Enhancing STL (Standard Template Library) with a file container

Senior lecturer, PhD. Marian DÂRDALA, professor, PhD. Ion SMEUREANU,
assistant lecturer Adriana REVEIU
Economy Informatics Department, A.S.E. Bucharest

Standard Template Library (STL) implements the work with dynamic data structures by defining a type of class template called data collections or containers. The universality of STL is ensured because the type of the useful information from the data structure is a generic one and by defining of some independent functions which implement generally valid algorithms for data collections. These functions treat the elements of a container unitary, through the concept of iterator. An iterator is an object defined, connected to a container and used to refer the elements from the container to which it is related. Starting from these elements which belong to STL, the paper intends to implement the file using the same model in order to be able later on to relate this to everything that already exists in STL.
***Keywords**: container, iterator, algorithm, template, file, file organization, file access*

Defining the file as a container

The file, as a data structure, is a collection of elements called records or articles. While the data structures in STL were defined in the internal memory, the file is a structure belonging to the external memory.

For the file structure there are already object-oriented implementation, for example the ***fstream*** classes from C++ library or ***CFile*** class in MFC. There are two main elements which are taken into account when a class implements a file: the type of access (read, write or both) and the way in which the input and the output are dealt with, that is the size of the elements being transferred (byte, word, string, record) and the way in which the data are formatted.

In this paper we extend the way in which a file is implemented by adding another perspective: the way in which the files are organized (sequential, direct, indexed) and their specific type of access. On the other hand we intend to ensure the universality of the implementation following the class template model, keeping the compatibility with the STL resources.

The file is implemented as a class template, the generic type being type of the record. The main class (*file*) implements the file with sequential access and includes an *iterator* class. In order to implement the other two types of file organization direct and indexed there have been derived from the template class file, two more templates (*file_d* and *file_i*) as shown in figure 1.

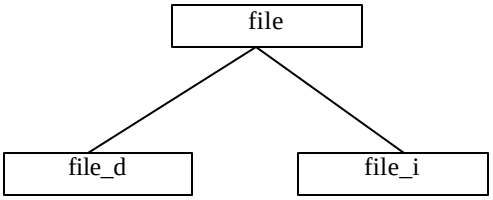

Fig. 1. Classes hierarchy for implement the file structure

These derived templates are built as two adaptive containers which overload the methods of inserting and consulting according to the type of access. The template class for implementing the indexed-organized file has two generic types: one for the record type and the second for the key type.

Sequential-organized file

Sequential-organized file has been implemented using the class *file* which includes the

class *iterator*. The constructor of the file class opens the file and the destructor closes it.

```
#define WRITE 0
#define READ 1
template <typename REC>
class file
{protected:
      FILE *pf;
      int md,eof;
      REC art, art1;
      void copy() { memcpy(&art1,&art,sizeof(REC)); }
      void rewrite();
public:
      class iterator
      {     friend file;
            file *pfis;
            int sf;
      public:
            iterator(int p=0, file *prf=NULL ) : pfis(prf),sf(p) { }
            iterator(file *prf):pfis(prf) { }
            REC& operator*() { return pfis->art;}
            void operator++()
                  {     pfis->rewrite();
                        pfis->eof=fread(&pfis->art,sizeof(REC),1,pfis->pf);
                        if(pfis->eof) pfis->copy();
                  }
            int operator!=(iterator& i) { return pfis->eof!=i.sf; }
      };
      friend iterator;
      file(char *,int=WRITE);
      void push_back(REC );
      iterator begin();
      iterator end() { return iterator(0,this); }
      ~file() { rewrite(); fclose(pf); }
};
```

The methods of this class are:

- *push_back* – writes a record in the file;
- *begin* – returns the iterator referring the first record;
- *end* – returns the iterator which refers the end of the file.

Like in STL in the *iterator* class, the following operators are overloaded:

- ++ to move to the next record;
- != to compare two iterators;
- * to get the record referred by an iterator.

In the *iterator* class has been defined two constructors which make the link with the file for which the iterator has been built. The constructor:

```
iterator(int p=0, file *prf=NULL ) : pfis(prf),sf(p) { }
```

plays the role of the implicit constructor and builds the iterator which refers the end of the file (sf = 0).

An important issue was being able to use the same file iterator both for inputs and outputs, which is to be used both in expressions like:

```
record = * iterator;
```

and like:

```
*iterator = record;
```

The solution of this problem was to keeping the current record, referred by the iterator, in two buffers (*art* and *art1*). You can change a record only in the *art* variable, and another operation on file leads to the rewriting of the current record only if there is a difference between the two buffers (*art* and *art1*). To implement this mechanism there have been defined two functions in the protected section of the *file* class:

- *copy* – to copy the current record, if valid, from art to *art1* ;
- *rewrite* – which rewrites the current record, if it has been modified.

The updating of the current record is possible because the method which overloads the *operator, from the *iterator* class, returns a reference to the record (to the *art* variable) not the value of record.

Files with direct and indexed access

In order to implement this type of access there have been defined two templates using the same principle as for the adaptive type of containers from STL, that is containers which have as purpose to redefine some operations already defined in the parent container.

The *file_d* class for implementing the direct organized file:

```
template <typename REC>
class file_d : public file<REC>
{public:
  file_d(char *s,int mod=WRITE) :
        file<REC> (s,mod) { }
    iterator find(unsigned );
    void push_at(unsigned, REC);};
```

defines the methods:

- *push_at* – to write a record at a specified position in the file (the first position is 0);
- *find* – to receive the position of the record that is supposed to be read from the file and returns an iterator which refers that record or the iterator which marks the end of the file.

If we wish to modify the record at the *i* position, in case it exists, we will write the following sequence:

```
*iterator = record;
```

The *file_i* class for implementing the indexed file:

```
template <typename REC, typename KY>
class file_i : public file<REC>, public index<KY>
{     public:
            file_i(char *s,char *nfi, int mod=WRITE) : file<REC>(s,mod)
            {     set_nume_f(nfi);
                  if(md==READ) restaurare();
            }
            iterator find(KY );
            void push(KY, REC);
};
```

defines the following methods:

- *push* – which inserts a record and receives as input parameters the value of the key and the record itself;
- *find* – which receives the value of the key and returns an iterator that refers the record with that key, if it exists, or else the iterator which refers the end of the file.

In order to manage the indexed access there has been built a template class two data type-parameterized (one type for the key - KY and another for the record itself - REC). In this paper we do not intend to give a full implementation of the index-structure itself but only to present the interface requirements of such a class. Methods like these are needed:

- of saving/restoring of the index in/from the file;
- of inserting a key in the index structure by giving the offset of the record;
- of searching for a key in the index structure and returning the offset, in case the key actually exists in the index structure.

The cooperation of the file container with other elements from the STL

We have said that we implement the file container using the STL library model in order to be able to integrate it with other elements defined in the STL library. We consider a file with *int* types records:

```
file<int> fi("fis_i.dat",READ);
file<int>::iterator it;
```

- Going through the file by using the iterator for type all records:

```
for( it=fi.begin(); it!=fi.end(); it++)
cout<<*it<<endl;
```

- Updating the files with the same algorithm, using the function *for_each*:

```
for_each ( fi.begin(), fi.end(), patrat );
```

where, *patrat* is an independent function:

```
void patrat(int& k) {      k *= k; }
```

- Displaying the records by using the predefined iterator *ostream_iterator* and the *copy* function:

```
ostream_iterator<int> os_it(cout," ");
copy(fi.begin(), fi.end(), os_it);
```

- Searching for a record in the file using the *find* function:

```
it = find( fi.begin(), fi.end(), val); // unde val este
                        //valoarea de cautat in fisier
if ( it != fi.end() ) cout<<"\n S-a gasit el "<<*it;
                    else cout<<"\n El inexistent!!";
```

Conclusions

The example presented above illustrates the flexibility and extensibility of the STL; the definition of new containers which follow the implementation principles of the STL ensures that the code can be reused and by this the achievement of higher performances when programming the applications. In this way there can be obtained new structures which make use of the STL resources, for example the implementation of graphs by using *list* container.

References

- Deitel, H.M., Deitel, P.J., *C++ How to program*, Prentice Hall Inc., New Jersey, 2001.
- Eckel, B., *Thinking in C++*, Prentice Hall Inc., New Jersey, 1999.
- Prata, S., *Manual de programare în C++*, Ed. Teora, Bucuresti, 2001.
- Smeureanu, I., Dârdala, M., *Programarea orientata obiect în limbajul C++*, Ed. CISON, Bucuresti, 2002.