Doc No: X3J16/95-0115
WG21/N0715
Date: 13. June 1995
Project: Programming Language C++
Reply to: Roland.Hartinger@mch.sni.de

Incorporate the long long integral data type in C++

The proposal describes an extension for the C++ programming language to make it possible to use large integral data in Standard conforming programs. Many modern processor architectures offer integrals which are larger than 32 bits (e.g. 64 bit long) and some of todays C and C++ compilers already support those integrals needed by new types of system applications. Therefore, the new type specifier long long *should be introduced in the C++ Standard to allow users to write conforming and portable programs. If the underlying processor does not support integrals larger than 32 bits,* long long *has at least the same size as* long int*.*

Roland Hartinger

Siemens Nixdorf Informationssysteme AG

Munich, Germany

I. Introduction

Some modern software applications written in C and C++ want to use extended integers, e.g. for implementing large file systems, data bases or Object Request Broker systems (e.g. Extended IDL types).

Modern processor architectures have implemented such large integrals (e.g. 64 bit long) and many existing C and C++ implementations have already extended their language with *long long* data types and corresponding literals to support such applications.

One could argue that it might be sufficient just to redefine long int and give it the desired larger size. But an application does not always want to use large integral data and is not always willing to pay the extra amount of memory space and computation time when using long int. They even want to use *long int* and *long long int* in parallel in one program using their different characteristics.

Consequently, the forthcoming C++ Standard should incorporate the *long long* integral data type as a distinct type into the language.

II. Compatibility issues

In the following 2 subsection I shall discuss some of the compatibility issues raised while writing this proposal.

1. Syntax

The following example describes the issue:

```
21LLLL;

// so far parsed as the two tokens:

21L   LLL;

// after the long long extension it will be parsed as:

21LL LL;    // (1)
```

This could result in a change in (1) if `LLL` is a macro substitution.

However, I think this kind of programming ought to be very seldom and it is at least strange programming style and should therefore avoided anyway.

2. Literal representation rules

The proposed changes in section 2.9.1 is based on my assumptions, that it is absolutly necessary that this extension must be compatible to the corresponding rules in ISO-C and the current C++ Working Paper. Only those literals will have type long long int, which previously have been ill-formed.

III. Implemention aspects

This extension does not need any new keywords, syntax or semantic rules. It fits smoothly into the existing language. If it is accepted by the X3J16/WG21 committee, I am going to propose it to the ISO-C language as well.

There are many C and C++ implementions today which already support the long long extension. These compilers (e.g. the EDG front end) have prooved the implementability in practice, so that it does not load the Standard's process and the implementors with additional new riscs in time and undetermined costs.

Furthermore, an implementation has the choice to support just the long long syntax, but give long long the same size as long int. So, the initial effort for implementations is rather low for this proposed extension.

IV. Changes to the Working Paper

The following subsections describe the necessary changes in the current working papers section, to incorporate the long long integral data types and the corresponding literals. I have used the Working Paper wording and have incorporated the extensions; in most cases I have marked the changes with underlines.

2.8 Keywords [lex.key]

The new data type can be declared as a signed or unsigned representation with existing keywords by the following syntax:

long long int$_{opt}$

unsigned long long int$_{opt}$

2.9.1 Integer literals [lex.literal]

Extend the syntax rule as follows:

integer-suffix:

unsigned-suffix long-long-suffix$_{opt}$

long-long-suffix unsigned-suffix$_{opt}$

long-long-suffix: one of

ll LL

2.9.1 Integer literals [lex.icon]

Change the existing §2, beginning at the second sentence, against:

„If it is decimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `long int`, `unsigned long int`, `long long int`, `unsigned long long int`.

If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, `unsigned long long int`.

If it is suffixed by `u` or `U`, its type is the first of these types in which its value can be represented: `unsigned int`, `unsigned long int`, `unsigned long long int`.

If it is suffixed by l or L, its type is the first of these types in which its value can be represented: `long int`, `unsigned long int`, `long long int`, `unsigned long long int`.

If it is suffixed by ul or UL, its type is the first of these types in which its value can be represented: `unsigned long int`, `unsigned long long int`.

If it is suffixed by ll or LL, its type is the first of these types in which its value can be represented: `long long int`, `unsigned long long int`.

If it is suffixed[1] by ull, llu, uLL, LLu, Ull, llU, ULL, or LLU, its type is `unsigned long long int`."

3.9.1 Fundamental types [basic.fundamental]

Change the first sentence in §4 to:

There are five signed integer types: "signed char", "short int", "int", "long int" and „long long int". In this list, each type provides at least as much storage as those preceding it in the list, but the implementation can otherwise make any of them equal in storage size.

Change the first sentence in §5 to:

For each of the signed integer types, there exists a corresponding (but different) unsigned integer type: "unsigned char", "unsigned short int", "unsigned int", "unsigned long int" and "unsigned long long int" each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type [24) ; ...

Change the footnote 27 in §9:

[27)] Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to `int, unsigned int, long, unsigned long, long long,` or `unsigned long long, as specified in 4.5.`

4.5 Integral promotions [conv.prom]

Change the §2 to the following:

An rvalue of type wchar_t (3.9.1) or an enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of the source type: `int, unsigned int, long, unsigned long, long long,` or `unsigned long long`.

5 Expressions [expr]

Add the following sentences to §12:

— If either operand is `unsigned long long,` the other shall be converted to `unsigned long long.`

— Otherwise, if one operand is a `long long int` and the other `unsigned long int`, then if a `long long int` can represent all the values of an `unsigned long int`, the `unsigned long int` shall be converted to a `long long int`; otherwise both operands shall be converted to `unsigned long long int`.

— Otherwise, if one operand is `unsigned int` and the other is `long long int`, then if a `long long int` can represent all the values of an `unsigned int,` the `unsigned int` shall be converted to a `long long int;` otherwise both operands shall be converted to `unsigned long long int`.

— Otherwise, if either operand is `long long,` the other shall be converted to `long long.`

5.2.2 Function call [expr.call]

Change the second sentence in §7 to:

Any argument of enumeration type is converted to `int`, `unsigned,` long, `unsigned long,` `long long` or `unsigned long long` by integral promotion.

5.3.3 Sizeof [expr.sizeof]

Change the note in §1 to:

[*Note:* `sizeof(char)` is `1,` but `sizeof(bool), sizeof(wchar_t)` and `sizeof(long long)` are implementation-defined. [45)] See 1.5 for the definition of byte and 3.9 for the definition of object representation.]

[1] combinations like `LuL` or `Ll` are not allowed.

7.1 Specifiers [decl.spec]

Change the note in §3:

[*Note:* since `signed, unsigned, long, long long`, and `short` by default imply `int`, a *type-name* appearing after one of those specifiers is treated as the name being (re)declared. [*Example:* ...

7.1.5 Type specifiers [decl.type]

Change the following sentence in §2:

— `short` or `long` can be combined with `int`.

to:

— `short`, `long` or `long long` can be combined with `int`.

7.1.5.2 Simple type specifier [decl.type.simple]

Add the following rows to the Table 7:

Specifier(s)	Type
`long long int`	„`long long int`"
`long long`	„`long long int`"
`signed long long int`	„`long long int`"
`signed long long`	„`long long int`"
`unsigned long long int`	„`unsigned long long int`"
`unsigned long long`	„`unsigned long long int`"

17.3.1.1 `Library` contents [lib.contents]

Add to the table 19 in §15 the following overloaded standard functions:

```
abs(long long);
div(long long, long long);
```

18.2.1 Numeric limits [lib.limits]

Add the following specializations to the namespace std in header <limits>:

```
class numeric_limits<long long>;
class numeric_limits<unsigned long long>;
```

18.2.2 C Library [lib.c.limits]

Add the following macros to the `Table 26—Header <climits> synopsis:`

```
LLONG_MAX   LLONG_MIN   ULLONG_MAX
```

22.2.2.1 Template class `num_get` [lib.locale.num.get]

Add to the template class `num_get` the following public `get` member function declarations:

```
iter_type get(iter_type in, iter_type end, ios&,
              const locale&, long long& v) const;
iter_type get(iter_type in, iter_type end, ios&,
              const locale&, unsigned long long& v) const;
```

Add to the template class num_get the following protected do_get function declarations:

```
virtual iter_type do_get(iter_type, iter_type, ios& ,
             const locale&, long long& v) const;
virtual iter_type do_get(iter_type in, iter_type end, ios& ,
             const locale&, unsigned long long& v) const;
```

22.2.2.1.1 num_get members [lib.facet.num.get.members]

Add to the template class num_get the following get member function declarations:

```
iter_type get(iter_type in, iter_type end, ios& str,
             const locale& loc, long long& v) const;
iter_type get(iter_type in, iter_type end, ios& str,
             const locale& loc, unsigned long long& v) const;
```

22.2.2.1.2 num_get virtual functions [lib.facet.num.get.virtual]

Add to the template class num_get the following do_get virtual member function declarations:

```
virtual iter_type do_get(iter_type in, iter_type end, ios& str,
             const locale& loc, long long& val) const;
virtual iter_type do_get(iter_type in, iter_type end, ios& str,
             const locale&, unsigned long long& val) const;
```

22.2.2.2 Template class num_put [lib.locale.num.put]

Add to the template class num_put the following public put member function declarations:

```
iter_type put(iter_type s, ios& f, const locale& loc,
              long long v) const;
iter_type put(iter_type s, ios& f, const locale& loc,
              unsigned long long v) const;
```

Add to the template class num_put the following protected do_put member function declarations:

```
virtual iter_type put(iter_type, ios&, const locale&,
                      long long v) const;
virtual iter_type put(iter_type, ios&, const locale&,
                      unsigned long long v) const;
```

22.2.2.2.1 num_put members [lib.facet.num.put.members]

Add the following non-virtual put member function declarations:

```
iter_type put(iter_type out, ios& str, const locale& loc,
              long long val) const;
iter_type put(iter_type out, ios& str, const locale& loc,
              unsigned long long val) const;
```

22.2.2.2.2 num_put virtual functions [lib.facet.num.put.virtuals]

Add the following virtual do_put member function declarations:

```
virtual iter_type do_put(iter_type out, ios& str, const locale& loc,
                  long long val) const;

virtual iter_type do_put(iter_type out, ios& str, const locale& loc,
                         unsigned long long val) const;
```

27.6.1.1 Template class basic_istream [lib.istream]

Add to the public part of the template class `basic_istream` the following extractor functions:

```
basic_istream<charT,traits>& operator>>(long long& n);

basic_istream<charT,traits>& operator>>(unsigned long long& n);
```

27.6.1.2.1 Common requirements [lib.istream.formatted.reqmts]

Add to §3 the following locale dependent extractors:

```
operator>>(long long& val);

operator>>(unsigned long long& val);
```

27.6.2.1 Template class basic_ostream [lib.ostream]

Add to the public part of the template class `basic_istream` the following inserter functions:

```
basic_ostream<charT,traits>& operator<<(long long n);

basic_ostream<charT,traits>& operator<<(unsigned long long n);
```

27.6.2.4.1 Common requirements [lib.ostream.formatted.reqmts]

Add to §2 the following locale dependent inserters:

```
operator<<(long long val);

operator<<(unsigned long long val);
```

V. Acknowledgements

Thanks to Dominik Straßer and Erwin Unruh for discussing and reviewing the proposal.