique, accessible base class and from a non-virtual base class to a derived class. `Static_cast` cannot convert between apparently unrelated pointer types where an inheritance relationship isn't known. For example:

```
class X { /* ... */ };
class Y { /* ... */ };

void f(X* px)
{
        Y* py = static_cast<Y*>(py);    // error: unrelated types
}
```

Like, `dynamic_cast`, `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. So they do if `pb==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 returns a pointer to the supposed `D` object. Worse, consider:

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

† These rules could probably be improved by introducing equivalents to `void*` for pointers to functions and pointers to members, but that is not relevant to this discussion.

```
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 sub-object of the `X`. Fundamentally, `static_cast` cannot check the assumption that the types of its argument and its result type are related through some base/derived relationship and relies on the programmer. On the other hand, `static_cast` can be used where no run-time information is available and will typically be significantly faster.

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.

Cast to Pointer to Member Type

`Static_cast` is defined exactly as old-style cast for pointers to members except that there are no conversions to or from non pointer to member types.

Cast to Pointer to Function Type

`Static_cast` is defined exactly as old-style cast for pointers to functions except that there are no conversions to or from non pointer to function types.

Cast to Reference Type

A static cast to a reference type

```
static_cast<T&>(obj)
```

is similar to a pointer cast except that `obj` must be a reference or an object.

Cast to Arithmetic Type

The usual arithmetic conversions are performed by `static_cast`. For example:

```
void f(int a, double d)
{
        int a2 = static_cast(d);
        double d2 = static_cast(a);

        int a3 = d;
        double d2 = a;
}
```

Some of these conversions are guaranteed to be mathematically correct, but most, such as the two examples above, are not because significant information can be lost through narrowing. In the example above, not all `doubles` can be represented as `ints` and, one can provide a legal C++ implementation where the largest `int` can't be represented exactly as a `double`.

The point of using explicit casts for conversions that are also performed implicitly is to make it possible for compilers to warn against narrowing conversions and eventually be able to disallow them (§8).

Cast to Enumeration

A `static_cast` can be user to turn an integral value into an enumeration value:

```
enum A { a, b, c };

void f(int i)
{
        A x = static_cast<A>(i);
}
```

The result is undefined unless the value of `i` is the same as the integer value of one of `A`'s enumerators.

User-Defined Conversions

User-defined conversions are applied implicitly (see §5) and `static_cast` will perform all implicit conversions. This leads to a question about whether to apply the built-in conversion or the user-defined conversion in the (rare) case where both are defined. For example:

```
class Y { };

class X : public Y {
public:
        operator Y&();
};


void f()
{
        X x;
        Y& r1 = static_cast<Y&>(x);  // ?
        Y& r2 = (Y&)x;               // invoke X::operator Y&();
}
```

For compatibility, I suggest that the user-defined conversion is invoked exactly as for old-style cast. This preserves the important property that in the cases that are legal for `static_cast` the semantics is unchanged from that of old-style casts. An alternative would be to deem such examples ambiguous for both old-style casts and `static_cast`.

Run-time Checking

May an implementation apply run-time checking to compensate for `static_cast`'s lack of guarantees? As for every other use of an undefined operation, an implementation of `static_cast` may detect undefined behavior and report it at compile time or run-time. Note that the result of `(char)666` is undefined even in C (assuming 8 bit characters) and an implementation is allowed to do anything, including a core dump upon encountering it.

The interesting question is if a C++ implementation may throw an exception if it detects `static_cast<char>(666)` at run time. My suggested answer is 'yes.' This allows exception handling and optional run-time checking to interact. Note that to make sense equivalent decisions must be made about the behavior of range errors for arrays and other common sources of undefined, yet, detectable behavior.

Comparison with Old-style Casts

`Static_cast` implements the subset of old-style casts that deals with well defined relationships between types. The main advantage of `Static_cast<t>(e)` over `(T)e` is therefore that one cannot accidentally cast unrelated types. A related advantage is that conversions of incomplete types are not performed by `Static_cast<t>`. Naturally, uses of in a program `Static_cast` is also easier to find.

5 The Constructor Call Notation

In addition to the old-style cast notation `(T)v`, C++ supports the constructor notation `T(v)`. I propose `T(v)` as a synonym for valid object construction as in initializations such as

```
T val = v;
```

(that we don't have a good name for) rather than `(T)arg`. This requires a transition because - like the suggested deprecation of `(T)arg` - this will break existing code. For example:

```
typedef char* Pchar;

void f(int* pi)
{
        char* pc = Pchar(pi); // conversion of unrelated pointer types
        // ...
}
```

I suspect that code of this kind is too common to ban outright. Consequently, `T(v)` will have to remain a synonym for `(T)v` for this standard, but I would like to see this equivalent "deprecated" (see §8) with the aim of having the meaning of `T(v)` be defined as construction

```
(temp_of_type_T=v,temp_of_type_T)
```

in a future standard. For example:

```
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)
}
```

I would hope to see compilers issue warnings for uses of `T(v)` that do not fall into the subset defined by construction.

User-defined Conversions

User-defined conversions, that is constructors taking a single argument and conversion operators are called implicitly and will therefore be invoked by the constructor notation. For example:

```
class X {
public:
        X(int);
        X(int,int);
        operator int();
};

X x1 = 1;          // invoke X::X(int)
X x2 = X(1);       // invoke X::X(int)
int i = x1;        // invoke X::operator int()
int i = int(x2);   // invoke X::operator int()
```

In addition, the constructor notation is of course used for multi-argument constructors:

```
X x3 = X(1,1);     // invoke X::X(int,int)
```

The decision to restrict invocation of user-defined conversion to the constructor notation (functional notation) was only reached slowly. The observation that user-defined conversions (rightly) belonged to the set of conversions applied implicitly was crucial. In that sense, user-defined conversions are classified as more fundamental and "safer" than the (built-in) conversions that requires explicit application.

The alternative would be to have user-defined conversions invoked by `static_cast` only (because user-defined conversions are chosen based on static information and may by narrowing), invoked by `dynamic_cast` only (because user-defined conversions can look at the value involved and may indicate failure through distinguished values, such as `0`, or through exceptions), or by both `static_cast` and `dynamic_cast`. The alternative of having different kind of user-defined conversions for the different kinds of cast operators was rejected as too complicated.

Because `static_cast` will invoke implicit conversions it can be used to invoke user-defined

conversions. This helps the writing of templates.

Because `dynamic_cast` is assumed to rely on specific run-time information, allowing it to invoke user-defined conversions could lead to confusion in the (rare, reference cast) cases where there is a choice between the built-in semantics and a user-defined conversion. I see no reason to introduce this possibility for confusion given that user-defined conversions can already be invoked implicitly, by the constructor notation, and by `static_cast`.

One crucial test for the classification of conversion operations implied by the choice of notation is how one can write templates. The choice made is based on the assumption that user-defined types with conversions behave like built-in types like `char` and `int` so that a template can deal uniformly with those types. Pointers on the other hand, are best manipulated separately in templates that knows about pointer semantics. User-defined types that rely on dereference operations will be grouped with pointers in this respect. Similarly, templates that do not dereference pointers can treat pointers like other types.

If this is not sufficient users can define their own conversion operators.

6 The `reinterpret_cast` operator

The

```
reinterpret_cast<T>e;
```

operator 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 `B` object passed whereas `pd2` will point to the start of the `D` object of with the `B` passed is a sub_object.

Operands

For `reinterpret_cast<T>(arg)`, `T` must be a pointer, reference, arithmetic type, pointer to function or pointer to member. Basically `reinterpret_cast<T>(arg)` provides all of the meanings of `(T)arg` not provided by `dynamic_cast`, `static_cast`, and `const_cast`.

Comparison with Old-style Casts

`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.

7 The `const_cast` Operator

The

```
const_cast<T>(e);
```

operator 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<const char*>(p), 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`.

Redundancy

Clearly, `const_cast<T>` is redundant in that we could define a `const_cast` operator that simply stripped off all `const` and `volatile` specifiers from the (unmentioned) type of its operand. However, this lack of redundancy would eliminate all possibilities for consistency checks and leave `const_cast` syntactically different from other casts.

Comparison with Old-style Casts

When using an old-style cast to cast away `const` there is no way for a compiler to know whether the type was (also) meant to be changed. For example:

```
const X* pc;
// ...
Y* py = (Y*)pc;              // ok, but what is the intent?
Y* pv = const_cast<Y*>(pc);  // error: pc is not a Y*
Y* pz = static_cast<Y*>(pc); // error: *pc is const
Y* pv = static_cast<Y*>(const_cast<X*>(pc)); // ok
```

`Const_cast` was reluctantly introduced after several attempts to find sufficient reason to ban "casting away `const`." See Appendix A for details.

8 Deprecating old-style casts

We cannot prohibit old-style casts. It is not the business of a standards committee to break all existing programs and I conjecture that there exist very few significant cast-free C++ programs. By providing better alternatives we can, however, enable individual programmers to adopt a cast-free style. Implementors can support such programmers by compiler options that warnings against old-style casts, implicit narrowing conversions, and uses of the function-style (constructor-style) casts that are not equivalent to implicit

conversion. I think we should encourage programmers to move away from these constructs. A statement to that effect should be part of the standard. These constructs could be put on a list of things we "expect not to be part of the next standard after this one" similar to the list found in the C standard.

9 Why Casts?

Casts are inherently ugly (semantically as well as syntactically), so why have them? In particular, why replace a single cast operator with four new ones? In addition to the points made above, it should be noted that if we support run-time type checks through a checked cast, such as `(?T*)p` or `dynamic_cast<T*>(e)` in addition to old-style casts, such as `(T)e`, then we already have two casts. The only way of keeping the number of casts to one is to

[1] Not to introduce a checked cast operation.
[2] Use the old-style cast notation for checked casts (also).
[3] Use a completely different notation for converting from a base class pointer to a derived class pointer based on a run-time type check.

It seems to me that there has been overwhelming sentiment against [1] and we tried [2] without any luck. The argument against [1] is basically "but we need it badly and we already fudge it in many incompatible ways." The argument against [2] is basically "but having old-style casts serving yet another purpose is far too confusing and some code will silently change meaning, slow down significantly, or both." I have only seen one potentially acceptable solution along the lines of [3] (suggested by someone at the London meeting, I don't recall who (sorry)): Allow conversion from base to derived in initializations placed in conditions (only). For example:

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

void f(B* pb)
{
        D* pd1 = pb;    // error: no conversion

        if (D* pd2 = pb) { // run-time checked conversion
                // here we have a D pointer to by pd2
        }
        else {
                // here we don't
        }
}
```

I consider this the only realistic alternative to a set of checked casts. A disadvantage is that it does not open a path to the abandonment of old-style casts and narrowing conversions and that it requires the introduction of a new variable to perform a run-time checked conversion. In particular, it implies that a run-time checked conversion can be done in a statement context only. I also conjecture that a fair amount of trouble and confusion would arise from this scheme because there is no new syntax for people to identify the new mechanism with.

10 Conversion Strategies

Given a program, how would one convert it to using the new cast notation? Before the new cast operators are available one could find all uses of casts (not an easy task without compiler support, but trivial given such support) and try to determine what the intent of each cast was. Then each cast can be eliminated, changed to a macro with a name reflecting the intent, or have a comment added to indicate its intent. Some organizations have already independently adopted such measures to alleviate the maintenance problems caused by old-style casts. Significantly, the classification of casts reflected in the definition of the four cast operators proposed here appears to have been independently discovered in several places.

Given, the new operators, the most effective strategy will probably be to change all casts to `static_cast` and have the compiler tell which ones are "ill behaved." Then the program can then be studied with the aim of eliminating casts and having the necessary ones expressed in terms of the most appropriate (new) cast operators.

11 References:

[1] Stroustrup, Bjarne: *The C++ Programming Language (2nd Edition)*. Addison Wesley. 1991.
[2] Stroustrup, Bjarne and Lenkov, Dmitry: *Run-time Type identification for C++ (Revised)*. Proc USENIX C++ Conference. August 1992.

12 Acknowledgements

Many people have provided inputs to this proposal. The most significant contributions were by Andrew Koenig and Jerry Schwarz. Several useful refinements were caused by the discussion of casting in general and of the notion of `const` that we had in the extensions working group in Boston.

13 Appendix A: Casting Away `const`

Ideally, we wouldn't have an operation for casting away `const`. So let's consider if we could make do without one. Consider:

```
const int x = 7;
void f(const int* p);

void g()
{
        f(&x);
        if (x == 7) { /* ... */ }
        // ...
}
```

Is a compiler entitled to optimize away the `x==7` test because it knows that `x` is a constant valued 7? Alternatively, must it assume that despite its promise `f()` might have changed the value of `x`?

```
void f(const int* p)
{
        ((int*)p)++;     // fooled you!
        // const_cast<int*>(p)++;
}
```

As usual, the cast notation obscures what is really going on. I seems desirable that a compiler should be entitled to optimize simple examples like this so that casting away const in these cases therefore is inherently unsafe.

A compiler is allowed to make the optimization because any `const` object of a type without a user-specified constructor may be put in ROM. Anyone who casts away `const` for an object of a ROMable type and try to change its value is asking for trouble by writing a program that is not guaranteed to work.

However, not all calls of `f()` leads to undefined behavior. Consider:

```
void g()
{
        int a = 1;
        f(&a);
        if (a == 2) { /* ... */ }
        // ...
}
```

Here a promise not to modify the argument is broken, but the result is well-define because the object modified wasn't defined to be `const`. In this case, the compiler is not entitled to optimize away the test.

Where `const` is used in interfaces to give an illusion of invariance of larger objects, casting away `const` is allowed and well defined. For example:

```
class X {                          // #1 caching of values
        V cache;
        // ...
public:
        V f(int arg) const
        {
                if (same argument as last time) return cache;
                // compute
                return ((X*)this)->cache = v;
                // return (const_cast<X*>(this))->cache = v;
        }
};

X x1;
const X x2;
// ...
V v1 = x1.f(1);
V v2 = x2.f(2);
```

The cast is necessary to allow an update of the `X.cache`. Disallowing casting away `const` invalidates this technique. We could re-allow it by accepting the proposal (now being considered by the committee) to allow a data member to be specified "never `const`:"

```
class X {
        ~const V cache;
        // ...
public:
        V f(int arg) const
        {
                if (same argument as last time) return cache;
                // compute
                return cache = v;
        }
};

X x1;
const X x2;
// ...
V v1 = x1.f(1);
V v2 = x2.f(2);
```

or something like that. This makes the manipulation of `const`ness declarative but I fail to see the really significant gain unless we could actually outlaw casting away `const`. This would force people to rewrite:

```
class X {
        V* cache;
        // ...
public:
        V f(int arg) const
        {
                if (same argument as last time) return cache;
                // compute
                return *cache = v;
        }
};

X x1;
const X x2;
// ...
V v1 = x1.f(1);
V v2 = x2.f(2);
```

If we were dealing with a new language rather than a standards effort I think I'd prefer the re-write to the

~const patch on the const concept. Unfortunately, there is a more general technique that relies on casting away const that - as far as I can gather - is important in real code:

```
void T::f() const                          // #2 abstract state
{
        T* This = (T*)this;
        // T* This = const_cast<T*>(this);
        ++This->changing_this_doesnt_change_abstract_state;
        // ...
}
```

Basically const exists at the user level and the implementation operates by its own rule: "So as soon as we descend into the untyped representation, consts are generally stripped off. (Keeping track of them, and stripping them only when absolutely necessary, wouldn't decrease the likelihood of a bug.)" The ~const notion for data members doesn't help much here except in the extreme case where every member is ~const so that the notion of constness is completely disabled for that type. I can imagine approaches to this problem in the form of more modifiers on member functions, but a third class of examples seems even worse from the perspective of managing without an operator for casting away const.

Casting away const can also be used to allow shared representation:

```
                                           // #3 shared representation
template <class T>
class List_of_p : private List<const void*> {

        T* first();  // must cast away const when T is non-const

        // ...
};
```

This really requires casting away const (and happens to be safe). No elaboration of class declaration along the lines of ~const seems to help.

The most prominent example of this use of casting away const is the implementation of C++ functions by calling C functions:

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

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

Yet another use of casting away const is various forms of data gathering:

```
complex f(const complex& aa)     // #4 data gathering
{
        complex c1 = aa+5;
        complex c2 = aa+7;
        // ...
}
```

Can f() rely on + not changing aa to the point where it can copy aa into a register and not copy it back again? Not if complex has an extra member used to count the number of operations performed. If so, the optimization will defeat the data gathering.

My conclusion is that casting away const is necessary in some form even if we were not constrained from breaking existing programs. This conclusion is independent of the outcome of the ~const effort. I now see ~const primarily as a potential mechanism for minimizing casts, rather than as a mechanism that potentially could eliminate casting away const. This implies that the problem is to find the most suitable form; const_cast<T>(v) is the result of these considerations.