x3j16/92-0121, WG21/N0198

Run-Time Type Identification for C++ (Revised yet again)

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 run-time 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. The basic parts of the proposal are a run-time checked type conversion operator `dynamic_cast` and an operator `typeid` that returns objects of class `Type_info` providing a run-time representation of types. Experimental implementations exist.
This paper is a version of the paper presented to the USENIX C++ conference in Portland Oregon in August 1992. The difference is that this version reflects a proposal to clean up the C++ conversion operators [13]. Appendix G is the proposed manual changes.

1 Introduction

Consider:

```
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 `dialog_box` and `dbox_w_str` 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 [3,4,6, and 14] and detailed explanations of how to implement them can be found in [1,5,9,10]. However, these mechanisms are mutually incompatible so that they become a barrier to the use of more than one library. Also, all require a considerable amount of foresight on the part of a base class designer. What is proposed here is a language supported mechanism.

† Dmitry wasn't involved in this latest revision so he is innocent of any errors introduced into and any confusion caused by this version.

A naive solution would be:

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

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

        // 'plain' dialog box
    }
}
```

Given the name of a type as the operand, `typeid()` operator returns an object that identifies it. Given an expression operand, `typeid()` returns an object that identifies the type of the object that the expression denotes. In particular, `typeid(*bp)` returns an object that allows the programmer to ask 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 the simplest question to ask, but it is typically *not* the right question. The reason to ask 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. Typically, we are not interested in the *exact* type of the object pointed to, but only in whether we can perform that cast. This question can be asked directly:

```
void my_fct(dialog_box* bp)
{
    dbox_w_str* dbp = dynamic_cast<dbox_w_str*>(bp);  // checked cast

    if (dbp) {

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

        // 'plain' dialog box
    }
}
```

The `dynamic_cast<T*>(p)` operator converts its operand `p` to the desired type `T*` if `*p` really is a `T`; otherwise, the value of `dynamic_cast<T*>(p)` is `0`.

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

Naturally, an implementation of the dynamic 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 operation:

- By using the information available in the type information objects it is often possible to cast from a virtual base class to a derived class; see Appendix B.
- By using the information available in the type information objects it is possible to cast to types that are not fully defined in the scope of the cast; see §8.
- A dynamic cast 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)) { // check, then cast
        dbox_w_str* dbp = static_cast<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)) { // check, then cast
        dbox_w_str* dbp = static_cast<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 = dynamic_cast<dbox_w_str*>(bp);
```

However, removing that redundancy would leave the programmer without a clearly visible clue that something "interesting" is going on. 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(dynamic_cast<dbox_w_str*>(bp)); // ok: run-time checked cast
}
```

The `dynamic_cast<dbox_w_str*>(bp)` notation has the advantage over the traditional notation that is is easy to spot in a program – both for a human and for a simple search tool (for example, grep).

As a final simplification we might adopt the 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 = dynamic_cast<dbox_w_str*>(bp)) {

        // use 'dbp'
    }

    // ...
}
```

The value of a declaration is the value of the declared variable after initialization. To avoid ambiguities, we do not suggest that declarations should be allowed in any new places in the grammar except as conditions. See Appendix A for further details.

In §8 we will return to the `typeid()` operator and examine what it and the objects it returns are good for.

2 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:

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

void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
        // do nothing
    }
    else if (typeid(r) == typeid(Triangle)) {
        // rotate triangle
    }
    else if (typeid(r) == typeid(Square)) {
        // rotate square
    }
    // ...
}
```

This style of code is usually best avoided through the use of virtual functions. It was the first author's experience with Simula code written this way that caused facilities for run-time type identification to be left out of C++ in the first place.

For many people trained in languages such as C, Pascal, Modula, Ada, etc. there is an almost irresistible urge to organize software as a set of switch statements. This urge should most often be resisted. Please note that even though we are proposing a RTTI mechanism for C++ we do not propose to support it with a type-switch statement (such as Simula's INSPECT statement, for example).

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 needed or not meaningful.

For people with a background in languages that rely heavily on dynamic type checking, such as Smalltalk, it is tempting to RTTI and overly general 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 = dynamic_cast<Ship*>(p2)) // run-time type check
        return p3;
    else {
        // do something else
    }
}
```

Here, class `Object` is an unnecessary implementation artifact. 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.

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

```
class X {
        // ...
public:
        X* greater(X* arg); // return greater of *this and *arg
        // ...
};

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

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

Note that there is no requirement that objects should only be compared to objects of their own type, and that the type of the returned object cannot be determined from the types of the operands only. Had either of those conditions been true, superior solutions could have be achieved without using RTTI. The recent relaxation of the virtual function overriding rules also provides an alternative to RTTI in some cases; see

Appendix E.

Finally, RTTI has an important role in 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 our code to handle `slists` separately:

```
void fct(set<T>* s)
{
    if (slist_set<T>* ss = dynamic_cast<slist_set<T>*>(s)) {

        slist<T>* sl = ss->get_list(); // sl is s's 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 `slist` algorithm but also avoid virtual function calls (on the abstract class `set`) in favor of inline functions (on the concrete class `slist`). Combined, these two optimizations can amount to one or two orders of magnitude. Please note that the "optimized" example is still as general as the original. It handles every argument properly (as opposed to the buggy `Shape` example above). All that has been done is to insert code dealing with an important special case. Should the representations used for `set<T>`s change, the grimy optimization code will simply become redundant; it will not become a source of bugs caused by false assumptions.

Note that the example assumes that the `slist` is a (hidden) part of the `slist_set`'s representation so that we can't get access to it except through the `get_list()` member function provided for that purpose. This example therefore differs from the one used in the USENIX paper [12] by respecting abstraction boundaries.

Evidence from library design and use suggests that almost everybody needs RTTI occasionally, but that one should aim to design systems so as to minimize its use. Where applicable, static type checking provides stronger guarantees, smaller and and faster code, and cleaner designs. Therefore RTTI should only be used where it is clearly needed. One should be suspicious of "arguments" of the form "RTTI is clearly needed in this case." In our experience, such arguments are often wrong and hide a lack of understanding of the problem area or of the design choices available in C++.

3 Dynamic and Static 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, from an implementation point of view, 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.

Further, a class with virtual functions is often called a polymorphic class and polymorphic classes are the only ones that can be safely manipulated through a base class†. It thus, from a programming point of view, seems natural to provide run-time type identification for polymorphic types (only): They are exactly the ones for which C++ supports manipulation through a base class. Supporting RTTI for a non-polymorphic type would simply provide support for switch-on-type-field programming. Naturally the language should not make that style impossible, but we see no need to complicate the language solely to accommodate it.

Experience shows that providing RTTI for polymorphic types (only) works acceptably. However, people can get confused about which objects are polymorphic and thus about whether a dynamic cast can be used. This is discussed further in Appendix C.

Applying `dynamic_cast<T*>()` to a pointer `p` of a non-polymorphic type is a compile time error unless the cast is known to be safe by the compiler. Given dynamic casts, a static (that is, not run-time checked) cast from a polymorphic type could be considered suspicious and we expect that good compilers will optionally issue warnings for such casts. For example:

```
class X {
        // no virtual functions
};

class B {
        virtual int f();
        // ...
};

void f(X* px, B* pb)
{
        Y* p = dynamic_cast<Y*>(px);  // error: X is not polymorphic
        D* q = static_cast<D*>(pb);   // optional warning: B is polymorphic,
                                      // you could have used dynamic_cast
}
```

Note that dynamic and static casts are fundamentally different in that an static cast is based (almost) exclusively on type information whereas a dynamic cast is based (almost) exclusively on the value of the object.

A dynamic cast of pointer with the value `0` yields `0` because `0` does not point to an object of a polymorphic type. For example:

```
X* p = 0;

Y* q1 = dynamic_cast<Y*>(p);  // q1 = 0
Y* q2 = dynamic_cast<Y*>(0);  // compile time error
```

The relationship between dynamic and static casts is discussed further in §8. Dynamic casts of references are considered in §5. Syntax issues are discussed in §8.

---

† Here "safely" means that the language provides guarantees that objects are used only according to their defined type. Naturally, individual programmers can in specific cases demonstrate that manipulations of a non-polymorphic don't violate the type system.

4 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 = dynamic_cast<X*>(pa); // X undefined: legal?
    B* pb = dynamic_cast<B*>(pa); // B apparently unrelated to A: legal?
}
```

In both cases a run-time check is possible and useful, thus both cases are legal.

In the case of a dynamic 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. This is not the case for ordinary casts; see §8. Note that a dynamic cast requires its operand to be of a known and polymorphic type.

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 { /* ... */ };
```

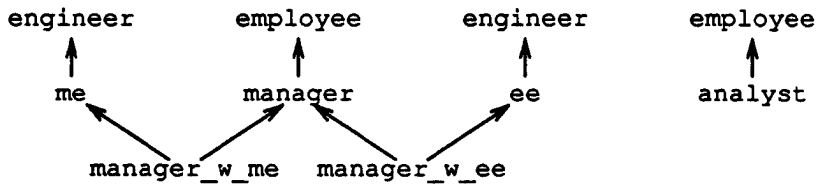
If we want to ask questions like:

- 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 based on data stored by the programmer, then we 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 dynamic casts like this:

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

Note that we can do this even where the connection between `employee` and `electrical_engineer` is unknown because no class derived from both, such as `manager_with_ee`, has yet been defined. In general, it is not possible to know that two classes are unrelated because there is always the possibility that a class defined in some other compilation unit is derived from both. However, given an object of polymorphic type we can always (at run time) determine if the classes are related for that object.

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.

5 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 §2 could be written using references instead of pointers. However, we cannot simply rewrite the critical test

```
if (slist_set<T>* ss = dynamic_cast<slist_set<T>*>(s))
```

to

```
if (slist_set<T>& ss = dynamic_cast<slist_set<T>&>(s))
```

That 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_set<T>& ss = dynamic_cast<slist_set<T>&>(s);

        slist* sl = ss.get_list();

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

            // souped up list algorithm
        }
    }
    catch(Bad_cast) {
        for (T* p = s.first(); p; p = s.next()) {

            // ordinary set algorithm
        }
    }

    // ...
}
```

This is a very poor example of a reference cast because it uses an exception to handle ordinary local control flow rather than an error. In this case, a pointer cast would have been more appropriate:

```
if (slist_list<T>* ss = dynamic_cast<slist_set<T>*>(&s) {
        // ...
}
```

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

Explicit tests against `0` can be – and therefore occasionally will be – accidentally omitted. One might argue that a dynamic pointer cast that fails should throw an exception just like a failed dynamic reference cast. However, this would only handle one minor source of `0` pointers and not all `0` pointers lead to errors. The programmer has a choice, though:

```
void f(dialog_box* p)
{
    dbox_w_string* p1 = dynamic_cast<dbox_w_string*>(p);    // p or 0
    dbox_w_string* p2 = dynamic_cast<dbox_w_string*&>(p);   // p or exception
    dbox_w_string* p3 = &dynamic_cast<dbox_w_string&>(*p); // p or exception
    // ...
}
```

A dynamic reference cast can be a good way of testing an assumption. In contrast, the dynamic pointer cast allows (and requires) a test to select between two reasonable alternatives.

6 How Much Information?

The basic notion of the RTTI mechanisms described here is that for maximal ease of programming and implementation independence we should minimizing the use of RTTI:

[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 dynamic 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`, but to do that we need to know the exact name of at least some of the types involved. It is assumed that "ordinary users" will never need to examine run-time 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 [1] – we can use operations on `typeids` to obtain more detailed information.

This approach of providing a series of facilities of increasing involvement with run-time properties of classes contrasts to the approach of providing a class giving a single standard view of the run-time type properties of classes. We feel that the proposed approach encourages greater reliance of the (more safer and efficient) static type system, has a smaller minimal cost (in time and comprehensibility) to users, and is also more general because of the possibility of providing multiple views of a class by providing more detailed type information.

The `typeid()` Operator

In §1, we presented the `typeid()` operator only briefly before making its use implicit in the dynamic cast mechanism. However, `typeid()` can be used explicitly to gain access to information about types at run time; `typeid()` is a built-in operator. Had it been a function its declaration would have looked something like this:

```
class Type_info;
const Type_info& typeid(type-name); // pseudo declaration
const Type_info& typeid(expression); // pseudo declaration
```

That is, `typeid()` returns a reference to an unknown type called `Type_info`. Given a *type-name* as its operand, `typeid()` returns a reference to a `Type_info` that represents the *type-name*. Given an *expression* as its operand, `typeid()` returns a reference to a `Type_info` that represents the type of the

object denoted by the *expression*. For example:

```
class B { /* ... */ virtual void f(); }; //  a polymorphic base class
class D : public B { /* ... */ };

B* p = new D; // a B* pointing to a D
B& r = *p;    // a B& referring to a D
int i;

typeid(p) == typeid(B*)
typeid(*p) == typeid(D)
typeid(r) == typeid(D)
typeid(&r) == typeid(B*)
typeid(7) == typeid(int)
typeid(0) == typeid(int)
typeid(i) == typeid(int)
typeid(&i) == typeid(int*)
```

Note that for a polymorphic type accessed through a pointer or a reference the actual object is examined and its (dynamic) type returned. For a non-polymorphic type the object returned represents the static type:

```
class X { int i };  //  a non-polymorphic class
class Y : public X { int j; };

X* xp = new Y;  // unwise: typeid(*xp) == typeid(X)
```

As ever, manipulating a non-polymorphic class through a base class relies on the programmer knowing exactly what is being done.

Because `typeid(*p)` involves examining the object `*p` the case `p==0` presents a problem. The solution is to throw an exception:

```
p = 0;
typeid(*p);     // throw Bad_typeid
```

Naturally, a simple test prevents the exception:

```
if (p == 0) {
        // ...
}
else {
        typeid(*p);
        // ...
}
```

Actually, `typeid()` slightly favors the use of references:

```
void f(B& r)
{
        if (typeid(r) == typeid(D) {
                // use r as a D
        }

        // used r as a plain B
}
```

Here we are entitled to assume that `r` refers to an object and we don't have to decorate `r` with any operators the way we had to decorate a pointer `p` with a dereference operator to get the type of the object `*p`.

The reason `typeid()` returns a reference to `Type_info` rather than a pointer 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 `Type_info&` there is no problem defining == to cope with such duplication.

Some `typeid()`s can be obtained only using the `typeid(`*type-name*`)` syntax. For example:

```
char& r = obj;
typeid(r) == typeid(char)   // NOT typeid(char&)
```

It is possible, however, to express the `typeid()` for every type that an object can have. This is important for writing code, such as some object I/O systems, that relies on using descriptions of objects at run-time.

Class Type_info

Class `Type_info` is defined in the standard header file `<Type_info.h>` which needs to be included for the result of `typeid()` to be used. The exact definition of class `Type_info` is implementation dependent, but it is a polymorphic type that supplies comparisons and an operation that returns the name of the type represented:

```
class Type_info {
        // implementation dependent representation
private:
        Type_info(const Type_info&);              // objects cannot
        Type_info& operator=(const Type_info&);   // be copied by users
public:
        virtual ~Type_info();                     // is polymorphic

        int operator==(const Type_info&) const;   // can be compared
        int operator!=(const Type_info&) const;
        int before(const Type_info&) const;       // define order among
                                                  // Type_info objects

        const char* name() const;                 // get the type name
};
```

More detailed information can be supplied and accessed as described below. However, because of the great diversity of the "more detailed information" desired by different people and because of the desire for minimal space overhead by others, the services offered by `Type_info` are deliberately minimal.

Extended Type Information

Consider 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 member of a class. 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:

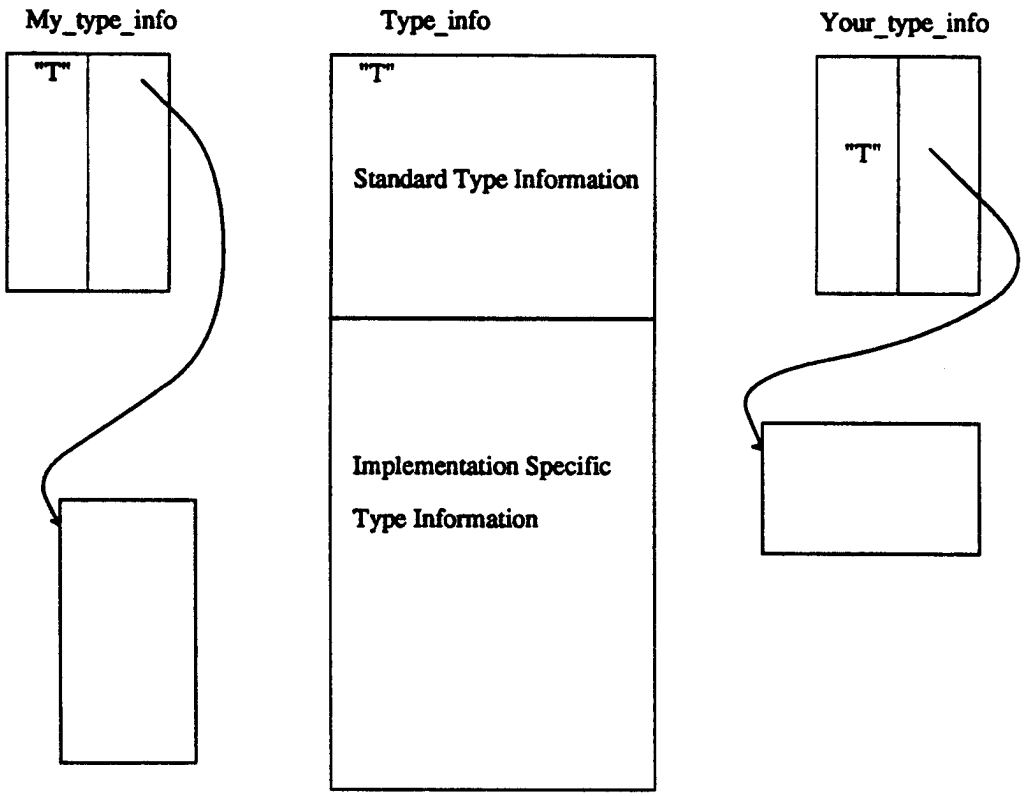
```
#include <Type_info.h>

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

where `My_member_info` is the name of the type of our information, and `my_type_table` is the name of the associative array in which we keep the *(typename,My_member_info*)* 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* pi = 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 users 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:



An implementation may chose to provide additional implementation specific type information. Such system provided extended type information could either be accessed through an associative array exactly as user-provided extended type information. Alternatively, the extended type information could be presented as a class `Extended_type_info` derived from class `Type_info`. A `dynamic_cast` can then be used to determine if a particular kind of extended type information is available:

```
#include <Type_info.h>

typedef My_Extended_type_info Eti;

if (Eti*p = dynamic_cast<Eti*>(&typeid(*p))) {
        // ...
}
```

What "extended" information might a tool or an implementation make available to a user? Basically any information that a compiler can provide and that some program might want to take advantage of at run time. For example:

- Object layouts for object I/O and/or debugging.
- Tables of functions together with their symbolic names for calls from interpreter code.
- Lists of all objects of a given type.
- References to source code for the member function.
- Online documentation for the class.

The reason such things are supported through libraries, possibly standard libraries, is that there are too many needs, too many potentially implementation specific details, and too much information to support every use in the language itself. Also, some of these uses subvert the static checking provided by the language. Others impose costs in run time and space that we do not feel appropriate for a language feature.

This reflect the observation an the implementation, and only an implementation, can supply extra

information by adding it to "the end" of the standard type information as shown in the figure. Naturally, such information can contain references to information allocated elsewhere.

The function `Type_info::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 obtain string representations of type names. We would prefer it to be trivially easy to print the name of a class. For example:

```
#include <Type_info.h>

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 string returned by `Type_info::name()` is a name suitable for printing and will identify the class for most practical purposes. It is not, however, guaranteed to be unique. For example, two functions may each have a local class called `Link` and `typeid(Link).name()` whould both be `Link`. The function `Type_info::before()` is provided to allow users to build ordered collections of type identifiers.

7 Implementation Issues

Consider how to implement RTTI. The `typeid` and the `dynamic_cast` operators affect syntax checking and type checking minimally. 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 `typeid` and `dynamic_cast`?

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

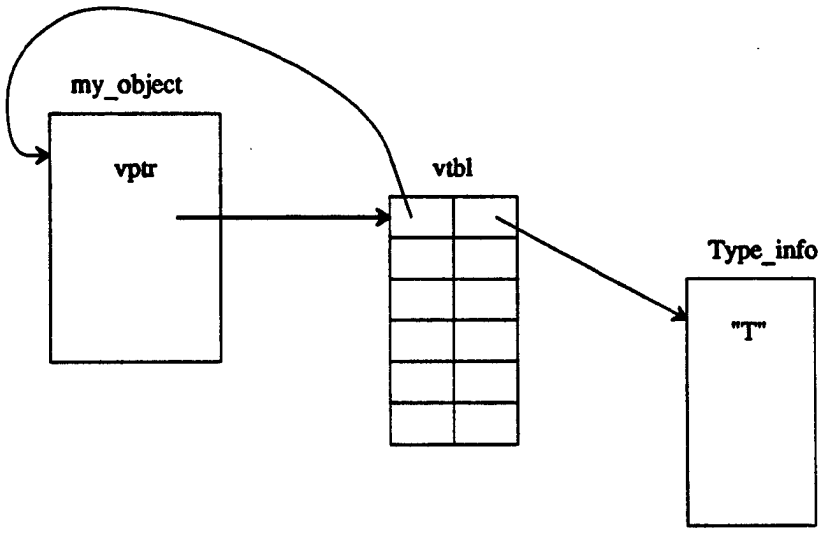
The implementation described here is only one of several possible. It assumes a traditional and fairly straightforward implementation of C++ along the lines described in [2]. That is, each object of a class with virtual functions contains a pointer (`vptr`) to a table of virtual functions (`vtbl`).

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`.

Basically `typeid(`*expression*`)` 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 `typeid(`*type-name*`)` degenerates into the name of the type's `Type_info` 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. An easy implementation simply places the `Type_info` object for a class right next to its `vtbl`.

Dynamic Casts

In most cases the implementation of a cast `dynamic_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 = dynamic_cast<D*>(px);  // pd should point to
                                   // the start of the D object
}
```

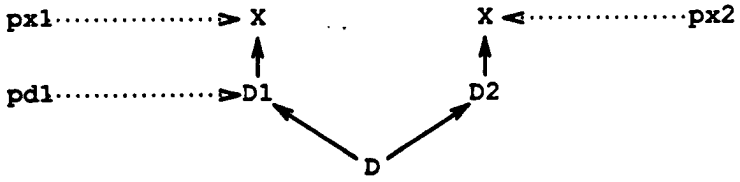
This adjustment is trivially implemented.

However, cases where a base class `X` appears more than once in a class hierarchy need 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 = static_cast<D1*>(pd);
    X* px2 = static_cast<D2*>(pd);

    pd =  dynamic_cast<D*>(px1); // make pd point to the start of the D
    pd =  dynamic_cast<D*>(px2); // make pd point to the start of the D

    D1* pd1 = dynamic_cast<D1*>(px1);
    pd1 = dynamic_cast<D1*>(px2);
}
```

Or graphically



Clearly the adjustments needed for the two `dynamic_cast<D*>` casts are different. Similarly, the adjustments needed for the two `static_cast<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. Virtual base classes are handled slightly differently; see Appendix B.

Another way of describing the complexity of this is to mention that the first author implemented and tested an experimental version of this proposal in two mornings (9am-11am). This was done with a C++ implementation that did not support exception handling. Adding RTTI to an implementation that supports exception handling ought to be simpler because such an implementation will already have the equivalent of `Type_info` objects. Naturally, a production quality implementation will take somewhat longer to produce.

8 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.

Dynamic and Ordinary Casts

Casts are one of the most error-prone facilities in C++. It is also one of the ugliest syntactically. Naturally we considered if we could

[1] eliminate casts;
[2] make casts safe;
[3] provide a cast syntax that makes it obvious that an unsafe operation is used;
[4] provide alternatives to casting and discourage the use of casts.

Basically, this proposal reflects our conclusion that a combination of [3] and [4] seems feasible.

Considering [1], we observed that no language supporting systems programming has completely eliminated the possibility of casting and that even effective support for numeric work requires some form of type conversion. Thus the aim must be to minimize the use of casts and make them as well behaved as possible. Starting from that premise we devised a proposal that unified dynamic and static casts using the old-style cast syntax [11]. This seemed a good idea, but upon closer examination several problems were uncovered:

[1] Dynamic casts and ordinary unchecked casts are fundamentally different operations. Dynamic casts look into objects to produce a result and may fail giving a run-time indication of that failure.

Ordinary casts perform an operation that is determined exclusively by the types involved and doesn't depend on the value of the object involved (except for occasional checking for 0 pointers). An ordinary cast doesn't fail; it simply produces a new value. Using the cast syntax for both dynamic and static casts led to confusion about what a give cast expression really did.

[2] If dynamic casts are not syntactically distinguished it is not possible to find them easily (grep for them, to use Unix-speak).

[3] If dynamic casts are not syntactically distinguished then it is not possible to have the compiler check for unsuitable uses of dynamic casts. If distinguished, we can make it an error to attempt a dynamic cast for objects that don't support run-time checking.

[4] Programs using ordinary casts would have their meaning changed if run-time checking were applied wherever feasible. Examples are casts to undefined classes and casts within multiple inheritance hierarchies. We did not manage to convince ourselves that this change of meaning would never break a reasonable program.

[5] The cost of checking would be incurred even for old programs that already carefully dynamic that casts were viable using other means.

[6] The suggested way of "turning off checking," casting to and from `void*`, wouldn't be perfectly reliable because the meaning would be changed in some cases. These cases might be perverted, but because understanding of the code would be required the process of "turning off checking" would be manual and error-prone. We are also against techniques that would add yet more uncheckable casts to programs.

[7] Making some casts "safe" would make casting more respectable; yet the long-term aim is to decrease the use of all casts (including dynamic casts).

After much discussion we found this formulation: "Would our ideal language have more than one notation for type conversion?" For a language that distinguishes fundamentally different operations syntactically the answer is "yes." Consequently we abandoned the attempt to "hijack" the old cast syntax.

We considered if it would be possible to "deprecate" the old cast syntax in favor of something like:

```
Checked<T*>(p);   // run-time checked conversion of p to a T*
Unchecked<T*>(p); // unchecked conversion of p to a T*
```

This would eventually make all conversions obvious, thus eliminating the problem that traditional casts are too hard to spot in C and C++ programs. It would also give all casts a common syntactic pattern and share the `<T*>` notation for types with templates. After some experimentation with other notations, especially `(?T*)p` for dynamic casts, and much discussion we settled on the current proposal.

The `(?T*)p` notation mirrored the cast syntax `(T*)p` too closely for some people's taste and was considered "far too cryptic" by others. Also, a critical flaw was found. The most common mistake when using `(?T*)p` was to forget the `?`. This turns what was meant to be a relative safe and checked conversion into a completely different, unchecked, and unsafe operation. For example:

```
if (dbox_w_string* p = (dbox_w_string*)q) // dynamic cast
{
        // q pointed to a dbox_w_string
}
```

Forgetting the `?` and thus turning the comments into lies was found to be uncomfortably common. Note that we cannot grep for occurences of old-style casts to protect against this kind of mistake and those of us with a strong C background are most prone to make the mistake and also to miss it when reading the code.

Early versions of the run-time checked cast could, like C-style casts, be used to violate abstarctions by casting to a private base class. After much debate, this was deemed neither necessary, sufficient, nor nice (see [13]).

Implementation and Tool Concerns

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 them in the language and then relying on compilers and other tools for optimization.

We were only partially successful. A previous proposal [11] had that property, but providing it

involved notations and concepts that many deemed confusing and too complicated.

The proposed solution involves three extensions to the syntax:

[1] The `dynamic_cast<`*type-name*`>(`*expression*`)` syntax for dynamic casts.

[2] The `typeid(`*type-name*`)` syntax for "`typeid` literals."

[3] The `typeid(`*expression*`)` syntax for getting type information from an object.

If the proposal to allow explicit qualification of template functions in [13] is accepted then [1] isn't new syntax. The syntax for [2] and [3] closely resembles the syntax for `sizeof`.

The `typeid()` Operator

We felt that the `typeid()` operator was more appropriate than a "magic" member function that could be applied to all objects. Had we defined `typeid()` as a member function we would have had to allow something like:

```
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 had been taken into account, the "magic" member function solutions looked messy.

Type Relations

We considered defining <, <=, etc., on `Type_info` objects to express relationships in a class hierarchy. That is easy, but too cute. It also suffers from the problems with an explicit type comparison operation as described in §1. We need a cast in any event so we can just as well use a dynamic cast.

Unconstrained Methods

There are many ways of using run-time type information in a language and a diverse set of facilities has been used in programming 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 dynamically-checked functions. However, we felt no need for that and considered that extension as 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 dynamic cast enables a check-and-call strategy:

```
if (D* pd = dynamic_cast<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 visible clue that something beyond the ordinary is going on.

Multi-methods

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. Such a language facility 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 in C++. In the context of C++, we would have to work out argument conversions and ambiguity rules, find a call mechanism that approached the virtual call mechanism in efficiency, and work out the interaction between multi-method declarations and separate compilation.

9 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] We use dynamic casts. The alternatives are checking all casts where sufficient information is available (§8) or relying on some alternative notion such as an `isKindOf` operator (§1, §8) or a relational operator on `typeid()`s (§8) for determining inheritance relationships.

[2] We use virtual functions to distinguish types that support run-time type identification from types that don't. The alternative would be to support RTTI for all types or to support RTTI for types explicitly declared to support it (§3, Appendix C).

[3] We use a syntax extension to allow declarations in conditions (Appendix A).

[4] We allow cross hierarchy casting. The alternative is to allow casts only within known class hierarchies (§4).

[5] We use reference casts. The alternative is to disallow reference casts and thus avoiding the use of exceptions (§5).

[6] We disallow objects of non-polymorphic types as operands for dynamic casts except for safe conversions (such as, a cast from a derived class to a public base). The alternative is to interpret such dynamic casts as ordinary static casts (§3).

[7] We allow casts to a non-unique sub-object from within an object. The alternative is to define casting as conversion from the run-time determined class of the object to the desired type and then consider a cast to a non-unique sub-object ambiguous (§7).

[8] The `dynamic_cast` operator respects abstraction boundaries §2.

[9] We use the `typeid` operator (§6). The alternatives are to provide no way of getting access to objects describing a type or to provide a complete `typeid` type for manipulating type identifiers instead of using `Type_info` objects directly [11].

[10] We allow non-polymorphic types as operands for `typeid()` and in such cases `typeid()` yields values that depend on the static type of its operand. The alternatives is to cause compile time errors or supplying RTTI for every object (§6).

[11] We allow expressions of any type as operands to `typeid()`. The alternative is to accept pointers and/or references only (Appendix D).

[12] We use a `Type_info` class defined in a standard library. The alternative is to support dynamic casts and type identity only (§6).

[13] We use a minimal `Type_info` class. The alternative is to guarantee the presence of a much more extensive type information class.

10 How to Manage until RTTI comes

This proposal for RTTI is most unlikely to be available on your C++ implementation any day soon. What can you do to get the benefits until some variant RTTI 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 [10]. The real problem is how to stay compatible with others and to make sure that you can convert the "real" RTTI system once it becomes available.

We suggest you write your code in terms of five macros

```
const Type_info& static_type_info(type)    // get Type_info for type
const Type_info& ptr_type_info(pointer)    // get Type_info for pointer
const Type_info& ref_type_info(reference) // get Type_info 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.

11 Acknowledgements

Jim Coplien, 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. 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. Steve Clamage found (too) many minor mistakes in an earlier version of this paper.

The current proposal evolved from the one presented to the ANSI/ISO C++ committee for discussion and published to solicit further comments [11] and from the version of this paper presented at USENIX [12]. The discussion extensions working group ANSI/ISO C++ committee at the London meeting was particularly useful. We found almost universal application of run-time type identification in various forms, confirmed the general structure of the proposal, and – somewhat to our surprise – demonstrated that static and dynamic casts could not be unified by a single syntax. Thanks to all who took part in that discussion.

The proposal has matured under discussion. The syntax is simpler than what was originally proposed and, significantly, `dynamic_cast` respects abstraction barriers. However, the fundamental idea as reflected in the run-time support described in §7 has remained completely unchanged.

12 References

[1] Frank Buschmann, Konrad Kiefer, and Michael Stal: *A Runtime Information System for C++*. Proc. TOOLS Europe 1992.

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

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

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

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

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

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

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

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

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

[11] Bjarne Stroustrup and Dmitry Lenkov: *Run-Time Type Identification for C++*. Long version for ANSI/ISO committee discussions: ANSI/X3J16 document 92-00028. Shorter version: The C++ Report, Vol.4 No.3, pp 32-42. March/April 1992.

[12] Bjarne Stroustrup and Dmitry Lenkov: *Run-Time Type Identification for C++ (Revised)*. Proc USENIX C++ Conference. Portland Oregon. August 1992.

[13] Bjarne Stroustrup: *An alternative to Old-style Casts*. ANSI/X3J16 document x3j16/92-0122,

WG21/N0199.

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

13 Appendix A: Declarations in Conditions

In §1 we mentioned in passing that we'd like to allow the use of declarations in conditions:

```
void my_fct(dialog_box* bp)
{
    if (dbox_w_str* dbp = static_cast<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. For example:

```
do f() while(int i = g());          // error: declaration in do condition

while(int i = g()) f();             // ok

while(int i = g(), j = g2()) f(); // error: two names declared in condition
```

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(double d)
{
    if (X x1 = d) {
        // 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'
    // ...
}
```

The usual declaration/expression ambiguity reappears in the context of conditions. For example:

```
if ( int(i) = j) // assignment to cast, or
                 // declaration with redundant parentheses?
```

As usual, the ambiguity is resolved in favor of the declaration. This implies an incompatibility in the (very rare) case where a function-style cast that is an lvalue is used on the left hand side of an assignment that is a condition:

```
typedef Rint& r;

void f(Rint& rr, int i)
{
        if (Rint(r)=i) // used to be an assignment, becomes declaration
                // ...
}
```

14 Appendix B: Casting from Virtual Bases

It is not possible to cast from a virtual base class to a derived class using an ordinary cast. This restriction does not apply to dynamic casts from polymorphic virtual base classes:

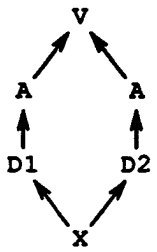
```
class B { /* ... */ virtual void f(); };
class V { /* ... */ virtual void g(); };

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

void g(D& d)
{
    B* pb = &d;
    D* pd1 = static_cast<D*>(pb);  // ok, unchecked
    D* pd2 = dynamic_cast<D*>(pb); // ok, run-time checked

    V* pv = &d;
    D* pd3 = static_cast<D*>(pv); // error: cannot cast from virtual base
    D* pd4 = dynamic_cast<?D*>(pv); // ok, run-time checked
}
```

The reason for the restriction to dynamic casts from polymorphic classes is that there isn't enough information available in other 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 dynamic 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 = dynamic_cast<A*>(pv); // pa will be initialized to 0
}
```

This ambiguity is not in general detectable at compile time:

```
void h2(V* pv)
{
    A* pa = dynamic_cast<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; §8.

15 Appendix C: Explicit RTTI Declaration

Experience shows that providing dynamic casts for polymorphic types (only) works acceptably. However, people can get confused about which types are polymorphic. 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 `rtti` 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 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 dynamic cast from or leave it to the compiler to tell you.

16 Appendix D: Alternative `typeid()` Semantics

We considered two alternatives for the semantics of `typeid()`. Both are consistent and roughly equivalent. This appendix explains why we chose the one we did. Because we saw no major flaw in either alternative the discussion gets a bit involved.

To help the discussion, let's here call the two alternatives `ptypeid()` and `otypeid()`. The `typeid()` semantics adopted and described in §6 is that of `otypeid()`. We will assume the definitions

```
class B { /* ... */ virtual void f(); }; //  a polymorphic base class
class D : public B { /* ... */ };

B* p = new D; // a B* pointing to a D
B& r = *p;    // a B& referring to a D
```

Semantics of `ptypeid()`

The original idea was `ptypeid(p)` meaning "the object describing the object pointed to by the pointer p." One could imagine `ptypeid()` to take an argument of any pointer type:

```
template<class T> Type_info& ptypeid(T*);
```

The expected most common uses are:

```
if (ptypeid(p) == typeid(D))  // is the object pointed to by p a D?
if (ptypeid(&r) == typeid(D)) // is the object referred to by r a D?
```

Clearly the design of `ptypeid()` is geared to making the use of pointers convenient. What then if p is `0`? This is easily handled by letting `typeid(0)` denote an object representing the `0` pointer; `typeid(0)` compares not equal to `typeid(T)` for every type `T`:

```
p = 0;
if (ptypeid(p) == typeid(D)) // fails
if (ptypeid(p) == typeid(0)) // succeeds
```

It is a compile time error to apply `ptypeid()` to a non-pointer:

```
int i;
ptypeid(i);   // error: 'i' is not a pointer
ptypeid(7);   // error: '7' is not a pointer
ptypeid(*p);  // error: '*p' is not a pointer
```

So far, so good. Now consider references:

```
ptypeid(r);   // == ptypeid(*p), that is, error, or
              // == typeid(D) ?

B*& pr = p;   // pr refers to a B*

ptypeid(pr);  // == ptypeid(p) == typeid(D) or
              // == typeid(B*) or
              // == typeid(D) ?
```

In both cases, the first alternative is obtained by assuming that there is no special interaction between `ptypeid()` and references so that `ptypeid()` is applied to the object referred to, that is `*p` (of type `B`) a `p` (of type `B*`) respectively. The second alternative assumes that `ptypeid()` treats a reference similar to the way it treats a pointer, that is, it looks at the object referred to and finds its type. The third alternative for `ptypeid(pr)` is obtained by saying that both references and pointers are followed until a non-pointer and non-reference is found and that object is examined. The third alternative, "just chase pointers and

references as far as we can" is an approach that has caused problems with other type systems and would be unique in C++ so we rejected that.

We considered the first two alternatives plausible. Using the first alternative, the result of `ptypeid(r)` will be surprising to many because it is the static type of the object referred to (`B`). Using the second alternative, the result of `ptypeid(pr)` will be surprising to many because `ptypeid(pr)` will differ from `ptypeid(p)` even though `pr` refers to `p`. In both cases comparisons with `typeid(D)` will fail.

Having `ptypeid(pr)` differ from `ptypeid(p)` even though `pr` refers to `p` seemed too odd a departure from the general rule that a name of an object and a reference to that object behave identically. Thus,

```
B& r = *p;     // r refers to a D
B*& pr = p;    // pr refers to a B*

ptypeid(r);    // == ptypeid(*p), that is, error
ptypeid(pr);   // == ptypeid(p) == typeid(D)
```

This means that to use `ptypeid()` effectively on a "typical reference argument" we must obtain a pointer using the address-of operator:

```
void f(B& r)
{
        if (ptypeid(&r) == typeid(D))
        // ...
}
```

This works nicely except where `&` is overloaded for `B`. It does look odd, though.

We conclude that `ptypeid()` can be made to work acceptably within the type system, but that there are a few details that are less than elegant.

Semantics of `otypeid()`

Consider the `otypeid()` alternative: `otypeid(obj)` yields an object representing the type of `obj`. One could imagine `otypeid()` to take an argument of type "reference to any type:"

```
template<class T> const Type_info& otypeid(T&);
```

The expected most common uses are

```
if (otypeid(r) == typeid(D))  // is the object referred to by r a D?
if (otypeid(*p) == typeid(D)) // is the object pointed to by p a D?
```

Clearly the design of `otypeid()` is geared to making the use of references convenient. The `otypeid()` operator differs from the mythical `typeof()` operator only in that its call by reference semantics ensured that the dynamic type of an object is determined rather than its static type. For example, the (static) type of the expression `*p` is `B`, whereas `otypeid(*p)` is the type of the (dynamic) object pointed to by `p`, that is, `D`.

Applying `otypeid()` to non-pointers is no problem:

```
int i;
otypeid(i);    // == typeid(int)
otypeid(7);    // == typeid(int)
```

This implies that `otypeid(0)` isn't special:

```
p = 0;
otypeid(p);    // == typeid(B*)
otypeid(0);    // == typeid(int)
```

There is however, however, a problem related to "zero references:"

```
p = 0;
otypeid(*p);  // == typeid(B) or
              // == typeid(void) or
              // throw exception ?
```

The first alternative simply returns the static type of `*p` if there is no object to examine. This is confusing and error prone. The second alternative returns a distinguished object (much as `ptypeid(0)` yields `typeid(0)`). The third alternative relies on the observation that any use of `*p` where `p==0` would be an error. The probable most common case would be the one where a reference `r` had somehow been bound to a non-object; that is, `&r==0`.

Any use of such an `r` will cause an error. The choice is between an explicit test to avoid such a use and the possibility of throwing an exception. Since one can already test for `&r==0` and `p==0` there is no need for an additional test `otypeid(*p)==typeid(void)`. Throwing an exception provides an implicit mechanism for detecting such errors. Therefore:

```
p = 0;
otypeid(*p);  // throw exception
```

Again, the difference between pointers and references shows up in the way a reference to a non-existent object is handled.

Comparison

Now consider a summary:

```
class B { /* ... */ virtual void f(); }; //  a polymorphic base class
class D : public B { /* ... */ };

B* p = new D; // a B* pointing to a D

B& r = *p;    // r refers to a D
B*& pr = p;   // pr refers to a B*

ptypeid(p) == typeid(D)        otypeid(p) == typeid(B*)
ptypeid(*p) error              otypeid(*p) == typeid(D)
ptypeid(r) error,              otypeid(r) == typeid(D)
ptypeid(&r) == typeid(D)       otypeid(&r) == typeid(B*)
ptypeid(pr) == typeid(D)       otypeid(pr) == typeid(B*)
p = 0;
ptypeid(p) == typeid(0)        otypeid(p) == typeid(B*)
ptypeid(*p) error              otypeid(*p) throw exception
```

If either `ptypeid()` or `otypeid()` should be called `typeid()` which would you choose? We chose `otypeid()` for several minor reasons.

We wanted a single `typeid()` operator that could be applied to both expressions and types. This weighed against the `ptypeid()` semantics. Having an implicit dereference for expression arguments but not for type arguments seemed odd:

```
typeid(p) == typeid(D) // but the type of p is B*, not D
```

This simple point becomes significant when thinking about templates. For example:

```
template<class T> const char* snameof(T& r)  // return name of static type
{
        return typeid(T).name();
}

template<class T> const char* dnameof(T& r)  // return name of dynamic type
{
        return typeid(r).name();
}
```

are simple to write using the `otypeid()` model.

Whatever we choose, somebody will make false assumptions about the model used for `typeid()` and write the equivalents to:

```
if (otypeid(p) == typeid(D))  // error or simply failed test?
// ...
if (ptypeid(*p) == typeid(D)) // error
```

The test will fail because `otypeid(p)` is `typeid(B*)`. However, this is a trap and might warrant a compiler warning. Making it an error to compare the `otypeid()` of a pointer to the `typeid()` of a non-pointer seems Draconian, though, and might complicate the writing of templates. Error handling is easier for the `ptypeid()` semantics so this favors the `ptypeid()` semantics.

If the address-of operator has been overloaded for a class then we cannot take the `ptypeid()` of a reference to that class:

```
class X {
        // ...
        Xhandle operator&();
};

X x;
X& r = x;
ptypeid(&r);    // error ptypeid() of non-pointer.
```

This could be a nasty problem. There is no equivalent problem for the `otypeid()` semantics because `*p` is the application of a built-in operator to a pointer.

Finally, consider zero pointers. Using `ptypeid()` we would write

```
void f(B* p)
{
        if (ptypeid(p) == typeid(D) {
                // use p as a D*
        }

        // used p as a plain B*
}
```

and get burned if we use `p` as a plain `B*` without testing for `0`. We could test for zero immediately upon entering `f()` or after the type test. Since `typeid(0)` is well defined it doesn't matter whether `p==0` is checked before or after.

Using `otypeid()` we would write

```
void f(B& r)
{
        if (otypeid(r) == typeid(D) {
                // use r as a D
        }

        // used r as a plain B
}
```

We are entitled to assume that `r` refers to an object and would normally not test for `&r==0`. If someone had cheated and passed a zero reference `otypeid(r)` will throw an exception. This is in our opinion preferable to the undefined behavior we'd get from using `r`. If we felt the need to check for `&r==0` we'd have to do it before using `otypeid(r)`:

```
void f(B& r)
{
        if (&r == 0) { // something is rotten
                // ...
        }
        else if (otypeid(r) == typeid(D) {
                // use r as a D
        }

        // used r as a plain B
}
```

We consider the behavior in the case of p==0 very marginally in the favor of the `otypeid()` semantics.

Allowing `typeid(oo)` where the result will not depend on any run-time information – that is, where `oo` does not refer to a polymorphic type – could be considered redundant and therefore a possible source of confusion and errors. Instead of `typeid(oo)` you could use `typeid(T)` where `T` is the type of the object referred to by `oo`. This is an argument for the `ptypeid()` semantics. However, if you didn't declare `oo`, you don't necessarily know its type and whether that type is polymorphic or not. This can happen with templates with templates.

Further, suppose you're trying to define some kind of smart pointer class. If `typeid()` applies only to pointers, then it would be impossible to make `typeid()` work transparently with smart pointers. That is, if `p` is a smart pointer, `ptypeid(p)` would be illegal because `p` isn't what C++ thinks of as a pointer. However, `otypeid(*p)` would be whatever the type of `*p` is and `operator*()` can be defined for a smart pointer type.

17 Appendix E: Return Types

The March 1992 meeting of the ANSI/ISO C++ standards committee in London decided – after almost two years of deliberations – to relax the rules for overriding virtual functions to allow a function returning a `B*` to be overridden by a function returning a `D*` when `B` is a public base class of `D`. Similarly, a function returning a `B&` can be overridden by a function returning a `D&`.

This provides an alternative to uses of casts that might have been considered candidates for RTTI. For example,

```
class X {
        // ...
        virtual X* clone(); // return copy of *this
};

class Y : public X {
        // ...
        X* clone();
}

void f(X* p, Y* q)
{
        X* pp = p->clone();
        Y* qq = static_cast<Y*>(q->clone()); // the clone of a Y is at least a Y

        if (Y* q2 = dynamic_cast<Y*>(pp)) { // was the X really a Y?
                // ...
        }
        // ...
}
```

The return type relaxation makes a better solution possible:

```
class Y : public X {
        // ...
        Y* clone(); // override X::clone()
}

void f(X* p, Y* q)
{
        X* pp = p->clone();
        Y* qq = q->clone(); //no cast (dynamic or static) needed

        if (Y* q2 = dynamic_cast<Y*>(pp)) { // was the X really a Y?
                // ...
        }
        // ...
}
```

We mention this to remind people that blindly changing all casts to dynamic casts isn't a good way to try to improve old programs.

18 Appendix F: `typeid()` During Construction

A call of `typeid(*p)` reflects the dynamic type of `*p` in the same was as a call of a virtual function `p->f()` would. This implies that in a constructor of destructor of `X` the call `typeid(*this)` will be `return(X)` rather than the `typeid()` of some derived class that the `X` object might be part of after construction and before destruction.

19 Appendix G: Reference Manual Changes

This appendix lists suggested manual changes.

Dynamic Cast
Add to §5.2:

postfix-expression:
... old stuff
`dynamic_cast <` *type-name* `> (` *expression* `)`

Add subsection §5.2.5 called "Dynamic Cast"

The result of `dynamic_cast<T>(v)` is of type `T`. The type `T` must be a pointer or a reference to a defined class or `void*`.

If `T` is a pointer and `v` is a pointer of a type for which `T` is an accessible base class the result is a pointer to the unique sub-object of that class. If `T` is a reference and `v` is an object of a type for which `T` is an accessible base class the result is a reference to the unique sub-object of that class. Otherwise `v` must be a pointer of reference to a polymorphic type. That is, the type pointed or referred to must have at least one virtual function.

If `T` is `void*` then `v` must be a pointer and the value is a pointer to the complete object (§12.6.2) pointed to by `v`.

Otherwise, a run-time check is applied to see if the value `v` can be converted to a `T`. If not, the cast is said to have failed. The value of a failed cast to pointer type is `0`. A cast to reference type throws `Bad_cast` if it fails.

The run-time check logically executes like this: If the object pointed (referred) to by `v` is an object representing a base class of an object of type `T` the result is a pointer (reference) to that object. Otherwise, find the complete object (§12.6.2) pointed to by `v`. If that object has a unique public base class of type `T`, the result is the object representing that base class. Otherwise, the run-time check fails.

Declarations in Conditions
Add to §6:

condition:
expression
type_specifier declarator = expression

and use *condition* instead of *expression* for the condition in the grammer for *selection-statement* (that is, `if` and `switch`) and in the `for` and `while` cases of *iteration-statement*. The *declarator* may not specify a function or and array. The *type_specifier* may not define a new class or enumeration.

A name introduced by a declaration in a *condition* is in scope from its point of declaration until the end of the statements controlled by the condition. The value of a *condition* that is an initialized declaration is the value of the initialized variable.

A variable, constant, etc. in the outermost block of a statement controlled by a condition may not have the same name as a variable, constant, etc. declared in the condition.

If a *condition* can be syntactically resolved as either an expression or the declaration of a local name, it is interpreted as a declaration.

`typeid` Operator
Add to §5.2:

postfix-expression:
... old stuff
`typeid ( expression )`
`typeid ( type-name )`

Add subsection §5.2.7 called "Type Identification"

The result of a `typeid` expression is of type `const Type_info&`. The value is a reference to a `Type_info` that represents the *type-name* or the type of the *expression* respectively.

Where the *expression* is a reference to a polymorphic type the *Type-info* for the complete object referred to is the result. Where the *expression* is a pointer to a polymorphic type dereferenced using * or []expression the *Type-info* for the complete object pointed to is the result. Otherwise, the result is the *Type-info* representing the (static) type of the *expression*.

`Type_info` Class
Class `Type_info` is declared in <`Type_info.h`> like this:

```
class Type_info {
        // implementation dependent representation
private:
        Type_info(const Type_info&);              // objects cannot
        Type_info& operator=(const Type_info&);   // be copied by users
public:
        virtual ~Type_info();                     // is polymorphic

        int operator==(const Type_info&) const;   // can be compared
        int operator!=(const Type_info&) const;
        int before(const Type_info&) const;       // define order among
                                                  // Type_info objects

        const char* name() const;                 // get the type name
};
```

precedes

more_info()

The ordering defined by `before` is complete and valid only for the duration of the execution of program. There is no guaranteed realtion between inheritance rerationships defined by `before` and inheritance relationships.