Document Numbers: X3J16/95-0174
WG21/N0774
Date: September 26, 1995
Reply To: Bill Gibbons
bgibbons@taligent.com

Definitions of "scalar type" and "fundamental type"

Introduction

The definition of "scalar type" in the working paper should include pointers to members, but the current wording excludes them. This appears to be an oversight, since pointers to members behave much more like scalars than like non-scalars. Making them scalars would be more consistent and simplifies the working paper.

The definition of "fundamental type" currently includes enumeration types. It seems odd that a user-defined type could be considered fundamental; it would be more appropriate to make enumeration a compound type. Doing so would be more consistent and simplifies the working paper.

These are very simple changes, but they affect very basic definitions. So it seems worthwhile to examine the effects in some detail.

Scalar Types

Data objects in C++ can be categorized as either scalar (e.g. integers and pointers) or non-scalar (e.g. arrays and classes), where scalars are primitive objects which contain a single value and are not composed of other C++ objects. Pointers to members appear to be scalars, but implementations generally represent them with multiple pieces of machine data. For example, a pointer to member function may contain an offset value and a function pointer.

Although implementors know that pointers to members usually consist of multiple parts, there is neither a good reason nor a portable mechanism for examining or modifying the individual parts. For purposes of understanding a C++ program, it is better to consider pointers to members to be indivisible primitive objects - that is, scalars.

The only significant impact of such a change is that pointers to members would have to be bitwise copyable, i.e. they could be copied using `memcpy`. Since existing implementations represent pointers to members using pointers and integers (which are bitwise copyable), it seems reasonable to allow `memcpy` on pointers to members.

Appendix A contains a complete list of uses of the term "scalar" in the working paper, together with a discussion of how scalar pointers to members would affect that section of the working paper. The overall impact is very small, and the net effect is a slight simplification of the working paper.

Fundamental Types

Enumerations are currently classified as fundamental types. But there is a small fixed set of other fundamental types, each of which is a builtin; and there is an infinite number of possible enumeration types, each of which is a user-defined type. So enumeration types do not resemble the other fundamental types very much.

The description of `numeric_limits` class is complicated by the fact that enumerations are fundamental, because `numeric_limits` is supposed to be specialized for all fundamental types. If enumeration types remain fundamental types, the description of `numeric_limits` must explicitly exclude them.

Similarly, under the current definition, the description of `limits.h` should explicitly exclude enumerations (obviously that header file cannot give limits for an infinite number of user-defined types).

There is nothing in the use of the term "fundamental type" in the working paper which would indicate that enumerations should be fundamental. So it would be simpler to move them to the "compound type" category.

Of course compound types are supposed to be based on fundamental types; but enumerations already have the concept of an *underlying type*; so this is the fundamental type on which the enumeration is based.

Appendix B contains a complete list of the uses of the term "fundamental type" in the working paper, together with a discussion of how making enumerations non-fundamental types would affect that section of the working paper. The overall impact is very small, and the net effect is a slight simplification of the working paper.

Other Clarifications

While considering these changes, I encountered two uses of the term "scalar" which were already incorrect or misleading. These should be corrected editorially independently of whether the other changes are approved.

In the description of `numeric_limits`, there are two references to "scalar type" which surely were intended to be "fundamental type". The editorial changes to correct this problem are listed with the proposals below.

The description of `valarray` uses the word "scalar" to mean "non-`valarray`". If the non-`valarray` parameters can actually have class type, the description is very misleading. The editorial changes to correct this problem are listed with the proposals below.

Proposals

(1) Proposal: Change the definition of “scalar type” to include pointer to member types.

Working paper changes:

-- In 3.9/9, change

> “Arithmetic and enumeration types (3.9.1) and pointer types (3.9.2) are scalar types.”

to

> “Arithmetic and enumeration types (3.9.1), pointer types (3.9.2) and pointer to member types (3.9.2) are scalar types.”

-- In 8.5/5, 12.8/8 and 12.8/13, remove the phrase “or pointer-to-member”, which is now redundant.

(2) Proposal: Make enumeration types “compound”, not “fundamental”.

Working paper changes:

-- Move the description of enumerations from 3.9.1 to 3.9.2.

-- In 8.3.4/2, add “from an enumeration” to the list of types from which an array can be constructed and remove the footnote.

(3) Suggested Editorial Correction Change `numeric_limits` to refer to fundamental types, not scalar types.

Working paper changes:

-- In 18.2.1/3, change “non-scalar” to “non-fundamental”.

-- In 18.2.1.1/1, change “scalar” to “fundamental”.

-- If proposal 2 is not accepted, change 3.9.1/1, 18.2.1/3 and 18.2.1.1/1 to indicate that `numeric_limits` is not specialized on the enumeration types; and change 2.8/5 to indicate that `<limits.h>` does not describe enumeration types.

(4) Suggested Editorial Correction: Change the description of `valarray` to use the term “non-`valarray`” instead of “scalar”.

Working paper changes:

-- In 26.3.1.6/5, 26.3.2.1/5 and 26.3.2.2/5 change “scalar” to “non-`valarray`”.

Appendix A - uses of "scalar"

Here are the uses of the term "scalar" in the working paper, together with a discussion of each use:

Copying via `memcpy`

> 3.9/3 "For any **scalar** type T, if two pointers to T point to distinct T objects `obj1` and `obj2`, if the value of `obj1` is copied into `obj2`, using the memcpy library function, `obj2` shall subsequently hold the same value as `obj1`.".

This is the only real issue. Since pointers and integers can be copied with `memcpy`, and existing implementations represent pointers to members using pointers and integers, it seems reasonable to allow `memcpy` on pointers to members.

Definition

> 3.9/9 "Arithmetic and enumeration types (3.9.1) and pointer types (3.9.2) are **scalar** types."

This is the definition which would have to be changed.

Sequence points

> 5/4 "Between the previous and next sequence point a **scalar** object shall have its stored value modified at most once by the evaluation of an expression."

This is reasonable for pointers to members.

Explicit destructor calls

> 5.2.4/2 "The id-expression shall name a member of that class, except that an imputed destructor can be explicitly invoked for a **scalar** type (12.4)."
>
> 12.4/12 "The notation for explicit call of a destructor can be used for any **scalar** type name. Using the notation for a type that does not have a destructor has no effect."

This is reasonable for pointers to members, and necessary for using them with templates.

Default zero-initialization

> 8.5/5 "To zero-initialize storage for an object of type T mean: ... if T is a **scalar** or pointer-to-member type, the storage is set to the value of 0 (zero) converted to T"

There is no effect except simplifying the text.

Direct vs. copy initialization

8.5/12 "If T is a **scalar** type, then a declaration of the form

```
T x = { a };
```

is equivalent to

```
T x = a;
```

Surely this is already true for pointers to members.

Generated copy constructor and copy assignment

12.8/8 "Each subobject is copied in the manner appropriate to its type: ... if the subobject is of **scalar** or pointer-to-member type, the built-in assignment operator is used."

12.8/13 "Each subobject is assigned in the manner appropriate to its type: ... if the subobject is of **scalar** or pointer-to-member type, the built-in assignment operator is used."

There is no effect except simplifying the text.

`numeric_limits` template

18.2.1 "[The `numeric_limits` template for] non-**scalar** types, such as complex<T> (26.2.1), shall not have specializations." and "The member is_specialized makes it possible to distinguish between **scalar** types, which have specializations, and non-**scalar** types, which do not."

This section uses "fundamental type" and "scalar type" as if they were the same. Clearly the intent is "fundamental type", and the uses of "scalar type" should be corrected.

There is no effect on pointers to members, since `numeric_limits` could not be specialized for all pointer to member types anyway.

`valarray` template

26.3.1.6/5 "Each of these operators applies the indicated operation to each element of the array and the **scalar** argument."

26.3.2.1/5 "Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the **scalar** argument."

26.3.2.2/5 "Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array and the **scalar** argument."

The meaning of "scalar" in this section is also different from the definition in 3.9. The three uses of this term should be changed to some other term, such as "non-array" or "non-`valarray`". Since the meaning of scalar is different here, there is no impact on pointers to members.

iostream formatted input & output

27.6.1.2.1/2 "Some formatted input functions endeavor to obtain the requested input by parsing characters extracted from the input sequence, converting the result to a value of some **scalar** data type, and storing the converted value in an object of that **scalar** data type."

27.6.1.2.1/6 "If the converted data value cannot be represented as a value of the specified **scalar** data type, a scan failure occurs."

27.6.2.4.1/4 "Some formatted output functions endeavor to generate the requested output by converting a value from some **scalar** or NTBS type to text form and inserting the converted text in the output sequence."

The meaning of "scalar" here is not intended to imply that all scalar types can be used. This can be inferred from the existing text, so there is no change needed.

Appendix B - uses of "fundamental type"

`limits.h`

2.8/5 "[*Note*: Certain implementation-defined properties, such as the type of a sizeof (5.3.3) expression, the ranges of fundamental types (3.9.1), and the types of the most basic library functions are defined in the standard headers `<limits>`, `<cstddef>`, and `<new>` (_lib.support_). —*end note*]"

Without the proposed change, this section would have to be edited to exclude enumeration types.

Basic Concepts

3/1 "Finally, this clause presents the fundamental types of the language and lists the ways of constructing compound types from these."

3.9/1 "There are two kinds of types: fundamental types and compound types."

3.9/2 "There is a conceptually infinite number of compound types constructed from the fundamental types in the following ways:"

3.9/4/1 "[*Note*: Fundamental and compound types can be given names by the `typedef` mechanism (7.1.3), and families of types and functions can be specified and named by the template mechanism (14).]"

There is no effect on these uses.

3.9.1/1 "There are several fundamental types. Specializations of the standard template `numeric_limits` (18.2) shall specify the largest and smallest values of each for an implementation."

Without the proposed change, this section would have to be edited to exclude enumeration types

3.9/3 "Each cv-unqualified fundamental type (3.9.1) has three corresponding cv-qualified versions of its type"

There is no effect on this use, but the paragraph is not entirely accurate and needs editorial work.

User-Defined Conversions

5/9 "User-defined conversions of class types to and from fundamental types, pointers, and so on, can be defined"

There is no effect; enumerations are now part of the "so on".

Typedefs

7.1.3/1 "Declarations containing the *decl-specifier* typedef declare identifiers that can be used later for naming fundamental (3.9.1) or compound (3.9.2) types."

There is no effect.

Type Specifiers

7.1.5.2/1 "The *simple-type-specifiers* specify either a previously-declared user-defined type or one of the fundamental types (3.9.1)."

There is no effect, other than making the paragraph slightly more clear. (Enumerations are user-defined types, and the paragraph already implies that they are not fundamental.)

Declarators

8/2 "The specifiers indicate the fundamental type, storage class, or other properties of the objects and functions being declared."

This paragraph already needs work, since it seems to disallow classes and typedefs from being used as type specifiers. With the proposed change, it would also exclude

enumerations so any editorial correction to this paragraph would have to take that into account.

Arrays

8.3.4/2 "An array can be constructed from one of the fundamental types) (except `void`), from a pointer, from a pointer to member, from a class, or from another array."
[*Footnote*: The enumeration types are included in the fundamental types.]

This paragraph would have to explicitly mention enumeration types, and the footnote would no longer be needed.

`numeric_limits`

18.2/1 "Characteristics of implementation-dependent fundamental types (3.9.1)."

18.2.1/1 "The `numeric_limits` component provides a C + + program with information about various properties of the implementation's representation of the fundamental types."

18.2.1/1 "Specializations shall be provided for each fundamental type, both floating point and integer, including bool."

Without this change, these paragraphs would have to be changed to exclude enumeration types.