CoCoALib - Design #1787

Iterator design: compatible with C++STL? Advancing beyond end?

07 Mar 2024 19:40 - John Abbott

Status:	In Progress	**Start date:**	07 Mar 2024
Priority:	Normal	**Due date:**	
Assignee:	John Abbott	**% Done:**	30%
Category:	Tidying	**Estimated time:**	0.00 hour
Target version:	CoCoALib-0.99880	**Spent time:**	1.10 hour

Description

Two (unrelated) matters regarding iterators in CoCoALib:

1. if we make them compatible with STL iterators then we can use the "new" for syntax (e.g. for (auto f:L))
2. what happens if someone tries to advance beyond the end? (error or not error or maybe error?)

Related issues:

Related to CoCoALib - Feature #839: SparsePolyIter: make more compatible with...	**In Progress**	**13 Jan 2016**
Related to CoCoALib - Feature #1692: Suggestion: Add "JumpTo" function to pri...	**In Progress**	**05 Aug 2022**
Related to CoCoALib - Feature #379: Iter for subsets/tuples	**Closed**	**19 Jun 2013**
Related to CoCoALib - Design #1572: Use noexcept	**Resolved**	**29 Jan 2021**

History

#1 - 07 Mar 2024 19:51 - John Abbott

I think it would be nice to be able to use CoCoALib iterators with the "new" for loop syntax. I note also that C++20 introduced "ranges": something like a pair of iterators (begin, end). We should study this, and consider whether using them would be a good way of adapting CoCoALib iterators to more general C++ use. An alternative is to define an end iterator for each iterator type; or maybe a "universal" end object which may be compared to any CoCoALib iterators -- the comparison would effectively call IsEnded. I admit that it does "feel strange" having a single "end iterator object" for all iterators -- quite unlike the way C++ iterators work!

The other point is what should happen if a user tries to advance a CoCoALib iterator which is already ended? I think there are two likely options: throw an error, or do nothing. The "do nothing" option has the extra risk of possibly allowing unexpected infinite loops; the "throw an error" option is probably more user-friendly. Either way, we must check through every implementation and ensure that it behaves according to design -- I suspect that the current situation is a messy mixture of behaviours.

#2 - 07 Mar 2024 19:51 - John Abbott

- Related to Feature #839: SparsePolyIter: make more compatible with STL added

#3 - 07 Mar 2024 19:52 - John Abbott

- Related to Feature #1692: Suggestion: Add "JumpTo" function to prime iterators added

#4 - 07 Mar 2024 19:52 - John Abbott

- Related to Feature #379: Iter for subsets/tuples added

#5 - 08 Mar 2024 18:26 - Anna Maria Bigatti

- Description updated

#6 - 10 Mar 2024 15:49 - John Abbott

- Status changed from New to In Progress

- Assignee set to John Abbott

- % Done changed from 0 to 10

Here is a list of the iterators in CoCoALib:

- **combinatorics.C** SubsetIter, TupleIter
- **SparsePoly** SparsePolyIter -- may need redesigning!!
- **NumTheory-ContFrac** continued fractions
- **NumTheory-prime.C** & **NumTheory-PrimeModSeq.C** several prime iterators
- **random** several random number generators
- *SparsePolyOps-Graeffe*

Note that some iters are never-ending!

#7 - 22 Apr 2024 21:22 - John Abbott

- Related to Design #1572: Use noexcept added

#8 - 22 Apr 2024 21:24 - John Abbott

- % Done changed from 10 to 20

If iterator operators can throw then they cannot be noexcept (obviously!). See issue #1572
Not sure how important this is; but we should bear it in mind when considering this issue!

#9 - 23 Apr 2024 22:26 - John Abbott

- % Done changed from 20 to 30

A quick look on StackOverflow found a discussion which claims that the C++ standard says that advancing an iterator beyond the one-past-the-last position is *undefined behaviour* (anything could happen, but it'll probably crash sooner or later).

So for iterators in CoCoALib we could:

- **(A)** leave unspecified the behaviour of advancing an ended iterator (similar to what C++ does)
- **(B)** in every call of operator++ check whether it is ended, and throw if so -- safer, but how much overhead?

My current preference is for **(B)**, but are there cases where the overhead could be significant? Maybe prime-sequence iterators?