Accredited Standards Committee*
X3, INFORMATION PROCESSING SYSTEMS

Doc No: X3J16/95–0187
WG21/N0787
Date: 26 September 1995
Project: Programming Language C++
Reply to: Andrew Koenig
AT&T Bell Laboratories
PO Box 636
600 Mountain Avenue
Room 2C–306
Murray Hill, NJ 07974 USA
ark@research.att.com

Input iterators

Andrew Koenig

Introduction

What should the Standard require from input iterators?

There is no single right answer to this question because input iterators do not follow a single obvious mathematical model the way forward iterators do. Instead, the requirements on input iterators are an attempt to balance utility and ease of implementation.

If utility were the only criterion, there would be no need for input iterators as a separate category, because even for very restricted input iterator types it is possible to create a supertype that meets the forward iterator criteria—provided only that one has enough cache memory to support the way that supertype is actually used.

We will look at each of the operations input iterators might plausibly provide and show examples that argue for or against providing them.

In all the discussion that follows, type `II` (standing for Input Iterator, not roman numeral 2) is assumed to meet the input iterator requirements and objects `i` and `j` are assumed to be of type `II`.

Copy and dereference

It should be obvious that it must be possible to copy an input iterator and dereference the copy. In other words

```
II k = i;   // must be legal
*k;         // must be legal if i was dereferenceable
```

Without these properties, even the simplest algorithms become impossible to write. For example, consider

---

* *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 1250 Eye Street NW, Suite 200, Washington DC 20005

```
template <class II, class X> II find(II begin, II end, const X& x)
{
      while (begin != end && *begin != x)
            ++begin;
      return begin;
}
```

Here the iterator `begin` is copied twice: once on the way in and again on the way out.

However, the input iterator requirements could consistently say that once an input iterator has been copied, there are no further requirements on the original. After all, after calling

```
f(i);
```

it is possible that f has done something like

```
II j = i;
++j;
```

after which `i` is no longer valid. So after calling `f(i)`, it is not possible to use `i` again unless you know that `f` did not invalidate it. The question is whether such knowledge is ever reliable.

Strong arguments can be made in both directions on this issue. For me, however, the following argument is conclusive. Consider this function, which finds the second occurrence of a value in a range if it exists and returns the end of the range otherwise:

```
template<class II, class X>
      II find2nd(II begin, II end, const X& x)
{
      II i = find(begin, end, x);
      if (i == end)
            return end;
      return find(i, end, x);
}
```

This algorithm passes `end` to `find` twice, which means that the first time had better not invalidate it.

When this example came up in reflector discussion, there was a counter-argument that off-the-end iterators had to be a special case to allow for functions like this one. But on reflection, I think that nothing about this function requires that `end` be an off-the-end iterator. It might be some kind of 'bookmark,' which compares equal to an iterator that has read a certain number of records from an input file. It is true that such an iterator cannot be created using only the input iterator requirements, but so what?

If copying an iterator invalided it unless it was an off-the-end iterator, the requirements for `find2nd` would have to say something like: ''If `II` is an input iterator type, the second argument to `find2nd` must be an off-the-end value. If `II` is a forward iterator type or higher, the second argument can be any value.''

Now suppose I write a class that meets the input iterator requirements and that also provides bookmarks. Does it make sense to tell me that I cannot use `find2nd` without also making my class meet all the forward iterator requirements?

So I conclude that it must be possible to copy an input iterator and still use the original, provided that the copy has not had `++` applied to it.

Assignment

The utility argument also convinces me that assignment of input iterators should be permitted. For example, suppose I rewrite `find2nd` as follows:

```
template<class II, class X>
      II find2nd(II begin, II end, const X& x)
{
      begin = find(begin, end, x);
      if (begin == end)
            return end;
      return find(i, end, x);
}
```

It would be uncomfortable to have to explain why this doesn't work and the other `find2nd` does. Even more telling is the following:

```
template<class II, class X>
      int count(II begin, II end, const X& x)
{
      int n = 0;
      while ((begin = find(begin, end, x)) != end) {
            ++begin;
            ++n;
      }
      return n;
}
```

This code relies on the ability to assign to `begin`. Yes, it can be rewritten so as not to use assignment, but it's a pain and the result is definitely less C++ish.

Copy, then dereference the original

Once an input iterator has been incremented, I believe that there should be no further requirements on copies of its old value. That is:

```
II k = i;
*i;         // OK
*k;         // OK
++k;        // OK
*k;         // OK
*i;         // (potentially) undefined
```

and

```
II k = i;
*i;         // OK
*k;         // OK
++i;        // OK
*i;         // OK
*k;         // (potentially) undefined
```

This is an ease-of-implementation issue. The idea is to make it possible to implement a conforming input iterator by a pointer directly into some kind of input buffer. Incrementing an input iterator might cause that buffer to be refilled, which would scribble the behavior of any stale input iterators.

I claim that preserving the result of dereferencing a stale input iterator is not very useful, because of this:

```
II k = i;
*k;         // OK
++i;
*k;         // If this is still OK...
++i;        // then do this some number of times
*k;         // And then what about this?
++k;        // Or this?
```

If you increment `i` many times past `k` and then increment `k` once, where does it point now? If the answer is 'where it would have pointed had you not incremented `i`,' then `II` might as well be a forward iterator. If the answer is 'undefined,' then I claim there is no legitimate use for a single item of lookahead.

Deference after postfix increment

... Or almost none. There is one important loophole:

```
X x = *i++; // This must be OK
```

The notion of using `*i++` to scan an input stream is so fundamental that we must allow it. Yet `i++` increments `i` and returns its old value, which I said could potentially be invalid. So `i++` could potentially be a garbage value. How do we reconcile this?

I think the answer is to require input iterators to support `*i++` even if they do not otherwise require `i++` to yield a useful value. This could be implemented by having `i++` return a value of a 'proxy' type that, when dereferenced, returns the right element.

In effect, implementing `*i++` requires auxiliary storage for a single element, and having requirements on `*i++` but not on `i++` leaves open the possibility of that element being an iterator or a data value.

Equality

I think the input iterator requirements should say as little about equality as possible. In particular, once an iterator has been invalidated, I think the requirements should say nothing about it.

On the other hand, I do think an iterator should be required to be equal to a copy of itself, because that allows one to express a null range by passing any valid iterator as the beginning and the end of the range. So for example:

```
II j = i;
i == j;          // should be required to be true
++i;
i == j;          // undefined
```

So far I see no argument that the requirements should be stronger than that. In particular, I do not think `==` should be required to be an equivalence relation except within the domain where it is required to be defined at all!

What should we do?

I think we should change the input iterator requirements to match the descriptions above. Here is a first cut at such a proposal.

Operation	Type	Semantics, pre/post consitions
`X(a)`		`X`
`X u(a);`		
`X u = a;`		post: `u` is a copy of `a`
`u = a;`	`X&`	post: `u` is a copy of `a`
`a==b`	convertible to `bool`	if `a` is a copy of `b`, `a==b` `==` is an equivalence relation over its domain. If `a` is a valid iterator, then `a==a`.
`a!=b`	convertible to `bool`	`a!=b` is equivalent to `!(a==b)` over the domain of `==`
`*a`	`T`	pre: `a` is dereferenceable `(a, b)` in the domain of `==` and `a == b` implies `*a == *b`
`++r`	`X&`	pre: `r` is deferenceable post: `r` is dereferenceable or `r` is past the end; there are no further requirements on the values of any copies of `r` except that they can be safely destroyed. After executing `++r`, copies of (the previous) `r` are not required to be in the domain of `==`.
`(void)r++`		equivalent to `(void)++r`

`*r++`	`T`	`{ T tmp = *r;`
		`++r;`
		`return tmp; }`

[Note: there are no requirements on the type or value of `r++` beyond the requirement that `*r++` work appropriately. In particular, `a == b` does not imply that `++a == ++b`.]