Doc. Number: X3J16/92-0028
WG21/N0105

Run-Time Type Identification for C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Dmitry Lenkov

HP Language Labs

ABSTRACT

This paper describes a proposal for a mechanism for run-time type identification and checked type casts. The mechanism is simple to use, easy to implement, and extensible. This proposal evolved through a series of earlier proposals and ideas. Experimental implementations exist. Warning: This is a proposal and the features described may never be accepted into C++.

1 Introduction

The need to use run-time type information explicitly in a program arises when one has a pointer or a reference to some object that may be of a class derived from the base class known at compile time and one wants to perform some operation that makes sense only for an object of the derived class. For example, given the classes

```
class dialog_box : public window {
    // ...
public:
    // ...
    virtual int ask();
    // ...
};

class dbox_w_str : public dialog_box {
    // ...
public:

    // ...
    int ask();
    virtual char* get_string();
    // ...
};
```

we may call `ask()` for every `dialog_box` but may call `get_string()` only for `dialog_boxs` known to be `dbox_w_strs`. Given only a `dialog_box*` how can we figure out if it really points to a `dbox_w_str`?

There are several ways of defining the `dialog_box` and `dbox_w_str` classes so that the answer can be found. The most popular are to place a type field in `dialog_box` and/or define a virtual function in `dialog_box` that gives the answer. Many C++ libraries provide mechanisms for explicit use of run-time type identification (RTTI) for their classes [2,3,5, and 10] and detailed explanations of how to implement them can be found in [4,8,9]. What is proposed here, however, is a language supported mechanism:

```
void my_fct(dialog_box* bp)
{
    if (typeid(bp) == Typeid<dbox_w_str>()) { // is bp a dbox_w_string?
        dbox_w_str* dbp = (dbox_w_str*)bp;

        // here we can use dbox_w_str::get_string()
    }
    else {

        // 'plain' dialog box
    }

    // ...
}
```

The operator `typeid` returns an object that identifies the type the object pointed to by its pointer argument. Similarly, the operator `Typeid` returns an object that identifies the type of its type name argument. In particular, `typeid(bp)` returns an object that allows the programmer to ask some questions about the type of the object pointed to by `bp`. In this case, we asked if that type was identical to the type `dbox_w_str`.

This is typically *not* the right question to ask. The reason to ask anything at all is to see if some detail of a derived class can be safely used. To use it, we need to obtain a pointer to the derived class. In the example, we used a cast on the line following the test. Also, we are typically not interested in the *exact* type of the object pointed to, but only in whether we can perform the cast. Instead, this question can be asked directly:

```
void my_fct(dialog_box* bp)
{
    dbox_w_str* dbp = (dbox_w_str*)bp;

    if (dbp) {

        // here we can use dbox_w_str::get_string()
    }
    else {

        // 'plain' dialog box
    }

    // ...
}
```

The type cast operation converts its argument to the desired type if possible and otherwise it returns `0`. That is, we change the semantics of casting to perform the run-time test. The compatibility implications of this change are less than what they appear at first glance and will be discussed in §4.

Such a cast is often called *safe* because the result of an attempt to cast a pointer to a type that is wrong for the object it points to results in the well-defined pointer `0`. It is also often called a *downcast* because many people draw classes with derived classes below their bases. To avoid making users overconfident, we prefer to call such casts *checked* rather than safe.

Naturally, an implementation of the checked cast will rely on the same kind of information as the `typeid` operator and share a large part of its implementation.

There are several advantages to merging the test and the cast into a single checked cast operation:

- The cast notation is less verbose than alternatives using named operations.
- The cast notation does not require the introduction of additional key words.
- By using the information available in the type information objects it is possible to cast from a virtual base class to a derived class; see §5.
- This notation makes it impossible to mismatch the test and the cast.

As examples of such mismatches, consider:

```
void my_fct(dialog_box* bp)
{
    if (typeid(bp) == Typeid<dialog_box>()) {
        dbox_w_str* dbp = (dbox_w_str*)bp;

        // here we can use dbox_w_str::get_string()
    }

    // ...
}
```

where the user checked against the type of the base class `dialog_box` instead of the derived class `dbox_w_str`, and

```
void my_fct(dialog_box* bp)
{
    if (typeid(bp) != Typeid<dbox_w_str>()) {
        dbox_w_str* dbp = (dbox_w_str*)bp;

        // here we can use dbox_w_str::get_string()
    }

    // ...
}
```

where the user applied the explicit cast on the wrong branch of the `if` statement. Both kinds of errors have been seen in real systems.

The notation is still redundant in that `dbox_w_str` is mentioned twice in

```
dbox_w_str* dbp = (dbox_w_str*) bp;
```

However, removing that redundancy would leave the programmer without a clearly visible clue that something "interesting" is going on, and would also allow the use of the checked cast in contexts where the desired type is not clearly visible. This redundancy also enables an added degree of checking:

```
extern void f(dbox_w_str* dbp);

// ...

void g(dialog_box* bp)
{
    f(bp); // error: cannot (implicitly) convert
           // from a base to a derived class

    f((dbox_w_str*)bp); // ok: explicit cast
}
```

2 Declarations in Conditions

As a final simplification we might adopt the Algol68 notion that declarations yield values and thereby allow declarations in conditions. We could then write this:

```
void my_fct(dialog_box* bp)
{
    if (dbox_w_str* dbp = (dbox_w_str*) bp) {

        // use 'dbp'
    }

    // ...
}
```

The value of a declaration is the value of the declared variable after initialization. To avoid syntax

problems, we do not suggest that declarations can appear everywhere an expression can (which would be the cleanest semantic notion) but only that declarations of a single initialized variable can appear in the condition part of `if`, `for`, `while`, and `switch` statements. Allowing declarations in conditions of conditional expressions and `do` statements seems to add complications rather than utility so we don't propose that.

This extension is, of course, independent of the notion of run-time type identification. It simply attacks the problem of use of uninitialized variables directly. For example:

```
void f(Iter<Name> it)
{
    while (Record* r = it.next()) {
        // process '*r'
    }
}
```

The scope of a variable declared in a condition is the statement or statements controlled by the condition. In particular, a variable declared a condition of an `if` statement is in scope in the `else` part of that statement. Naturally, the variable will most often be 0 in the `else` statement, but it is possible to construct examples where it is not. For example, consider a class `X` with an `operator int()`:

```
void g(int b)
{
    if (X x1 = b) {
        // we get here if x1.operator int()
        // doesn't yield 0
    }
    else {
        // x1 has a meaningful value even here
    }
}
```

It is not legal to declare a variable with the same name in both the condition and in the outermost block of a statement controlled by the condition. For example:

```
if (Name* p = find(s))
{
    char* p; // error: multiple definition of 'p
    // ...
}
```

This rule parallels the rule that an argument name may not be redefined in the outermost block of a function:

```
void f(Name* p)
{
    char* p; // error: multiple definition of 'p'
    // ...
}
```

3 Uses and Misuses of RTTI

One should use explicit run-time type information only when one has to; static (compile-time) checking is safer, implies less overhead, and – where applicable – leads to better structured programs. For example, RTTI can be used to write thinly disguised switch statements:

```
void rotate(const Shape* ps)    // misuse of RTTI
{
    if (typeid(ps) == Typeid<Circle>()) {
        // do nothing
    }
    else if (typeid(ps) == Typeid<Triangle>()) {
        // rotate triangle
    }
    else if (typeid(ps) == Typeid<Square>()) {
        // rotate square
    }
    // ...
}
```

This style of code is most often best avoided through the use of virtual functions.

Many examples of proper use of RTTI arise where some service code is expressed in terms of one class and a user wants to add functionality through derivation. The `dialog_box` example from §1 is an example of this. If the user is willing and able to modify the definitions of the library classes, say `dialog_box`, then the use of RTTI can be avoided; if not, it is needed. Even if the user is willing to modify the base classes, such modification may have its own problems. For example, it may be necessary to introduce dummy implementations of virtual functions such as `get_string()` in classes for which the virtual functions are not really needed or not particularly meaningful.

For people with a background in languages relying heavily on dynamic type checking, it is tempting to overuse RTTI together with overly general object types. For example:

```
// misuse of run-time type information:

class Object { /* ... */ };

class Container : public Object {
    // ...
public:
    void put(Object*);
    Object* get();
    // ...
};

class Ship : public Object { /* ... */ };

Ship* f(Ship* p1, Container* c)
{
    c->put(p1);
    // ...
    Object* p2 = c->get();
    if (Ship* p3 = (Ship*) p2) // run-time type check
        return p3;
    else {
        // do something else
    }
}
```

Problems of this kind are often better solved by using container templates holding only a single kind of pointer.

```
template<class T>
class Container {
    // ...
public:
    void put(T*);
    T* get();
    // ...
};

Ship* f(Ship* p1, Container<Ship>* c)
{
    c->put(p1);
    // ...
    return c->get();
}
```

Combined with the use of virtual functions, this technique handles most cases.

However, where the type of an object returned from some function cannot be determined at compile time from the types of its arguments, RTTI again becomes a reasonable choice. For example, consider a couple of classes where objects can be compared using information from a common base class only:

```
class X { /* ... */ int key; /* ... */ };
class D1 : public X { /* ... */ };
class D2 : public X { /* ... */ };

X* greater(X* p, X* q) { return (p->key > q->key) ? p : q; }

void f(D1* a, D2* b)
{
    X* res = greater(a,b);
    if (D1* p = (D1*)res) {
        // ...
    }
    else {
        // ...
    }
}
```

Finally, RTTI has an important role in pure optimizations. Consider a function using an abstract `set` class:

```
void fct(set<T>* s)
{
    for (T* p = s->first(); p; p = s->next()) {

        // ordinary set algorithm
    }

    // ...
}
```

This is nice and general, but what if we knew that many of the `sets` passed were implemented by singly linked lists, `slists`, if we knew an algorithm for the loop that was significantly more efficient for lists than for general sets, and if we knew (from measurement) that this loop was a bottleneck for our system? It would then be worth our while to expand my code to handle `slists` separately:

```
void fct(set<T>* s)
{
    if (slist<T>* sl = (slist<T>*)s) { // s is an slist

        for (T* p = sl->first(); p; p = sl->next()) {

            // souped up list algorithm
        }
    }
    else {
        for (T* p = s->first(); p; p = s->next()) {

            // ordinary set algorithm
        }
    }

    // ...
}
```

Naturally, this leads to messier code and makes `fct()` depend directly on the the `slist` class, but that can sometimes be a worthwhile price to pay. In particular, in the case above we not only get the benefit from an improved algorithm but also avoid virtual function calls (on the abstract class `set`) in favor of the inline functions (on the concrete class `slist`). However, one should aim to design systems so as to minimize the use of RTTI.

4 Checked and Unchecked Casts

The introduction of run-time type identification separates objects into two categories: The ones that have run-time type information associated so that their type can be determined (almost) independently of context and those that haven't. Why? We cannot impose the burden of being able to identify an object's type at run-time on built-in types such as `int` and `double` without unacceptable costs in run-time, space, and layout compatibility problems. A similar argument applies to simple class objects and C-style structs. Consequently the first acceptable dividing line is between objects of classes with virtual functions and classes without. The former can easily provide run-time type information, the latter cannot.

Experience shows that this works acceptably, but that it is possible for people to get confused about which classes have virtual functions and thus about what the real meaning of a cast is. This leads to a wish for an explicit way of saying "this class supports RTTI whether it has virtual functions or not."

First we note that there already is a way. Simply define a class with a virtual function and derive from it any class that you desire to be explicit about:

```
class rtti { virtual void __dummy() = 0; };
class X : public rtti { /* ... */ };
```

Unfortunately, this implies a space overhead (especially if it is included in lots of places) and because class `rtti` is so small, making it a virtual base will not provide any significant saving:

```
class X : public virtual rtti { /* ... */ };
```

It is also clear that "`public virtual rtti`" is long enough to be tedious to write after a while so we considered some syntactic sugar:

```
class X : virtual { /* ... */ };
virtual class X { /* ... */ };
class X { virtual; /* ... */ };
```

However, people instantly started imagining a variety of meanings for such notations. In particular, "Oh neat, so `X` is an abstract class!" and "I have always wanted to be able to declare all functions `virtual` in one place" were not uncommon reactions. For now, we don't have an acceptable suggestion for a more explicit way of saying "this class has run-time type information." If you want such information, be sure to have at least one virtual function in the base class you want a checked cast from.

The cast `(D*)p` is legal for any type `D` and for any pointer `p`. If `p`'s static type is "pointer to a class

without virtual functions," the semantics of the cast remain unchanged. This implies that C compatibility is strictly maintained and that no overheads are imposed except for classes with virtual functions. If on the other hand p's static type is "pointer to a class *with* virtual functions" the meaning of `(D*)p` is "check if the type of `*p` is `D` or has `D` as a unique base; if so, return a pointer to the object of class `D` containing the object of class `B` that `p` points to; otherwise, return 0." The implementation requires a combination of compile-time and run-time support work. A plausible implementation will generate code like `(D*)__ptr_cast(p,offset,Typeid<D>(),typeid(p))` where `__ptr_cast()` is some library function that returns a `void*` for a checked cast `(D*)p`.

This will imply the overhead of a double run-time type check for C++ programs that are written using some user-implemented run-time type identification scheme and not converted to the use of the new facilities. For example:

```
void f(dBox* p)
{
    if (p->isKindOf(stringBox)) {      // explicit type check
        stringBox* q = (stringBox)p;   // implicit type check
        // ...
    }
}
```

However, the program will run correctly provided the `isKindOf()` function was correctly implemented. The trivial conversion will yield this shorter and more efficient version:

```
void f(dBox* p)
{
    if (stringBox* q = (stringBox)p) {
        // ...
    }
}
```

In general, old programs using casts to derived classes will yield the same result under the checked cast semantics as they did under the old (unchecked and fast) semantics, providing the object pointed to really was of the type asserted in the cast. This has been the case for all reasonable examples we could think of.

Note that a cast to `void*` can be used to suppress any sort of checking and allow completely arbitrary type conversion:

```
class A { /* some virtual functions declared */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };

void f()
{
    A* pa = new B;
    B* pb = (B*)pa; // same result as ever
    C* pc = (C*)pa; // used to initialize pc with a pointer that didn't
                    // point to a C and would cause wrong results
                    // if used as a C*.

                    // Now it initializes pc with 0; will cause wrong
                    // results if used as a C*.

    pb = (B*)(void*)pb;  // on your head be it!
    pc = (C*)(void*)pa;  // on your head be it!
}
```

Thus a cast of `void*` serves as an operator explicitly suppressing checking and thus relieving us of the burden of introducing a new operator to allow us to explicitly distinguish checked conversions from or unchecked ones.

When many or even most pointer casts thus becomes checked, it becomes practical to detect and warn about unchecked casts so as to allow the programmer to determine if they are in fact dangerous. This must be done with a reasonable amount of taste, though, to avoid fanatical condemnation of all casts. Such a

need always work : TI

condemnation would imply a rejection of C compatibility and also cause real problems for people who actually need unchecked casts. Casting between class pointer types without using `(void*)` to explicitly suppress checking might be considered suspicious. For example

```
class X { /* no virtual functions */ };
class Y { /* ... */ };

void g(X* px)
{
    Y* py1 = (Y*)px; // suspicious cast: warn
    Y* py2 = (Y*)(void*)px;  // on your head be it!
}
```

5 Casting from Virtual Bases

Note that the prohibition of casting from a virtual base class to its derived class can be lifted for classes with virtual functions so that casting from ordinary and from virtual bases are handled identically:

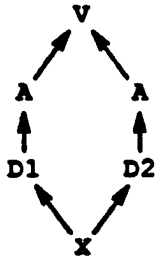
```
class B { /* ... */ virtual void f(); };
class V { /* ... */ virtual void g(); };

class D : public B, public virtual V { /* ... */ };

void g(D& d)
{
    B* pb = &d;
    D* pd1 = (D*)pb; // ok

    V* pv = &d;
    D* pd2 = (D*)pv; // ok (didn't use to work)
}
```

The reason for the prohibition was that there wasn't enough information available about a base class object to do the cast from a virtual base. In particular, an object of a type with layout constraints determined by some other language such as Fortran or C may be used as a virtual base class and for objects of such types only static type information will be available. However, the information needed to provide run time type identification includes the information needed to implement the checked cast.

Naturally, such a cast can only be performed when it is unambiguous. Consider:

```
class A : public virtual V { /* ... */ };
class D1 : public A { /* ... */ };
class D2 : public A { /* ... */ };
class X : public D1, public D2  { /* ... */ };
```

Or graphically:



Here an `X` object has two sub-objects of class `A`. Consequently, a cast from `V` to `A` within an `X` will be ambiguous and return a 0 rather than a pointer to an `A`:

```
void h1(X& x)
{
    V* pv = &X;
    A* pa = (A*)pv;  // pa will be initialized to 0
}
```

This ambiguity is not in general detectable at compile time:

```
void h2(V* pv)
{
    A* pa = (A*)pv;  // pv might point to an X
                     // and then 0 will be returned

                     // or it might point to a ''plain A''
                     // and then a correct pointer to A will be returned
}
```

This kind of run-time ambiguity detection is only needed for virtual bases. For ordinary bases, the proper sub-object to cast to can always be found. Consider the example above rewritten to use ordinary bases:

```
class A : public V { /* ... */ };
class D1 : public A { /* ... */ };
class D2 : public A { /* ... */ };
class X : public D1, public D2  { /* ... */ };
```

Here an X object has two sub-objects of class A each with a sub-object of class V.

```
void h1(X& x)
{
    V* pv = &X;     // compile time error: ambigucus: which A?

    V* pv1 = (V*)(D1*)&x;  // ok: D1's V
    V* pv2 = (V*)(D2*)&x;  // ok: D2's V

    A* pa1 = (A*)pv1; // ok: D1's A
    A* pa2 = (A*)pv2; // ok: D2's A
}
```

Accepting this example makes the definition of checked casting messier, but it only complicates the implementation of the run-time support code by a dozen lines out of three dozen or so. If we disallowed it, we would never see the end of people who also discovered the algorithm and felt cheated by the language not requiring a resolution that was possible.

Note that because checking of casts is triggered by information in a base class and cannot be done without information accessible through the base class, adding information to a derived class is of no help; in particular it is not possible to require casts to a class with a virtual base class to be checked. For example:

```
class V { /* no virtual functions */ };
class A : public virtual V { /* ... */ };

void f()
{
    V* pv1 = new V; // no RTTI, no connection to any A
    V* pv2 = new A;

    A* p1 = (A*)pv1; // (compile time) error: no information to go by
    A* p2 = (A*)pv2; // (compile time) error: no information to go by
}
```

Thus, a cast from a virtual base class to a derived class is legal and checke i or illegal dependent on the definition of the base class (only).

6 Cross Hierarchy Casting

Two related questions must be answered:

- Should casting be constrained to derivation relationships known at compile time?
- Should it be possible to cast from a class to a sibling class in a multiple inheritance hierarchy?

For example:

```
class A { /* ... */ virtual void f(); };
class B { /* ... */ virtual void g(); };
class D : public A, public B { /* ... */ };
class X;

void f(A* pa)
{
    X* px = (X*)pa; // X undefined: legal?
    B* pb = (B*)pa; // B apparently unrelated to A: legal?
}
```

Is the cast `(X*)pa` legal, and if so what is its meaning? If it is legal two interpretations are possible:

[1] `px` is initialized with `pa`'s bit pattern.

[2] The object pointed to by `pa` is examined and a checked cast is performed, that is, if `*pa` really has a unique sub-object of class `X` then `px` is made to point to it; otherwise `px` will be initialized to `0`.

A similar choice exists for the meaning of the cast `(B*)pa`.

In both cases the suggested resolution is to do the checked cast. In the case of a cast to an undefined class this decision ensures that the same result is obtained independently of whether the class declaration has been seen or not.

Consider the following set of classes:

```
class employee { /* ... */ };
class manager : public employee { /* ... */ };
class analyst : public employee { /* ... */ };

class engineer { /* ... */ };
class electrical_engineer : public engineer { /* ... */ };
class mechanical_engineer : public engineer { /* ... */ };
```

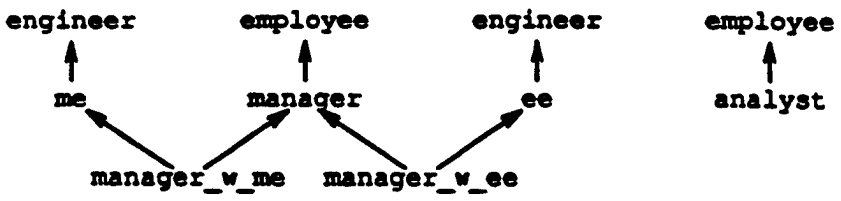
If we want to ask questions like:

- Is this engineer a manager ?
- Does this employee have an EE degree ?
- How many analysts have an engineering degree ?

and we want to use language features rather than algorithms, then we will define:

```
class manager_with_ee
    : public manager, public electrical_engineer
{ /* ... */ };

class manager_with_me
    : public manager, public mechanical_engineer
{ /* ... */ };
```

Or graphically:



We can then use checked casts like this:

```
my_fct(engineer* pe1, employee* pe2)
{
    if (manager* m = (manager*)pe1) {
        // this engineer is a manager
    }
    // ...
    if (electrical_engineer* ee = (electrical_engineer*)pe2) {
        // this employee has an EE degree
    }
    // ...
}
```

The decision to allow cross-hierarchy casting also matches the rule that a virtual function can be defined on one branch of a multiple inheritance hierarchy and called through another.

One way of looking at this checked cast proposal is as a cleanup of the concept of explicit type conversion to make it less error-prone. Given the – generally very reasonable – demands of C compatibility, not all casts can be checked. It might, however, be worth considering a restriction that would close another well known loophole. Casting an object without virtual functions to an undefined class cannot be checked and is thus inherently error-prone. As the casting of objects with virtual functions becomes better behaved such completely unconstrained casts become an anomaly and the programmers that have become used to the checked cast are more likely to be surprised by the old semantics. Thus it is suggested that

```
class X;

X* g(ClassWithoutVirtuals* p)
{
    return (X*)p;
}
```

should become illegal; that is, it should be a compile time error.

7 References

The discussion thus far has focussed on pointers. However, a reference can also refer to objects of a variety of base and derived classes and is subject to casting in a way very similar to pointers. For example the `set` example from §3 could be written using references instead of pointers. However, simply rewriting the critical cast

```
if (slist<T>* sl = (slist<T>*)s)
```

to

```
if (slist<T>& sl = (slist<T>&)s)
```

wouldn't make sense in general because there is no "zero reference" to test. Consequently, a reference cast throws an exception if the cast cannot be performed. The example thus becomes:

```
void my(set<T>& s)
{
    try {
        slist<T>& sl = (slist<T>&)s; // s is an slist

        for (T* p = sl.first(); p; p = sl.next()) {

            // souped up list algorithm
        }
    }
```

```
        catch(BadCast) {
            for (T* p = s.first(); p; p = s.next()) {
                // ordinary set algorithm
            }
        }
        // ...
    }
```

The difference reflects a fundamental difference between reference and pointers. A pointer may or may not point to an object, whereas a reference can be assumed to refer to one. As ever, the possibility of zero pointers makes explicit tests necessary where pointers are used.

One might argue that because explicit tests against `0` can be – and therefore occationally will be – accidentally omitted, a checked pointer cast that fails should throw an exception just like a failed checked reference cast. However, this would only handle one minor source of `0` pointers and not all of those lead to errors.

8 How Much Information?

The basic notion of the RTTI mechanisms described here is that users need only know as much about them as they need to and that for maximal ease of programming and implementation independence we should aim at using as little as possible:

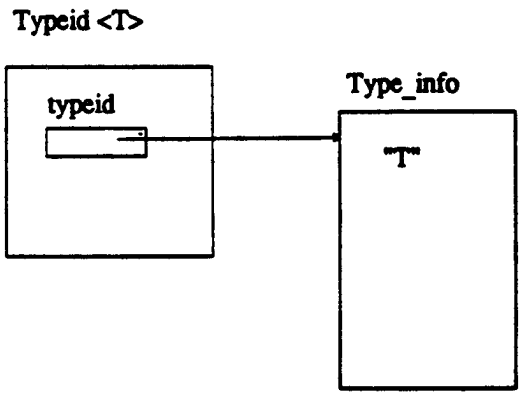
[1] Preferably, we should use no run-time type information at all and rely exclusively on static (compile time) checking).

[2] If that is not possible, we should use only checked casts. In that case, we don't even have to know the exact name of the object's type and don't need to include any header files related to RTTI.

[3] If we must, we can compare `typeids` and `Typeids`, but to do that we have to explicitly use `typeid()`, and typically also know the exact name of at least some of the types involved. It is assumed that "ordinary users" will never need to examine run-tim type information further.

[4] Finally, if we absolutely do need more information about a type, say because we are trying to implement a debugger, a data base system, or some other form of object I/O system, we can use operations on `typeids` to obtain more detailed information.

The subsections below explains the details of this. The relationship between the objects involved can be illustrated like this:



Both `typeid` and `Typeid<T>` are handles to the `Type_info` object that really holds the type information such as the name string.

Class typeid

Above, we presented the `typeid` operator only briefly before making its use implicit in the checked cast mechanism. However, `typeid` can be used explicitly to gain access to information about types at run time. It can be applied to a pointer (only). For example:

```
void f(X* p, Y& r, int i)
{
    typeid(p);
    typeid(0);      // error

    typeid(&r);
    typeid(r);      // error

    typeid(&i);
    typeid(  ;      // error
    typeid(  ;      // error
}
```

Actually, `typeid()` is a constructor for class `typeid`:

```
class Type_info;

class typeid {
private:
    // ...
public:
    typeid(const void*); // this constructor is special

    int operator==(typeid) const;
    int operator!=(typeid) const;

    const char* name() const; // return string representation of type name
    const Type_info* info() const; // return type descriptor object

    // copy and address-of predefined as usual
};
```

That is, a `typeid` can be used for comparisons, for getting more information about the type, and for getting a string representation of the name of the type. A `typeid` is assumed to be a small object that can be cheaply copied. Typically, its representation will simply be a pointer to an object of type `Type_info`, the one returned by `info()`, that really holds information about the type. In particular, information `Type_info` is used to implement checked casts.

The `typeid` constructor takes a `const void*` argument to allow type checking to be done using ordinary rules. However, its implementation is special in that it performs a service that cannot be described in the language because it depends on information that is directly available only to a compiler. Given a pointer to an object of a type without virtual functions it returns a `typeid` determined by the name of the type. Given a pointer to an object of a type with virtual functions it returns a `typeid` somehow found by examining the object. This is further described in §10.

Allowing the `typeid` constructor to take reference arguments was considered but rejected as inessential and requiring changes to the type checking rules.

The reason for using a `typeid` class and not simply a pointer type is that it is not clear that every implementation will be able to guarantee uniqueness of type identification objects. In particular, it is not obvious that every dynamic loading and linking mechanism will be able to avoid occasional duplication of such objects. With a `typeid` class there is no problem defining `==` to cope with such duplication. It is also an advantage of the class `typeid` approach that meaningless operations such as `++` are not supplied by the language as defaults.

Template Class `Typeid`

The `typeid` class is a perfectly ordinary class as far as syntax analysis and type checking is concerned, only in the implementation of its constructor is special help from the compiler needed. This was considered important enough to reject the idea of allowing the `typeid` constructor to take a type name as an argument; that is to allow the test from §1 to be written like this:

```
if (typeid(bp) == typeid(dbox_w_string)) {
```

rather than the initially somewhat odd looking

```
if (typeid(bp) == Typeid<dbox_w_string>()) {
```

To provide type identification objects for named types we introduce a template class:

```
template<class T> class Typeid : public typeid {
    // typeid needs to be a base to allow comparisons, etc.
public:
    Typeid(); // this constructor is special

    // copy and address-of predefined as usual
};
```

Just like `typeid` is a perfectly ordinary class except for the implementation of its default constructor requiring compiler help, `Typeid` is a perfectly ordinary template class except for the implementation of its default constructor requiring compiler help. This minimizes the impact on compilers and ensures that the type identification mechanism fits into the language semantically and syntactically. Once you get used to templates the

```
if (typeid(bp) == Typeid<dbox_w_string>()) {
```

syntax becomes normal and natural. In fact, it is the `sizeof` operator that begins to look distinctly odd because it sometimes takes an object and sometimes a type name as an argument†.

Ideally, `Typeid` and `typeid` would have a single name, but because C++ doesn't allow the overloading of a template name two distinct names are needed. It is a cause for worry that `Typeid` and `typeid` differ only slightly and undoubtedly someone will become confused. However, we feel that the confusion would be greater if the identifiers differed more significantly.

In addition to providing "constants" describing types for which we know the name, the `Typeid` template allows us to provide operations that can only be expressed in terms of the name of a type. We don't propose any such functions just now, but prefer to wait for more experience to be gathered. However, we will discuss the most obvious such operation, `new_object()`, to illustrate the possibilities and point to some design choices. Assume we defined

```
template<class T> class Typeid : public typeid {
    // ...
    T* new_object(); // return pointer to default initialized
                     // object allocated using new
};
```

Having `new_object()` as a member allows a function to call another with an argument that specifies the type of some objects to be created. For example:

```
void tree_constructor(Typeid<Node> node_type)
{
    // ...
    Node* np = node_type.new_object();
    // ...
}
```

but what if the node type we call `tree_constructor` with doesn't have a default constructor? To

† Could C++ be re-designed from scratch, we would most likely abandon the `sizeof` operator in favor of a `sizeof` member function on `typeid` and `Typeid`.

define `Typeid<T>::node_type.new_object()` we must decide on a strategy for dealing with run-time errors. We would also need another constructor for `Typeid` to allow `tree_constructor()` to be called with `Typeids` for classes derived from `Node`. For example:

```
// call with Mynode derived from Node:
tree_constructor(Typeid<MyNode>())
```

on the other hand we would have to reject

```
// try to fool type system:
tree_constructor(Typeid<int>())
```

To handle this we would need:

```
template<class T> class Typeid : public typeid {
    // ...
    Typeid(typeid t); // construct Typeid<T> from t
                      // check at run time that t describes a T

    T* new_object(); // return pointer to default initialized
                     // object allocated using new
};
```

Again, it is not hard to check (at run time) whether an argument `t` is of the required type `T`. The hard part is to decide what error handling strategy to use. In essence, `Typeid(typeid)` would be the checked cast for type identifiers.

Detailed Type Information

Consider for the moment how an implementation or a tool could make information about types available to users at run-time. Say we have a tool that generates a table of *(member_name,offset,typeid)* entries for each type. The preferred way of presenting this to the user is to provide an associative array (map, dictionary) of type names and such tables. To get such a member table for a type a user would write:

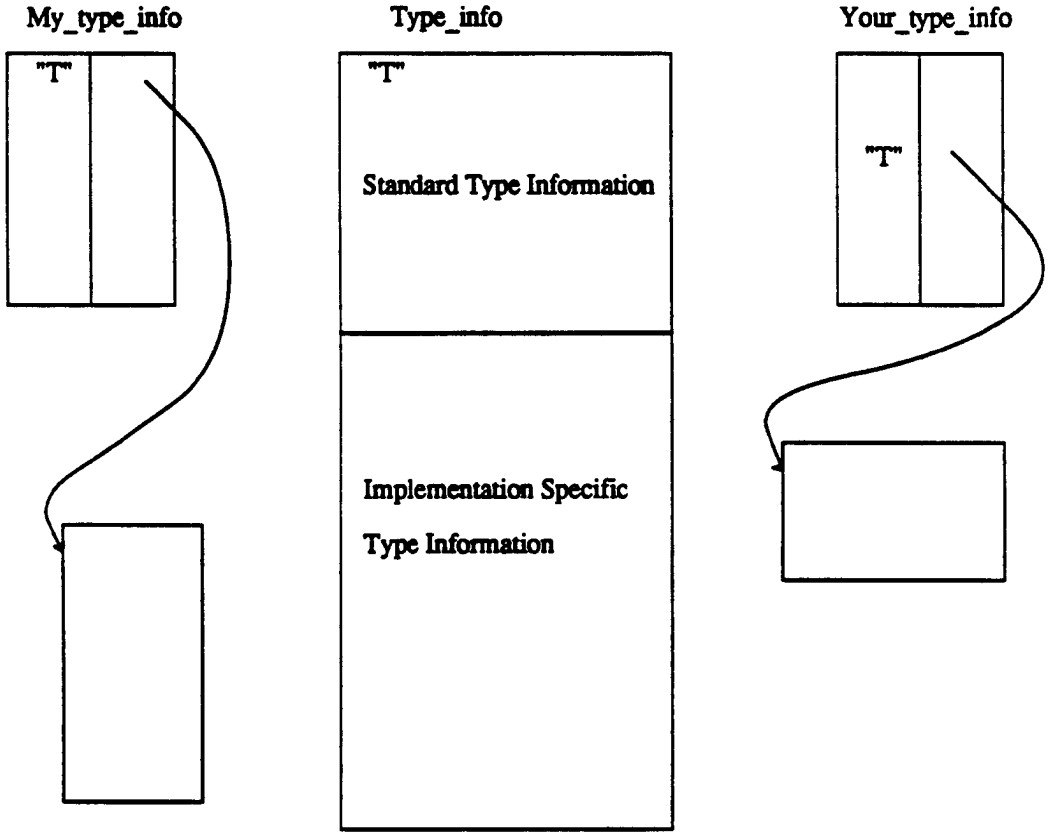
```
void f(B* p)
{
    My_member_info* tbl = my_type_table[typeid(p).name()];
    // use tbl
}
```

where `My_member_info` is the name of the type of our information and `my_type_table` is the name of the associative array (Map, Dictionary) in which we keep the *(typename,table)* pairs. If we wanted to, we could index the tables directly with `typeids` rather than requiring the user to use the `name()` string:

```
My_member_info* tbl = my_type_table[typeid(p)];
```

It is important to note that this way of associating `typeids` with information allows several people or tools to associate different information to types without interfering with each other. This is most important because the likelihood that someone can come up with a set of information that satisfies all users is zero. In particular, any set of information that would satisfy most uses would be so large that it would be unacceptable overhead for users that need only minimal run-time type information.

Using these techniques, we might have several independent sets of information about types in a program:



The function `typeid::info()` is logically redundant in that *all* information beyond the identity of the type could be obtained through the association technique described above. It is assumed, though, that C++ implementations will contain mechanisms for providing run-time type information beyond the minimum and that a class `Type_info` will be defined in the standard library to describe such objects. The `typeid::info()` function is intended to provide a standard way of gaining access to such information. The alternative would be to have a definition for a standard association class.

Actually, even the function `typeid::name()` is logically redundant in that the name string could be obtained through the association technique described above. However, that wouldn't allow association tables to be sorted according to the spelling of type names and would make it less easy for programmers to provide string representations of type names. We would prefer it to be trivially easy to print the name of a class. For example:

```
template<class T>
class Vector {
    // ...
    void my_name1() { cout << "Vector<" << Typeid<T>().name() << '>'; }
    void my_name2() { cout << Typeid< Vector<T> >().name(); }
    void my_name3() { cout << Typeid<Vector>().name(); }
};
```

where all functions happen to be equivalent.

The definition of `typeid` and `Typeid` are found in `<typeid.h>` and the definition of class `Type_info` in `<Type_info.h>`. Note that `<typeid.h>` must be included if any use is made of the `typeid` operator because even the `typeid` comparison operators are defined there.

9 Types without RTTI

Unchecked casts and the absence of run-time determined information for the `typeid` operator to return can be seen as an optimization that is made necessary by the requirements of run-time and space efficiency and of layout compatibility. However, what does happen if we apply `typeid` to a pointer to an object of a type that does not have a virtual function? There appears to be two practical design alternatives:

[1] Give a (compile-time) error

[2] Return type identification for the static type of the object

We chose the second. The reason was to allow code that operated on every kind of object using run-time type information. For objects of types without virtual functions the information (and thus the program) is only correct provided the pointer or reference used to access it hasn't suffered conversions that lost type information. This is exactly the same situation as with every other use of an object. It will therefore be possible to write programs that extract run-time type information from every object and use it some uniform way.

Because `typeid` takes a pointer it is sometimes necessary to use the address-of operator to get type information for an object. For example;

```
void f(int i, int& r)
{
    typeid id = typeid(i); // error: 'i' not a pointer
    id = typeid(&i);       // fine: '&i' is a pointer

    id = typeid(r);        // error: 'r' not a pointer
    id = typeid(&r);       // fine: '&r' is a pointer
}
```

The `typeid` of a pointer with the value `0` is a `typeid` that compare equal to `typeid(0)` and different to every `typeid` that is not the result of using `typeid()` on an expression with a value `0`.

The `typeid` of an array, a pointer or reference to a pointer is, a pointer or reference to a pointer is currently undefined, but will be presented later together with details of class `Type_info`.

10 Implementation Issues

Consider how to implement RTTI. Because the mechanism has been designed not to add new syntax or to affect the type checking rules compiler issues are reduced to implementing the semantics of the `typeid()` and `Typeid()` constructors and the checked cast.

To deal with run-time aspects of the mechanism three separate issues must be addressed:

[1] How do we get hold of run-time type information given a pointer or a reference?

[2] How do we use the run-time type information to implement the `typeid()` constructor and checked casts?

[3] How do we generate the run-time type information?

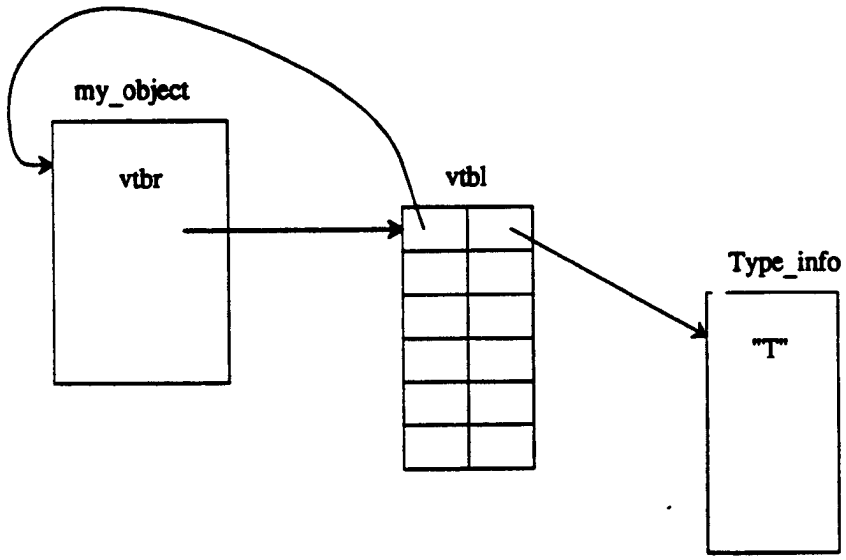
The implementation described here is only one of several possible. It assumes a traditional and fairly straightforward implementation of C++ along the lines described in [1]. That is, each object of a class with virtual functions contains a pointer (`vptr`) to a table of virtual functions (`vtbl`).

The basic idea is to place a pointer to an object describing an object's type in the `vtbl`. Such description objects will be of some type derived from class `Type_info` and a pointer to a `Type_info` will be the representation of an object of class `typeid`.

This basic scheme allows for the `typeid()` constructor to be implemented. Basically `typeid()` is nothing but a test to protect against zero-valued pointers followed by a double indirection to retrieve the pointer to the `Type_info` object.

A call of the `Typeid()` constructor `Typeid<MyType>()` degenerates into the name of `MyType`'s run-time type identification object.

Here is a plausible memory layout for an object with virtual function table and type information object:



For each type with virtual functions an object of type `Type_info` is generated. These objects need not be unique. However, a good implementation will generate unique `Type_info` objects wherever possible and only generate `Type_info` objects for types where some form of run-time type information is actually used. A easy implementation simply places the `Type_info` object for a class right next to its `vtbl`.

Checked Casts

In most cases the implementation of a cast `(D*)px` where the static type of `*px` is `X` is straightforward: Retrieve a pointer to the run-time type identification object from `*px`, Generate a pointer to the run-time type identification object for `D`, and have a library routine see if `*px`'s class is `D` or a base of `D` and return a – possibly slightly adjusted – pointer. The adjustment is needed when `X` class isn't a first base of `D` class. For example:

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

void f()
{
    X* px = new D;   // px doesn't point to the start of the D object
    D* pd = (D*)px;  // pd should point to the start of the D object
}
```

This adjustment is trivially implemented.

However, cases where a base class `X` appears more than once in a class hierarchy needs more care. Consider first ordinary (non-virtual) base classes:

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

```
void f(D* pd)
{
    X* px1 = (D1*)pd;
    X* px2 = (D1*)pd;

    pd = (D*)px1; // pd should point to the start of the D object
    pd = (D*)px2; // pd should point to the start of the D object

    D1* pd1 = (D1*)px1;
    pd1 = (D1*)px2;
}
```

Clearly the adjustments needed for the two `(D*)` casts are different. Similarly, the adjustments needed for the two `(D1*)` casts are different. Consequently, we need to store (in the `vtbl` or equivalent) the offset of the sub-object in the overall object. Given that, we can not only perform the correct adjustment of pointers but also resolve the case of multiple sub-objects of types `V` and `A` mentioned in §5.

This also has the beneficial effect of correcting an error in the way ordinary casts used to work. Under the old, unchecked, semantics `pd1=(D1*)px2` simply assumes that `px2` pointed to a `B` sub-object of `D1` and (wrongly) adjusts the pointer accordingly.

11 Alternatives

The current proposal is a result of a series of ideas and experiments with both the syntax and semantics of run-time type identification. Here, we would like to explain some of the alternatives we considered. The ideals we looked for were the usual: Ease of learning, ease of reading, direct representation of the underlying semantics, no pointless redundancy, minimal syntactic innovation, minimal compatibility problems (including a minimal number of new keywords), ease of implementation, reasonable run-time and space efficiency, etc.

A key line of thought was to try to define a notation for run-time type identification that did not involve anything a user couldn't define in C++ itself; that is, trying to guarantee that the new mechanisms would fit smoothly into the language by actually defining it in the language and then relying on compilers and other tools for optimization.

We felt that the `typeid()` constructor was more appropriate than a "magic" member function that could be applied to all objects. For example:

```
void f(X* p, Y& r, int i, char*a[])
{
    p->typeid();
    r.typeid();
    i.typeid();
    a.typeid();
    X::typeid();
    int::typeid();
    char*::typeid();
}
```

Once all possibilities (incl. the built-in types) had been taken into account the "magic" member function solutions looked messy.

We considered defining `<`, `<=`, etc. on `Typeid` objects to express relationships in a class hierarchy. That is easy, but in addition to being too cute it also suffers from the problems with an explicit type comparison operation as described in §1. Because C++ relies on static type checking, we need a cast in all cases so we can just as well make that the test.

There are many ways of using run-time type information in a language and a diverse set of facilities has been used in a variety of languages. We considered a couple of alternatives with implications beyond run-time type identification. Given, RTTI one can support "unconstrained methods;" that is, one could hold enough information in the RTTI for a class to check at run time whether a given function was supported or not. Thus one could support Smalltalk-style, exclusively dynamically-checked functions. However, we felt no need for that and considered that extension as going contrary to our effort to encourage efficient and type-safe programming. In other words, that extension would take C++ in a new direction contrary to its

direction so far. The checked cast enables a check-and-call strategy:

```
if (D* pd = (D*)pb) { // is *pb a D?
    pd->dfct(); // call D function
    // ...
}
```

rather than the call-and-have-the-call-check strategy of Smalltalk:

```
pb->dfct(); // hope pd has a dfct
```

The check-and-call strategy provides more static checking (we know at compile time that `dfct` is defined for class `D`), doesn't impose an overhead on the vast majority of calls that don't need the check, and provides a highly visible clue that something beyond the ordinary isgoing on.

A more promising use of RTTI would be to support "multi-methods," that is the ability to select a virtual function based on more than one object. This would be a boon to writers of code that deals with binary operations on diverse objects. Generalized addition, geometric intersect operations, and other reasonably common operations belong to this class of problem. We make no such proposal, however, because we cannot clearly grasp the implications of such a change and do not want to propose a major new extension without experience.

12 Survey of Issues

There are several issues and proposals wrapped up into the RTTI mechanism. They can and should be considered individually but we feel that the final evaluation of any run-time type identification scheme should be based on the utility and elegance of a complete set of features. The individual aspects of the proposal here are:

[1] The checked cast notion (as opposed to an explicit checked cast operator or some alternative notion such as an `isKindOf` operator).

[2] The use of virtual functions to distinguish types that support run-time type identification from other types (as opposed to supporting RTTI for all types or to support RTTI for types explicitly declared to support it).

[3] The syntax extension allowing declarations in conditions.

[4] The notion of cross hierarchy casting (as opposed to allowing casts only within known class hierarchies).

[5] The notion of casting to a reference (as opposed to disallowing reference casts and thus avoiding the use of exceptions).

[6] The introduction of a `typeid` class with a "magic" constructor (as opposed to either no way of getting access to objects describing a type or some special operator for gaining access).

[7] The introduction of a `typeid` class (as opposed to using a pointer to a `Type_info` object).

[8] The introduction of a `Typeid` class template derived from `typeid` to provide operations that can only be described using actual type names (as opposed to introducing special syntax and special operators for that).

[9] The notion that objects of types that do not support RTTI are acceptable to the cast and `typeid` operators yielding values that depend on their static type (as opposed to causing compile time errors or supplying RTTI for every object).

[10] Support for casting to a non-unique sub-object from within that sub-object (as opposed to simply defining casting as conversion from the run-time determined class of the object to the desired type); §5.

[11] The notion of a `Type_info` class defined in a standard library (as opposed to supporting checked casts and type identity only).

There are of course many additional details, such as the exact name of the `BadCast` exception and whether class `typeid` and class `Type_info` ought to be declared in the same header file, but we feel that any RTTI facility designed along the lines we suggest will be characterized by the choices outlined here.

13 How to Manage until RTTI comes

This proposal for RTTI is most unlikely to be available on your C++ implementation any day soon. What then – if you need to – can you do to get the benefits some variant RTTI until becomes generally available? If you use one of the major libraries, you already have some mechanism available and even if you don't you can build your own using the technique described in [9]. The real problem is how to stay compatible with others and to make sure that you can convert the the "real" RTTI system once it becomes available.

We suggest you write your code in terms of five macros

```
typeid static_type_info(type)    // get typeid for type
typeid ptr_type_info(pointer)    // get typeid for pointer
typeid ref_type_info(reference) // get typeid for reference
pointer ptr_cast(type,pointer)      // convert pointer to type*
reference ref_cast(type,reference) // convert reference to type&
```

We believe that these can be defined for any reasonable RTTI mechanism so that your user code becomes independent of its particulars. That makes portability manageable and once your C++ implementation provides a standard RTTI mechanism you can either redefine your macros or (preferably) rewrite the code to use it directly.

14 Acknowledgements

Brian Kernighan, Andrew Koenig, Doug McIlroy, Rob Murray, and Jonathan Shopiro provided valuable insights that helped shape this proposal. Tom Penello checked that allowing declarations in conditions would not introduce any new syntax ambiguities. Doug McIlroy was the one that caused our thinking to shift from an explicit (and named) checked cast operator to simply checking ordinary casts wherever possible. 'Michey Mehta and Shankar Unni provided many ideas of different approaches to run-time type identification and its implementation that helped better understand problems and solutions presented in this proposal.

15 References

[1] Margaret A. Ellis, Bjarne Stroustrup: *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[2] Mary Fontana, Martin Neath: *Checked Out And Long Overdue: Experience in the Design of a C++ Class Library.* USENIX C++ Conference Proceedings, April, 1991.

[3] Keith E. Gorlen: *An Object-Oriented Class Library for C++ Programs.* Proceedings of the USENIX C++ Workshop, 1987.

[4] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico: *Data Abstraction and Object-Oriented Programming in C++. Wiley, 1990.*

[5] John A. Interrante, Mark A. Linton: *Runtime Access to Type Information in C++.* USENIX C++ Conference Proceedings, 1990.

[6] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++.* USENIX C++ Conference Proceedings, 1990.

[7] Mark A. Linton, John M. Vlissides, and Paul R. Calder: *Composing user interfaces with InterViews.* Computer, 22(2):8-22, February 1989.

[8] Dmitry Lenkov, Michey Mehta, Shankar Unni: *Type Identification in C++.* USENIX C++ Conference Proceedings, April, 1991.

[9] Bjarne Stroustrup: *The C++ Programming Language (Second Edition).* Addison-Wesley, 1991.

[10] Andre Weinand, Erich Gamma, and Rudolf Marty: *ET++ - An Object-Oriented Application Framework in C++.* ACM OOPSLA'88 Conference Proceedings, 1988.