10 External sorting

Summary: *External sorting is needed if a file cannot be sorted in memory because available memory is too small or the file is too large, and so mass storage must be used as a medium for sorting. The elements of the STL are used to construct an iterator for sorted subsequences which is used for external sorting. A priority queue can accelerate the sorting process.*

To start with, the following questions should be answered to establish whether external sorting can be avoided:

- Is the entire available RAM used as virtual memory without having to swap memory pages?
- Can keys and an index file be used? For example, an address file could be sorted by using only the names for sorting. Then, the index file contains only the sorted names and, for each name, a pointer to the location of the complete address file, where all other information, such as street and town, can be found.

Copy processes in mass storage are very expensive compared to copy processes in RAM. When memory access takes 50 nanoseconds, and hard disk access 10 milliseconds, then the mass storage is slower by a factor of 200 000, if no buffer is used. When all else fails, it could be helpful to divide the problem into smaller subproblems:

1. The large file of N elements is split into n small files of approximately equal size, where n is chosen in such a way that the small file fits into memory.
2. All small files are sorted separately.
3. The sorted files are merged back into one large file. Section 5.5.4 describes how two sorted subsequences are merged into one single sorted sequence.

10.1 External sorting by merging

Step 3 above hardly consumes any memory because only n elements are read and compared. However, the operating system creates a buffer for each open file, which can amount to a considerable quantity of memory. Frequently, the maximum number of open files allowed is also insufficient for this purpose.

Therefore, the process is modified: the large file F is only split into two temporary auxiliary files t_1 and t_2 which are again put together in a large file with a higher degree of sorting. The files F, F', and so on are the same; they are reused. The same applies to t_1 and t_2. Therefore, someone with a little foresight creates a backup copy of F.

This process is repeated with the new file until sorting is achieved (see Figure 10.1). Thus, you only need a total of three files. You could, however, take more than two files for splitting. The only important point is that the temporary files contain sorted subsequences which are merged into each other. A sorted subsequence is also called *run*.

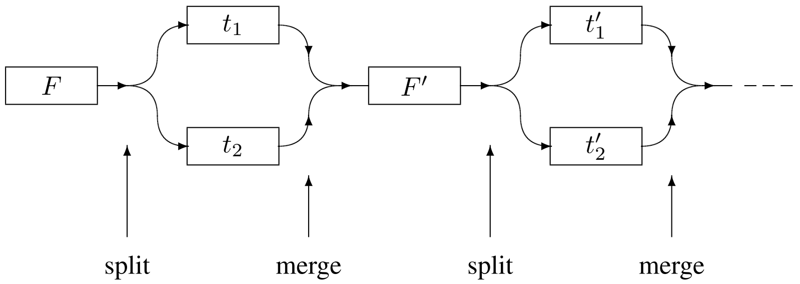

Figure 10.1: External sorting with two runs shown.

As an example, an unsorted sequence of 17 numbers is to be sorted in ascending order into a file. The numbers are:

F : 13 44 7 3 3 9 99 37 61 71 2 6 8 11 14 15 1

This sequence is split into the auxiliary files in such a way that sorted subsequences are maintained. These are shown by way of square brackets:

F : [13 44] [7] [3 3 9 99] [37 61 71] [2 6 8 11 14 15] [1]

Splitting yields:

t_1 : [13 44] [3 3 9 99] [2 6 8 11 14 15]
t_2 : [7] [37 61 71] [1]

The first two subsequences of t_2 can be considered as *one* sorted subsequence:

F : [13 44] [7] [3 3 9 99] [37 61 71] [2 6 8 11 14 15] [1]
t_1 : [13 44] [3 3 9 99] [2 6 8 11 14 15]
t_2 : [7 37 61 71] [1]

Now, the subsequences of the auxiliary files are merged, resulting in the new file F. Merging is carried out in the sense of Section 5.5.4: when one subsequence is exhausted, the remainder of the other subsequence is copied.

merge:
F : [7 13 37 44 61 71] [1 3 3 9 99] [2 6 8 11 14 15]

Further split and merge operations yield:

split:
t_1 : [7 13 37 44 61 71] [2 6 8 11 14 15]
t_2 : [1 3 3 9 99]
merge:
F : [1 3 3 7 9 13 37 44 61 71 99] [2 6 8 11 14 15]
split:
t_1 : [1 3 3 7 9 13 37 44 61 71 99]
t_2 : [2 6 8 11 14 15]
merge:
F : [1 2 3 3 6 7 8 9 11 13 14 15 37 44 61 71 99]

Thus, only three runs with one split and one merge process each are needed. A closer analysis shows that for a file of N elements, a total of about $\log_2 N - 1$ runs is needed. Each run means N copy processes (read + write), so that the total cost is $O(N \log N)$. Later, we will see how we can accelerate this process. Those who find the description too brief should refer to the 'essential' Wirth (1979).

Thus, we have three files, which can also be magnetic tapes, and two passes, namely splitting and merging. Therefore, the method is called 3-way 2-pass sort-merge. When we talk about merging and tapes, it is understood that only sequential access to individual elements is possible. An algorithm for external sorting must take this into account.

The following `main()` program calls a function for external sorting. The file is arbitrarily called *random.dat* and contains numbers of type `long`.

```
// k10/extsort.cpp   Sorting of a large file
#include"extsort.h"          // see below
#include<functional>         // greater<>, less<>
using namespace std;

int main() {
    // less<long> Comparison;      // descending
    std::greater<long> Comparison;              // ascending
    std::istream_iterator<long> suitable_iterator;

    std::cout << externalSorting(
                  suitable_iterator,   // type of file
                  "random.dat",        // file name
                  "\n",                // separator
                  Comparison)          // sorting criterion
              << " sorting runs" << std::endl;
}
```

The function returns the number of necessary runs. Since no information on the type of elements can be derived from the file name, a suitable iterator is passed whose type contains the necessary information. The separator string is inserted between two elements which are written to a temporary file, because this example uses the >> operator for input and the << operator for output. The comparison object determines the sorting criterion. The components needed for this algorithm are described individually.

One important component is an iterator that works on a stream and recognizes subsequences. This iterator will be called `SubsequenceIterator`. It inherits from the class `istream_iterator` which is described on page 38. The subsequence iterator behaves in the same way as an `istream_iterator`, but in addition determines whether the elements of the stream are sorted according to the sorting criterion `comp`. This requires a comparison between a read object and the previous one which here is a private variable named `previousValue`.

```
// Template classes and functions for sorting of large files
// k10/extsort.h
#ifndef EXTSORT_H
#define EXTSORT_H
#include<fstream>
#include<algorithm>
#include<iterator>

template<class T, class Compare>
class SubsequenceIterator : public std::istream_iterator<T>
{
   public:
     typedef T value_type;   // public type

     SubsequenceIterator()
     : comp(Compare()) {
     }

     SubsequenceIterator(std::istream& is, const Compare& c)
     : std::istream_iterator<T>(is), comp(c), sorted_(true),
       previousValue(std::istream_iterator<T>::operator*()) {
     }

     /*The private attribute previousValue can be initialized with value, because
       the initialization of the base class subobject has already read a value. The follow-
       ing ++ operators now ensure that the end of a sorted subsequence is recognized by
       setting the private variable sorted_. A subsequence is in any case also closed
       when the stream is terminated. This is checked by comparing the subsequence
       iterator (i.e. *this) to an end-iterator which is generated by the default construc-
       tor.
```

```
        It is important to write !comp(previousValue, value) and not
        comp(value, previousValue). The second notation would erroneously
        already signal the end of a subsequence when two equal elements follow each
        other. You can easily imagine this by assuming, for example, Compare ==
        less<int>.
     */

     SubsequenceIterator& operator++() {
          std::istream_iterator<T>::operator++();
          const T& value
                   = std::istream_iterator<T>::operator*();
          sorted_ = !comp(previousValue, value)   // right order
                    // end not yet reached?
                 && *this != SubsequenceIterator<T, Compare>();
          previousValue = value;
          return *this;
      }

      SubsequenceIterator operator++(int) {
          SubsequenceIterator tmp = *this;
          operator++();
          return tmp;
      }

      bool sorted() const { return sorted_;}

      /*When the end of a subsequence is recognized, the internal flag for this can be reset
        with nextSubsequence() to process the next subsequence:
      */
      void nextSubsequence() {
         sorted_ = *this != SubsequenceIterator<T, Compare>();
      }

      Compare Compareobject() const { return comp;}

      /*Compareobject() supplies a copy of the internal comp object. In addition to
        the inherited variables, the following ones are needed:
      */
    private:
      Compare comp;
      bool sorted_;
      T previousValue;
};
```

Next, the function `externalSorting()` is described, which constitutes the user interface in `main()`. This function determines the type of the values by means of the `iterator_traits`-class.

```
template<class IstreamIterator, class Compare>
int externalSorting(IstreamIterator& InputIterator,
                  const char *SortFile,
                  const char *Separator,
                  const Compare& comp) {
    typedef typename std::iterator_traits<IstreamIterator>
                                   ::value_type value_type;
    bool sorted = false;
    // arbitrary names for intermediate files
    const char *TempFile1 = "esort001.tmp",
               *TempFile2 = "esort002.tmp";

    int Run = 0;          // number of split/merge-runs
    do {
       std::ifstream Inputfile(SortFile);
       SubsequenceIterator<value_type, Compare>
                       FileIterator(Inputfile, comp);

       /*The file to be sorted must exist. A suitable subsequence iterator for reading
         is passed to the function split() which writes sorted subsequences of the
         main file F, as it was called earlier, into the two auxiliary files t1 and t2.
       */
       split(FileIterator, TempFile1, TempFile2, sorted);
       Inputfile.close();

       /*During this process, split() determines whether F is already sorted. Only
         if this is not the case are further steps necessary. These steps consist in gener-
         ating subsequence iterators for the function mergeSubsequences() and
         opening the output file F'. Then, the subsequences are merged.
       */
       if(!sorted) {
          // prepare for merging
          std::ifstream Source1(TempFile1);
          std::ifstream Source2(TempFile2);

        SubsequenceIterator<value_type,Compare>
                                          I1(Source1,comp),
                                          I2(Source2,comp),
                                          End;

        // open SortFile for writing
        std::ofstream Output(SortFile);
        std::ostream_iterator<value_type>
              Result(Outputfile, Separator);

          mergeSubsequences(I1, End, I2, End, Result, comp);
          ++Run;
       }
    } while(!sorted);
    return Run;
}
```

The function `mergeSubsequences()` has the same interface as the standard function `merge()` (see page 130). `merge()` cannot be used because `merge()` extracts one element at a time via the input iterators according to `comp`, but ignores the subsequence structure.

```
// SubSeqIterator is a placeholder for the data type of a subsequence-iterator
template<class SubSeqIterator>
void split(SubSeqIterator& InputIterator,
              const char *Filename1,
              const char *Filename2,
              bool& sorted) {
    std::ofstream Target1(Filename1);
    std::ofstream Target2(Filename2);
    typedef typename SubSeqIterator::value_type value_type;

    std::ostream_iterator<value_type> Output1(Target1, "\n");
    std::ostream_iterator<value_type> Output2(Target2, "\n");
    SubSeqIterator End;

    /*The functioning is quite simple: as long as the input stream supplies a sorted sub-
      sequence, all data is written to an output stream. Once the end of a sorted subse-
      quence is reached, flipflop is used to switch to the other output stream. In or-
      der to save the caller unnecessary work, the variable sorted remembers whether
      there has been any violation of the sorting order in the input stream.
    */
    sorted = true;
    bool flipflop = true;
    while(InputIterator != End) {
       while(InputIterator.sorted())
          if(flipflop) *Output1++ =  *InputIterator++;
          else         *Output2++ =  *InputIterator++;

       if(InputIterator != End) {
          sorted = false;
          flipflop = !flipflop;
          InputIterator.nextSubsequence();
       }
    }
}

/*After splitting a file into two temporary auxiliary files, the file is rebuilt on a 'higher
  sorting level' by merging the auxiliary files.
*/
template <class SubsequenceIterator, class OutputIterator,
          class Compare>
void mergeSubsequences(SubsequenceIterator first1,
                       SubsequenceIterator last1,
                       SubsequenceIterator first2,
                       SubsequenceIterator last2,
                           OutputIterator result,
                           Compare& comp) {

    // as long as both the auxiliary files are not exhausted
    while (first1 != last1 && first2 != last2) {
        // merge sorted subsequences
        while(first1.sorted() && first2.sorted())
             if (comp(*first1, *first2))
                 *result++ = *first2++;
             else
                 *result++ = *first1++;

        // At this point, (at least) one of the subsequences is terminated.
        // Now copy the rest of the other subsequence:
        while(first1.sorted()) *result++ = *first1++;
        while(first2.sorted()) *result++ = *first2++;

        // Process the next subsequence in both auxiliary files,
        // provided there is one:
        first1.nextSubsequence();
        first2.nextSubsequence();
    }

    // At least one of the temporary files is exhausted.
    // Copy the rest of the other one:
    std::copy(first1, last1, result);
    std::copy(first2, last2, result);
}
```

10.2 External sorting with accelerator

External sorting is designed only for sorting processes where the internal memory of the computer is not sufficient. However, (almost) no memory was used in the above program. The best solution for external sorting is to employ as much internal memory as possible.

An ideal tool for this purpose is the priority queue presented in Section 4.3. It has the property of putting all incoming elements into the correct position, so that when one element is removed, the one with the highest priority according to the sorting criterion is immediately available, for example, the greatest element.

If the priority queue can take N_p elements, then for all input files F with N_p or less elements, only one sorting run is needed. For larger input files, the priority queue allows longer sorted subsequences, so that fewer runs are needed. It is evident that the effect of a priority queue diminishes when the subsequences to be processed are longer than the size of the priority queue. For this reason, the effect of a priority queue is that in the first run, subsequences of a length $\geq N_p$ are already generated, thus saving $(\log_2 N_p - 1)$ runs. At least one run is needed.

The complexity of external sorting does not change when a priority queue is employed. Since, however, copy operations to mass storage are time-consuming, the saving of constant ($\log_2 N_p - 1$) runs is very desirable.

When placing and using the priority queue in the data flow, care must be taken not to use it directly as a sorting filter. The reason for this is that the initial fast generation of long subsequences would become impossible. ***tip***

Let us assume that the number of elements in a file substantially exceeds N_p and that the sorting criterion is to generate a descending sequence, that is, the greatest element is removed from the priority queue. This removal frees a place, and the next element is inserted into the priority queue. This element, however, can be greater than the element just removed, so that the subsequence of removed elements is immediately terminated.

Figure 10.2 shows that to achieve the longest possible subsequences, the priority queue is used inside the splitting process.

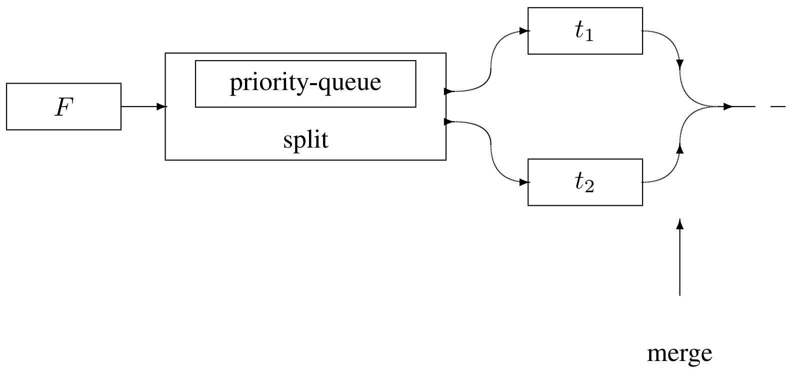

Figure 10.2: External sorting with priority queue.

The decisive factor is that the read elements are not simply passed sorted. Instead, reading of an element greater than the one that stands at the top of the priority queue must lead to the whole priority queue being emptied. Only then can the new element be inserted. As can be seen from the figure, this involves the `split()` function whose modified variation is shown as a conclusion. `#include<algorithm>` can now be omitted, since `copy()` is no longer used. Instead,

```
#include<vector>
#include<queue>
```

are required if the priority queue is to be implemented with a vector. Since the priority queue must know not only the data type of the elements, but also the sorting criterion, the function determines the necessary types from the type of the passed subsequence iterator.

In the example below (function `split()`), the size of the priority queue is specified as 30 000: depending on the computer type, memory size, and operating system, it should, on the one hand, be set as large as possible; on the other hand, it should still be small enough not to need memory swapping to hard disk.

There is no member function `capacity()` for the priority queue of the STL, which would return the capacity of the underlying container. Unfortunately, such a function is not easy to write, because it strongly depends on the operating system.

How much space can be allocated to the program depends on the current usage of the computer by other users and programs, and can therefore only be determined at a given time. Information on the amount of available memory can be given only by the operating system. Therefore, it is best to allocate the program a guaranteed amount of memory at the call.

```
template<class SubSeqIterator>
void split(SubSeqIterator& InputIterator,
           const char *Filename1,
           const char *Filename2,
           bool& sorted) {
    typedef typename SubSeqIterator::value_type value_type;
    typedef typename SubSeqIterator::compare_type Compare;

    const size_t maxSize = 30000; // maximize, see text

    // The size of the priority queue is dynamically increased
    // up to the given limit (see below)
    std::priority_queue<value_type,
                        std::vector<value_type>,
                        Compare>
                 PQ(InputIterator.Compareobject());

    std::ofstream Target1(Filename1);
    std::ofstream Target2(Filename2);
    std::ostream_iterator<value_type> Output1(Target1, "\n");
    std::ostream_iterator<value_type> Output2(Target2, "\n");
    SubSeqIterator End;

    sorted = true;
    bool flipflop = true;    // for switching the output

    while(InputIterator != End) {
        // fill priority queue
        while(InputIterator != End && PQ.size() < maxSize) {
           if(!InputIterator.sorted())
              sorted = false;
           PQ.push(*InputIterator++);
        }

        while(!PQ.empty()) {
            // Write to output files. Selection of file
            // by way of the variable flipflop
            if(flipflop) *Output1++ =  PQ.top();
            else         *Output2++ =  PQ.top();

            // create space and fill if needed
            PQ.pop();
            if(InputIterator != End) {
              if(!InputIterator.sorted())
                  sorted = false;

              // The next element is inserted only if it does not
              // violate the subsequence ordering.
              if(!InputIterator.Compareobject()
                                (PQ.top(), *InputIterator))
                 PQ.push(*InputIterator++);
            }
        }

        // The priority queue is now empty; the sorted sequence
        // output is terminated. For outputting the next sorted
        // sequence we switch to the next channel.
        flipflop = !flipflop;
    }
}
```

A final hint: the last run generates a completely sorted file. This is, however, determined only by the following split, where one of the temporary files is empty and the other one is identical to the result file. The above algorithm could be optimized, so that the last split is no longer needed. For this, it would be necessary to determine during merging whether the result is sorted. One method of achieving this is to construct a more 'intelligent' output iterator `result` which determines this information.

tip