Extended Type Information (revised)

Konrad Kiefer, Frank Buschmann,
Frances Paulisch, Michael Stal
Siemens AG, Corporate Research and Development,
ZFE BT SE 3, Otto-Hahn-Ring 6, 8000 Munich 83, Germany
e-mail: kk@ztivax.zfe.siemens.de
date: October 28, 1992

Contents

Main modifications of the last version /7/

The discussion in the extensions working group at the Toronto meeting has been considered.

- All additional information has been moved to ExtTypeInfo
- The extended type-information has been reduced to support object I/O only
- A discussion of the functions newObject() and clone() is added
- const (volatile) types have a different Type_info object than non-const (non-volatile) types
- The structure of the information is simplified

1) Purpose of this paper

The proposal about run-time type identification /6/ (Ansi document X3J16/92-0068) includes a proposal for a standard library class Type_info. According to the discussion in the extensions working group two functions should be added: a comparison function, to allow storing of Type_info objects in a map, and a function to get access to more information. These topics will be addressed in section 2.

In sections 3 and 4, we discuss extended type information built on the minimal type info. The aim of extended type information is to support object I/O. In addition the extended type information should support a "generic constructor" that allows the construction of skeleton objects and a function "clone()" for copying objects. We discuss the problems of the functions newObject() and clone() but we didn't find any satisfying solution yet.

If extended type information can support object I/O with acceptable amount of information, it should be part of the standard library with the following status:

- the initialization of extended type information is implementation dependent; an implementation might offer options to generate extended type information or ignore it.
-Suppliers are allowed to extend the extended type information.

2) The standard library class Type_info

As the typeid operator may be applied to all types and expressions, it needs to be clarified, what the name of a type is.

Typenames for fundamental types

The typenames for fundamental types should be fixed in the definition of the class Type_info if we want to avoid incompatibilities. E. g. it must be stated, if we want to use "short" or "short int". We propose the following names for fundamental types:

"void",
"char", "signed char", "unsigned char", "wchar_t",
"short", "unsigned short",
"int", "unsigned int",
"long", unsigned long",
"float", "double", "long double".

Although we never have an object of type void but only void*, a definition of typeid(void) makes the type inquiry more consistent.

Typedef names

A typedef doesn't introduce a new type but an alias for an existing type. Therefore a typedef name shouldn't count as a type name in the class Type_info.

If we have:

```
class X;
typedef X Y;
X x;
Y y;
```

The following comparisons should yield true:

```
typeid(x) == typeid(y)
typeid(X) == typeid(Y)
```

Consequently we should have one Type_info object in all cases, returning the typename "X".

Typename for derived types

We propose the following rule:

If we have a derived type, we remove typedefnames and take the remaining type, as the essential type. The name in Type_info is then the C++ declaration name, where all superfluous blanks are removed.

Examples:

```
typedef int* pi;
pi* i1;
const int const** i2;
```

According to the rule i1 and i2 have the typename "int**" and "const int const**" respectively.

Further examples:

```
const char* a[10];
pi (*f)(int i);
```

The type of a is "const char*[10]" and f is "int*(*)(int)" (pointer to a function taking an int argument and returning int*). These rules seem to be consistent with the ARM /2/ section 8.1.

Typename for templates

A template declaration defines a family of types. Each member of this family is a single type and needs a corresponding Type_info object. A Type_info object for the family doesn't seem to be adequate. As for derived types the typename of templates should be the C++ declaration name by

leaving out typedefs. Examples taken from the ARM /2/, section 14:

```
template<class E, int size> class buffer;
buffer<char, 2*512> x;
buffer<char, 1024> y;
```

Then x and y have the same type and the same Type_info object with the typename "buffer<char,1024>".

The Functions moreInfo() and precedes()

The function Type_info::moreInfo() allows access to extended information.The minimal type information does not specify the class ExtTypeInfo. If only minimal type information is used, the result of moreInfo() is a NULL pointer. moreInfo only specifies the interface to more information. It is left to the implementation how additional information is added.

The function Type_info::precedes() allows ordering of Type_info objects. The ordering is needed for data structures and has no further semantic. (The name of the function has not been fixed in the extensions working group.)
Let X and Y be types. typeid(X).precedes(typeid(Y) is true, if the types X and Y are not identical an the internal (implementation dependent) order of typeid(X) is less then the one of Y.

"before"

Summary of the library class Type_info (see also /6/ p. 12)

```
class ExtTypeInfo;
class Type_info {
      // implementation dependent
private:
       Type_info(const Type_info&);
      Type_info& operator=(const Type_info&);
public:
      virtual        ~Type_info();
      int            operator==(const Type_info&)const;
      int            operator!=(const Type_info&)const;
      const char*    name() const; // String class instead of char* might be preferable
      int            precedes(const Type_info&)const;
      const class ExtTypeInfo* moreInfo() const;
};
```

ask Dmitry

The class Type_info is intentionally kept as minimal as possible. All additional information will be put in the extended type information.

3) The extended type information

Why object I/O based on type information?

If we do not want to write I/O operators for every class or rely on a special compiler that supports it, we may generate the operations with a tool. This has the disadvantage that we have to change the classes, which is often not desirable or even possible. Another approach to get the desired feature is to store enough information about the program, that a clever algorithm might handle 'most' cases. The principles of this algorithm are:

We get the address of objects and know what kind of object we have.

- If we have a simple type, we may simply store the value.
- If we have a pointer or a reference, it might be necessary to follow them.
- If we have a class or structure, we must get the address of the components (offsets) and apply the algorithm to them.

The main problems of such an algorithm in C++ are some type leaks inherited from C. A simple example is "char*". Without further information, the type information system cannot know if the declaration represents a string or a simple character pointer. It is possible to solve this and related problems with programming conventions, but there doesn't seem to be a general solution. The handling of these cases is therefore left to the special I/O algorithm.

Structure of extended information

We redesigned the MOP system /1/ to get a structure that

- is compatible to minimal type information in /5/ and /6/,
- reflects some basis aspects of the type system of C++,
- allows extensions
- stores enough information to support object I/O for 'most' cases

In addition to the class Type_info we have the following classes.

- The class ExtTypeInfo offers much more detailed information about a type than the class Type_info.
- The class BaseInfo stores Information about base classes.
- The class DataInfo holds information about data of classes

In addition we propose simple iterator classes to iterate over the information.

The semantic of the information falls into three layers.

- Layer 1: Information needed for implementing the checked cast
- Layer 2: Type information needed for object I/O
- Layer 3: Information about offsets and store functions (newObject())

Layer 3 is needed by an I/O algorithm based on type information, but it would be dangerous to make this information available to the public. Visible offsets of private members break the protection of a class. Therefore we propose to protect the offsets functions in ExtTypeInfo. An implementation specific store function added to ExtTypeInfo is allowed to use the offsets, but not the user of ExtTypeInfo in general.

The functions newObject() and clone() are discussed separately in section 5, because we didn't find a satisfying solution until now.

4) The public interface of the classes for extended type information

The public interface of the class ExtTypeInfo

```
class BaseIter;
class DataIter;
class TypeIter;

class ExtTypeInfo {
      // implementation dependent
 public:
```

```
// Information layer 1
    BaseIter*           bases(int direct = 0) const;
    BaseIter*           vbases(int direct = 0) const;

// Information layer 2
    enum typeKind {
        // fundamental types
        simpleKind,
        // derived types
        arrayKind,
        ptrKind, ptrMemberKind,
        refKind,
        functionKind,
        // enums
        enumKind,
        // classes, structs and unions
        classKind, structKind, unionKind
        };
    enum protection {
        PRIVATE,
        PROTECTED,
        PUBLIC
        };
    size_t              size() const;
    typeKind            kindOf() const;
    DataIter*           data(int direct = 0) const;
    DataIter*           staticData(int direct = 0) const;
    TypeIter*           typeComponents() const;
    const  Type_info*   ptrMemType() const;
};
```

Description of the member functions of ExtTypeInfo

BaseIter* bases(int direct = 0) const

for classes: Pointer to an iterator over the non virtual base classes.
Otherwise: NULL pointer

BaseIter* vbases(int direct = 0) const

for classes: Pointer to an iterator over the virtual base classes.
Otherwise: NULL pointer

size_t size() const;

The result of size() is the sizeof() operator applied to the type.

typeKind kindOf()const

for all types: returns the kind of the type

DataIter* data(int direct = 0)const
for classes (including structs and unions): returns an iterator over the direct (optional also inherited) non-static data members
other types: returns a NULL pointer

DataIter* staticData(int direct = 0)const
for classes (including structs): returns an iterator over the direct (optional also inherited) data members
other types: returns a NULL pointer

TypeIter* typeComponents()const
derived types: returns an iterator over the type components. The function TypeIter::next returns Type_info objects that describe the type.
other types: NULL pointer
examples:
1) const char* a[10];
The actual type information yields: array
The TypeIter has two further steps with the information:
- Pointer
- simple type (the name "const char" can be obtained in the corresponding Type_info object.)
2) char* (*f)(int i)
The actual type information yields: Pointer
The TypeIter has three further steps with the information:
- function
- Pointer
- simple type

Type_info* ptrMemType()const
pointer to member: Type_info of the class
other types: NULL pointer

The public and protected interface of the class BaseInfo
The class BaseInfo stores information about base classes. Objects of BaseInfo are returned by the iterator BaseIter.

```
class BaseInfo {
    // implementation dependent
public:
// Information layer 1
    ExtTypeInfo::protection protectSpec()const;
    const Type_info*    typeOf() const;
// Information layer 3
protected:
    long                offset () const;
};
```

Description of the member functions of BaseInfo

ExtTypeInfo::protection protectSpec()const
returns the protection specifier of the corresponding base class;

const Type_info* typeOf()const
returns the Type_info object of the base class

long offset ()const
returns the offset of the base class part in this class;
Note that it is necessary to iterate over all virtual bases classes (call vbases() without the optional argument direct) to get the correct offsets of virtual bases.

Example:
class A {...};
class B: public virtual A {...}; class C: public virtual A {...};
class D: public B, public C {...};
Then the offset of A in D is in general not the sum of the offset of B in D and the offset of A in B.

The public and protected interface of the class DataInfo

The class DataInfo stores information about data members of classes. Objects of DataInfo are returned by the iterator DataIter.

```
class DataInfo {
    // implementation dependent
public:
// Information layer 2
    const Type_info*      typeOf() const;
    ExtTypeInfo::protection protectSpec() const;
    int                   bitfieldSize() const;
// Information layer 3
protected:
    long                  offset() const;
};
```

Description of the member functions of DataInfo

const Type_info* typeOf()const
returns typeid(member)

protection protectSpec()const
returns the protection specifier:

int bitfieldSize() const
returns the number of bits if the member is a bitfield
-1 otherwise

long offset()const
for non-static data members: returns offset in bytes
for static data members: returns the absolute address

The public interface of the iterator classes
A template iterator class might replace the iterator classes. The constructor of the iterator needs a pointer or reference to the data structure (here someData). The data structure is implementation dependent and not specified here.

```
class BaseIter {
    // implementation dependent;
public:
    const BaseInfo*     next();
    const BaseInfo*     reset();
    BaseIter(const someData* p, int direct = 0);
};

class DataIter {
    // implementation dependent;
public:
    const DataInfo*     next();
    const DataInfo*     reset();
    DataIter(const someData* p, int direct = 0);
};

class TypeIter {
    // implementation dependent;
public:
    const Type_info*    next();
    const Type_info*    reset();
    TypeIter(const someData* p);
};
```

Description of the public interface of the iterator classes

const SomeInfo* next()
The iteration function. After the construction of the iterator the function next returns the first element.

const SomeInfo* reset()
Sets the iterator at the beginning and returns the first element

5) Some comments on the functions newObject and clone()
The function newObject() and clone() can be easily defined as templates.

```
template <class T> class Typeid {
public:
      T* newObject(){return new T;};
      T* clone(T arg) {return new T(arg);};
};
```

The function newObject() refers to the default constructor and clone() to the copy constructor. The template class Typeid can only be created if default and copy constructor of the class are accessible.
However, the use of the template functions requires that we know the type of the object we want to create at compile-time.

```
class A {};
Typeid <A> ATypeid;
A* pA; // may point to a subclass of A
```

Now we cannot create a copy or call newObject for the actual type of pA. To manage this, we can put newObject() and clone() in the extended type-information and call the right template function for every class. E.g:

```
class ExtTypeInfo{
public:
//...
      void*   newObject();
      void*   clone(void* arg);
};
```

The problem in this example is the void* interface. An ugly and unchecked cast from void* would be necessary and would allow allocation storage for everything without control of the type system:

```
A* pA = (A*) typeid(int*).moreInfo->newObject().
```

The result would be, that pA points to the storage of an int.
We do not have a satisfying solution for this problem now.

6) References

/1/ Frank Buschmann, Konrad Kiefer and Michael Stal: A Run-time Information System for C++. Proc. TOOLS Europe 1992.

/2/ Margaret A. Ellis, Bjarne Stroustrup: The Annotated C++ Reference Manual, Addison Wesley, 1990

/3/ Dmitry Lenkov, Michey Metha, Shankar Unni: Type Identification in C++. Usenix C++ Conference Proceedings, April 1991.

/4/ Bjarne Stroustrup: The C++ Programming Language (Second Edition). Addison-Wesley, 1991.

/5/ Bjarne Stroustrup and Dmitry Lenkov: Run-Time Type Identification for C++. ANSI/X3j16 document 92-0028

/6/ Bjarne Stroustrup and Dmitry Lenkov: Run-Time Type Identification for C++ (Revised) ANSI/X3j16 document 92-0068

/7/ Konrad Kiefer, Frank Buschmann, Frances Paulisch, Michael Stal: Proposal for the library class Type_info, Example of extended type information. ANSI document X3J16/92-0074, WG21/ N0151