long long, size_t and compatibility

Introduction

This proposal is NOT about long long as such.

Whether or not introducing long long was essential, unnecessary or harmful, that battle has been lost and won. For good or ill, we have long long defined as a type that holds at least 64 bits, with the naïve assuming that it holds exactly 64 bits.

This proposal is how to alleviate the source code incompatibility introduced by the implicit permission to allow size_t and ptrdiff_t to be longer than unsigned long and long, respectively. See later for why this is serious.

Proposal

One of the following should be done:

1. C++ should state that an implementation must not make size_t and ptrdiff_t to be longer than unsigned long and long, respectively.

2. C++ should require a diagnostic in the following cases:
   - Either of size_t or ptrdiff_t is cast or converted to unsigned long or long, including when it is done via other integer types that are potentially longer than unsigned long or long.
   - When the new rules would cause a different type to passed as an argument of a <stdargs.h> function if size_t or ptrdiff_t were longer than unsigned long and long, respectively.

3. C++ should require such diagnostics only for implementations that choose to make size_t and ptrdiff_t longer than unsigned long and long, respectively.

The first option maintains source compatibility, the second provided the maximum help with diagnosing problems, and the third has least impact.

Justification

Since the very early days of K&R C, there have been exactly 4 notional lengths of built-in integer type. C89 and C++ (3.9.1) say:

There are four signed integer types, designated as signed char, short int, int, long int.

and:

    ptrdiff_t

which is the signed integer type of the result of subtracting two pointers;

    size_t
which is the unsigned integer type of the result of the sizeof operator;

C89 and C++ usual arithmetic conversions (5 paragraph 9, not quoted for brevity) also spell out that unsigned long and long are required to be the longest built-in integer types.

C99 broke that guarantee. On the grounds of not restarting old flame wars, I shall not go into details.

Based on experience and investigations, this is essentially an issue solely for size_t and ptrdiff_t — in theory, it could affect wchar_t, clock_t, time_t and many POSIX types (from pid_t to off_t) but, in practice, it doesn’t. In practice, all other standard integer types are no longer than int, except for ones that have often been implemented as structures or other non-integers (e.g. off_t and time_t) and hence are known to be problematic.

However, a great many clean, portable, conforming (and even strictly conforming) programs include code like the following:

```c
ptrdiff_t length;
printf("%ld\n",(long)length);
or:
#define OFFSET 161
printf("%ld\n",OFFSET+sizeof(double));
or:
#define BLOCK_SIZE 16384
ptrdiff_t offset, block, entry;
ldiv_t result;
result = ldiv((long)offset,(long)BLOCK_SIZE);
block = result.quot;
block = result.rem;
```

Furthermore, programs that wanted to maintain K&R compatibility often used long as a calculation type for indices. In fact, quite a few newer C89 programs do, where the author prefers to use long for all variables rather than flipping between ptrdiff_t and long for calculations.

All of the above are undefined behaviour in C99, in any implementation that chooses to make ptrdiff_t longer than long. Worse, two of them will often not show up on test data (i.e. small sizes), but will when given larger values, and one of them will sometimes appear to work on some implementations with ptrdiff_t longer than long.

### Appendix: The Evidence

This is what I posted to the SC22WG14 reflector. To the best of my knowledge, it is the only hard evidence that has ever been published on this topic.

I took a copy of gcc and hacked it around enough to produce diagnostics for some of the problem cases, where C9X introduces a quiet change over C89 in the area of ‘long’ and ‘long long’. However, this hack has the following properties:

1) It flags only some traps.
2) It produces a large number of false positives.
3) It requires header hacks, and produces broken code.

I then ran it on a range of widely-used and important public-domain

WG21/N1983   J16/06-0053   2006-04-21
codes, taken from the Debian 1.3.1 CD-ROM set. Many of these are
effectively the same codes that are shipped with commercial systems,
and others are relied on heavily by many sites.

Most of the codes used "long" to hold object and file positions, or as
a way of printing an unknown integer type. The ones that I have marked
as "Yes" will almost certainly invoke undefined behaviour if faced with
a C9X compiler where ptrdiff_t is longer than "long", and probably will if
off_t is. The ones that I have marked "Maybe" could well have checks
to prevent this, or were too spaghettified to investigate.

Only 4 had any reference to "long long" whatsoever, and it was in a
single non-default #if'd out section in 3 of them; one of those defined
a symbol that was never referred to, another was solely for Irix 6 file
positions, and the last could trivially have been replaced by double.
The ONLY program that either had any reference to "long long" by
default, or used it seriously, was gcc itself.

<table>
<thead>
<tr>
<th>Loss of data</th>
<th>printf fails</th>
<th>Uses long long</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>bison</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>bash</td>
<td>Maybe</td>
<td>Yes</td>
</tr>
<tr>
<td>cpio</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>csh</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>diff</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>elm</td>
<td>Build process failed</td>
<td>No</td>
</tr>
<tr>
<td>exim</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>fileutils</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>findutils</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>flex</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>gawk</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>gcc</td>
<td>Build process failed</td>
<td>Yes</td>
</tr>
<tr>
<td>gnuplot</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>gzip</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>icon</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>inn</td>
<td>Build process failed</td>
<td>No</td>
</tr>
<tr>
<td>nvi</td>
<td>Maybe</td>
<td>Yes</td>
</tr>
<tr>
<td>pari</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>perl</td>
<td>Build process failed</td>
<td>Effectively not</td>
</tr>
<tr>
<td>sendmail</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>trn</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>wu-ftpd</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>zip</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

We absolutely MUST have some MANDATED migration aids in C9X to detect
at least the worst of these problems. If not, then we need to preserve
C89 as an alternative standard for at least the next 5 years, and I
really don’t want that!