Reengineering of C/C++ Programs using Generic Components

Georg Trausmuth and Roland Knor
Distributed Systems Group
Technical University of Vienna
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{g.trausmuth,r.knor}@infosys.tuwien.ac.at

Abstract

Complex data structures such as lists and trees are difficult to implement in C in a way that programmers have little or no difficulties to maintain and reuse the source code. Thus, converting such data structures into instances of generic C++ components to increase the maintainability of the code is a reasonable task. Replacing data structures with instances of generic data structures is a non-trivial issue; handling dependencies accordingly causes even more effort in the conversion. This paper points out the issues of replacing data structures with instances of generic components provided by libraries. We briefly describe the transformation process, related issues and give an outlook on future activities in this area. This paper shows the benefits of generic source code components for reengineering and maintenance activities.

1 Introduction

The ideas in this paper originate in the attempt of converting an application[1] from C to C++ with emphasize on applying generic components as provided by STL[Stepanov95]. The motivation was to reduce the complexity and simultaneously increase the maintainability of the source code by this transformation. Data structures like the following were the starting point for our work:

```
struct aclEntry {
    int aclTarget;
    int aclSubject;
    int aclResources;
    // ...

    struct aclEntry *reserved;
    struct aclEntry *next;
};
```

Similar data structures that can also be replaced by their STL equivalent, had to be found, transformed and syntactically adjusted. Data structures like

[1]An SNMP-library, which includes the main routines to run the Simple Network Management Protocol.

the one above were found in a number of modules. The code looked almost the same except for some adaptations to store different attributes. The reuse of the code fragments was obviously based on the copy-paste paradigm.

STL as a basic library contains generic definitions of data structures, algorithms, and related concepts. This structure was also reflected in our attempt to port the software. The conversion of data-structures and related algorithms was divided into two sub-tasks: first, changing the definition of a data structure, and, second, handling dependencies.

2 Change the definition of a data structure

After some preparations[2], data-structures and related algorithms were replaced by STL components. The list definitions found in the code mixed up two different aspects: parts of the data structure were application specific (e.g. attributes like `aclTarget`, `aclSource`) and parts were used for organizational purposes (e.g. `next`). These two parts were separated and subject to particular changes. The application specific information (data types) was modified in the following way:

1. Remove organization related information from the structure.
2. Add the default and copy constructor and a destructor.
3. Define $==$ and $<$ operators for the structure.

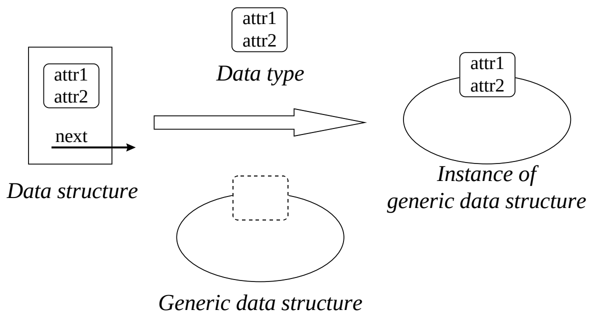

Figure 1: Replacement of data structures

Figure 1 shows how the appropriate generic component was then instantiated for the newly defined data type that only contained the essential information. The information concerned with data organization is handled by the container object (generic data structure) provided by STL. Further additions of some C++ specific operations were necessary to provide a *complete* data type and to fulfill the orthodox canonical form as described in [Coplien92].

[2]Complex expressions in the code were replaced by function objects to increase the maintainability.

```
struct aclEntry {
    int aclTarget;
    int aclSubject;
    int aclResources;
    // ...

    aclEntry ();
    aclEntry (int target, int subject, int resources);
    aclEntry (const aclEntry &);
    ~aclEntry ();
};

inline operator==(const aclEntry &,const aclEntry &);
inline operator<(const aclEntry &,const aclEntry &);

typedef list<aclEntry> acl_list;
```

These replacements were simple, since only initialization values had to be derived from existing initialization routines. The default- and copy-constructor were necessary for the list template. While the default-constructor simply initialized the attributes of the entry, the copy-constructor had to assign the values of a given entry to the attributes of the structure.

3 Dependencies

After replacing a data structure with an instance of a generic data structure, every function and module that had access to this data structure had to be modified accordingly. Depending on the interfaces between various C-modules, the conversion was rather sophisticated and took most of the effort involved in the whole C to C++ transformation. We classified the dependencies between different modules as follows: access dependency, function dependency, and structure dependency.

3.1 Access dependency

The C source code used pointers to directly access attributes of data structures. The access method for containers in STL are called iterators, which represent a well defined interface for gaining access to the items of a list.

Expressions such as	had to be replaced by
`aclEntry *ptr;`	`acl_list::iterator itr;`
`ptr=ptr->next;`	`itr++;`
or	or
`aclEntry *ptr;`	`acl_list::iterator itr;`
`ptr->subject=3;`	`(*itr).subject=3;`

The introduction of such components resulted in changes to all read and write operations on the variables kept in a list entry and on the list entries themselves (e.g. adding/removing new entries). Additionally, data encapsulation would make it necessary to add access methods for all attributes of an entry. However, the requirement of strict data encapsulation was not enforced by the transformation. Access dependency could be handled by a rather simple search and replace procedure [Paul94].

Malloc, free: Another task to be performed was changing the creation of a new entry. Instead of using *malloc* or *free* to create or delete an item, those statements were replaced by the *insert* (resp. *delete*) methods and a constructor was introduced for the initialization.

Casting: Some programmers (ab)use the effect that the order of attributes within a structure corresponds to the order of memory location of every single variable within a structure type. The technique is called *casting* which puts a different scheme onto an existing memory location. This approach is used to realize a kind of polymorphic behavior in C. It was used to put data items of different types into the same list. The conversion of data structures into STL containers caused severe problems since *casting* could not be projected onto generic components. Inheritance and one additional level of indirection could be used to solve this problem of loosing information when inserting such data items into STL containers. The initial approach of STL only allows to put objects of one type into the same container.

List vs. iterator: Since a pointer to a list entry in C can be used as a pointer to a single item as well as a pointer to a whole list, a more sophisticated problem was the distinction between a particular item and a whole list. During our work a heuristic approach was chosen:

- If the variable was dereferenced to gain access to the entry's attributes, then the variable was converted into an iterator.
- If only the next pointer was affected by the statements in the scope of the variable, then the variable's type was changed to a list.

This problem of distinguishing between a list and a single item was essential also for the next kind of dependency.

3.2 Function dependency

This type of dependency occurred when a data structure was involved in the declaration and definition of a function, either as parameter or return-value.

Parameter:

A function declaration of the kind	has to be replaced by either
`void f(struct aclEntry *)`	`f(acl_list::iterator)`
	or
	`f(acl_list &)`

The calling statements of such a function had to be changed too. The problem was to determine, whether the whole list would be needed within the function or just a particular entry. Analyzing the function was the only solution to solve this problem.

Return-value:

A function-declaration	**could not** be simply replaced by
`struct aclEntry * f()`	`acl_list::iterator f()`
	or
	`acl_list & f()`

Some functions returned a pointer to the beginning of a newly (locally) created list. Thus, another technique had to be used to overcome this problem: an additional parameter was provided for returning the newly created list instead of using the return-value. The assignment took place within the function:

```
int f(acl_list &);
```

Simultaneously, the return-value was changed to *int* to supply the calling function with information whether an error has occurred or not.

3.3 Structure dependency

Apart from lists, we discovered a tree data structure with a definition very similar to those of lists. However, it was not possible to apply the same transformations that were suitable for lists.

```
struct tree {
    struct tree *child_list;
    struct tree *next_peer;
    struct tree *parent;

    char label[MAXLABEL];
    u_long subid;
    int type;
};
```

In this case, it was not possible to map one data structure directly on an instance of a generic data structure. The data structure had to be analyzed in order to find possible mappings.

Recognizing and handling those dependencies might be seen as the most expensive task in terms of time, since all possibilities have to be checked to make a decision how to change a particular instance of the old data structure. This problem might be interesting for further studies.

4 Further Work

So far we succeeded in the task of replacing simple data structures and related algorithms with instances of generic definitions. Complex data structures require far more effort. We discovered that, for example, the data structure *tree* mentioned above could be replaced by an instance of the generic data structure set. This was the result of analyzing all functions that used a variable of type *tree*.

An evaluation of the resulting program size showed that the total amount of lines of code did not differ much. However, the size of the C source code modules could be decreased by about 10% due to the reduction of organizational overhead, which was necessary in the C source to maintain the list.

Our approach is also applicable to C++ programs that do not currently use generic components. The transformation process for C++ programs should be easier because of the encapsulation facilities and inheritance provided by the language.

The template facility of C++ that provides the basis for our approach offers further possibilities that can be exploited in the future. This has been shown in a paper on program generalization [Siff96]. Templates can also be used to configure subcomponents [VanHilst96]. We currently work on the integration of those approaches into our re-engineering activities.

5 Conclusions

We have presented a reengineering approach that is applicable to both C and C++ programs. It utilized generic components (data structures and algorithms) to replace common code fragments that are usually gained from copy-and-paste programming.

The replacement process of simple data structures resulted in higher maintainability of the source code because of encapsulation and locality. Data structure organization and memory management were separated from user specific data types.

Although the replacement of complex data structures is still an issue, we could empirically test a mapping from one data structure onto an instance of a related generic component. This requires a thorough analysis as well as more adaptations because data organization cannot be transformed in a kind of one-to-one mapping.

References

[Coplien92] James O. Coplien. *Advanced C++: programming styles and idioms.* Addison-Wesley, Reading, Mass. and London, 1992.

[Paul94] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering,* **20**(6):463–75, June 1994.

[Siff96] M. Siff and T. Reps. Program generalization for software reuse: from C to C++. *Forth Symposium on the Foundations of Software Engineering* (October 1996, San Francisco). ACM, 1996.

[Stepanov95] A. Stepanov and M. Lee. The standard template library. Hewlett-Packard Laboratories, 7 July 1995.

[VanHilst96] M. VanHilst and D. Notkin. Decoupling Change from Design. *Forth Symposium on the Foundations of Software Engineering* (October 1996, San Francisco). ACM, 1996.