WG21/N2274 **J16/07-0134** **2007-05-01**

Nick Maclaren
University of Cambridge Computing Service,
New Museums Site, Pembroke Street,
Cambridge CB2 3QH, England.
Email: nmm1@cam.ac.uk
Tel.: +44 1223 334761
Fax: +44 1223 334679

Object Aliasing and Threads

1.0. Introduction

This is another go to try to explain one aspect of the "What is an object, really?" problem that I have seen cause trouble with shared memory parallelism. While this problem is not restricted to PODs (in either the old or new senses), it shows up for non-PODs only for non-portable programs. I shall talk in terms of PODs.

The concept of when objects are distinct can be very complex at the hardware level, but the basic C++ model is that they are distinct if and only if they don't overlap at the byte level. For example:

> *3.9.2: If an object of type T is located at an address A, a pointer of type cv T* whose value is the address A is said to point to that object, regardless of how the value was obtained.*

Unfortunately, C++ imposes extra rules on the accessibility of data, to control when and when objects may not be aliased. For example, the last paragraph of *3.10 Lvalues and rvalues* describes the rules linking lvalue types and data, and the first paragraph of *9.5 Unions* describes a further rule for unions.

In the concurrency discussions, the general assumption has been that the rules apply to the basic memory actions on the built-in scalar types, out of which all other types/classes are constructed. Unfortunately, that is not totally well-defined for reasons described below. The details are mainly a library matter, but the basic rules are a core one.

The problem is that many existing, harmless uncertainties become exposed and hence harmful when concurrency is introduced.

1.1. The Issues

The first issue is that many implementations (perfectly correctly) optimise actions on PODs to handle the whole POD as if it were a single scalar, and do not decompose actions into those on the basic built-in scalars. For example, in the following:

```
struct { int a; int b; } P, Q;
P = Q;
```

there is no particular reason why the copy should not be implemented like:

```
for (int i = 0; i < sizeof(int); ++i)
     { (&P.a)[i] = (&Q.a)[i]; (&P.b)[i] = (&Q.b)[i]; }
```

A second is that it is not always clear when a POD is being handled as a unit, as an aggregate of its basic, built-in scalar types and as an array of `char`, especially with the untyped library actions like `memcpy`.

A third is that pointers to scalars and arrays are usually indistinguishable, and so the length of the POD being addressed depends on the semantics of the operation. That is, of course, the reason that C++ cannot easily provide the Fortran 90 facility to copy one array to another as a unit.

A fourth is that *9.5 Unions* keeps using terms like "at any time". That is not an unambiguous concept once one introduces concurrency.

There is another, extremely arcane, issue to do with machines where the synchronisation and sequencing rules (or even mechanisms) are different between types (typically integer and floating-point). At present, it rarely hits but, when it does, it causes chaos. But I think that a solution to the above problem would reduce this one to manageability, by both requiring and enabling the implementor to hide it from the programmer.

I can witness that there is considerable confusion already, which now arises mostly in the semi-standardised context of C and OpenMP or POSIX threads. I have had a vendor reject a bug report, saying that two accesses from different threads needed to be serialised, despite the fact that they referred to non-overlapping memory. I could read the relevant standards as supporting either the programmer or the implementor, which did not help anyone.

1.2. The Proposal

C++ needs to clarify this area, though I am afraid that I am at a bit of a loss to know how. Here is a proposal, **not** in standardese, and see below for some potential problems with it. It makes the rules for inter-thread aliasing more liberal than those for intra-thread aliasing; I can't see any major problems, but it will confuse some people.

Proposal: *The standard should state that objects are distinct as far as the inter-thread synchronisation and sequencing rules are concerned if and only if they have no bytes in common.*

The main technical alternative is to provide **some** wording that indicates what sections like *3.10 Lvalues and rvalues* and *9.5 Unions* mean in the context of concurrent execution. That is beyond my ability.

2.0. Examples

These may be skipped by anyone who believes my statements of the current situation; they are only an attempt at clarification. There is also the much longer *Objects* message that I posted; while I could put it in a formal mailing, I doubt that it will clarify anything, especially as about half of it does not apply to C++, but only to C99.

2.1. Union and Similar Examples

Consider the following declarations:

```
typedef struct { int a; int b; } P;
typedef struct { int a; int b; int c; } Q;
union { P p; Q q; } z;
```

Now, can each member of the following pairs of statements be executed in two different threads with no ordering between the pair?

```
z.p.a = 1;
z.q.c = 1;
```

or

```
((P *)&z)->a = 1;
((Q *)&z)->c = 1;
```

or

```
(reinterpret_cast<P&>(z)).a = 1;
(reinterpret_cast<Q&>(z)).c = 1;
```

While that sort of code is most definitely regrettable, it is also fairly common – for example, the X Windowing System and Python rely heavily on such practices.In serial code, the first pair may not be used together, but the C and C++ standards are silent about the others. I have seen code that used casts to bypass the *9.5 Unions* restriction, thus turning the first pair into one of the others – but is that conforming?

The question is what constraints there should be on threaded programs. If we take the simple approach, all of the above become defined across threads, but not necessarily within a thread (unless sequenced).

2.2. Pointer Examples

There is the problem of whether a pointer refers to a scalar or an array and, if the latter, how long the array is. My understanding is that C++ avoids that problem by ensuring that the answer is always clear from the context. However, C99 avoids it by not having any language actions that operate on arrays, and this leads to a minor conflict between C++ (e.g. *12.8 paragraph 13*) and C99 (e.g. *6.7.2.1 paragraph 2*). While I can imagine circumstances under which this would matter, I think that it will impact only people who are programming dangerously.

The same does not apply to the C++ library. I believe that this is solidly a library issue, and needs the library to provide some sort of statement of exactly what object (including the size of any array) each argument refers to. At present, the standard is relatively unambiguous only because there is an implied serialisation. Here is a simple example that everyone will agree is defined:

```
double P[100];
memcpy(P,&P[50],10*sizeof(double));
```

But you can change that by small increments into code that everyone will agree is undefined, without having any clear point at which the C++ standard calls a halt. For example, what about the following?

```
struct { int a; int b; } P, Q;
memcpy(&P,&Q.b,sizeof(int));
memcpy(&Q,&P.b,sizeof(int));
```

The first argument is clearly a pointer to a structure but, in this use, does not refer to the structure as a whole. Many programs are solid with such uses, which we can agree is as regrettable as the union and structure examples. But we have to live with it.

As far as I know, there are no such examples in the core language; if I am wrong, they will need addressing.

2.3. The Type-Specific Mechanism Issue

This is another place where C90/C++ and C99 differ slightly. I can witness that it already causes problems in code that uses POSIX threads, and can see no reason why the same should not occur in C++. Where it arises is in an implementation that supports (say) sequential consistency, but not for two actions like the following:

```
double a;
for (int i = 0; i < sizeof(double); ++i)
    ((unsigned char *)&a)[i] = 0;
< memory release >
< memory acquire >
... = a;
```

3.10 Lvalues and rvalues says that it is permitted for serial code, but leaves enough leeway that vendors can (and have) said that it is not permitted for concurrent code. My belief is that, if we have a suitably precise memory model, that will become a plain bug. All that is needed is sufficiently clear rules that both the implementors and developers agree on what is allowed. But, without sufficiently clear rules, that will continue to be a problem.