Olympiads in Informatics, 2012, Vol. 6, 21–30

FCL-STL, a Generics-Based Template Library for FreePascal

Vladimír BOŽA, Michal FORIŠEK
Comenius University, Bratislava, Slovakia
e-mail: {usama,misof}@ksp.sk

Abstract. We present the design and usage of a new set of FreePascal units we created. These units aim to be the counterpart of the Standard Template Library in C++. The library is already included in the development branch of FreePascal (v. 2.7.1), and it should be a part of the stable branch (v. 2.6.?) at some point in future. Available data structures include vectors, sets, maps (both ordered and unordered) and more.

For many tasks used in past programming competitions there were C++ solutions that were significantly easier to implement than any Pascal solution. Problem setters usually needed to ensure that a reasonable Pascal solution exists. This library, once available, will mitigate the difference between the powers of these two languages.

Key words: FreePascal, templates, generic programming, algorithm library, STL.

1. Overview

In this section we give an overview of topics related to this paper. First, we discuss generic programming, in particular its use when creating algorithm and data structure libraries. Next, we explore the connection between such libraries and programming contests. At the end of this section we give an outline of the rest of the paper.

Generic Programming

In recent years the global trend in programming is towards library reuse. Modern programming languages such as Python come equipped with an extensive set of libraries that cover all the common needs of programmers.

One particular area contains the basic data structures and algorithms. From our point of view, this area is special and really stands out among the others. More precisely, the one thing that makes the data structure/algorithm libraries special is the need for a generic approach. For most other libraries the input domain is clearly defined. E.g., the regex library processes strings, the datetime library works with timestamps, the OS library provides a platform-independent API to access the directory structure. The situation is different with algorithms and data structures: everybody needs to store, access and sort records, but the content of those records and their correct comparison differ between programs. The libraries that provide data structures and algorithms have to be prepared to deal with this issue.

Modern programming languages deal with this issue by introducing *generic programming* (Musser and Stepanov, 1988). The basic syntactic elements of generic programming are usually called *templates*. A template essentially allows the programmer to use metavariables that represent unknown *types*. This language construct allows the library authors to write generic functions that can later be *instantiated* by a library user to work with any suitable data type(s). For instance, the following is a simple templated version of a swap function in C++:

```
template<typename T>
void swap(T &a, T &b) { T c; c=a; a=b; b=c; }
```

Note how the template metavariable `T` plays the role of an unknown type. Once you replace `T` by a particular type (e.g., `int`), you will get a valid C++ function that swaps the elements of that type.

The benefit of generic programming both for library authors and for library users is obvious. First of all, it prevents unnecessary duplication of code. E.g., there is just one sorting metafunction, instead of thousands ("a sort for lists of ints", "a sort for arrays of strings", etc.). This directly reduces bloat – the resulting library is physically smaller – and makes testing much simpler.

Finally, this approach is more versatile than traditional libraries in that it can be used even for data types not explicitly known to the library author. The users can define their own data types and process them using the same functions provided by the library.

Algorithm Libraries in Programming Languages; Their Use in Contests

Most of the modern programming languages come with extensive algorithm and data structure libraries. To name a few:

C++: The Standard Template Library (STL) has all the basic data structures and algorithms. (Many additional ones, e.g. including graph algorithms, are included in the Boost libraries (Boost C++ Libraries). Note that Boost is not allowed in most programming contests.)

Java: All basic data structures are in the Collections package.

Python: Some data structures (array-lists, dictionaries) are built-in, others are provided in the "Data Types" and "Numeric and Mathematical Modules" parts of the Python Standard Library.

On the contrary, no such library was available for FreePascal until now. Some data structures, but with no consistent interface and without generic programming, were available in the component library (`AVL_Tree`: an AVL tree containing pointers; `contnrs`: lists, stacks and queues of pointers or objects).

At programming contests this provided a clear disadvantage to contestants who use Pascal. For instance, this is the case at the International Olympiad in Informatics (IOI) where for many years the allowed programming languages are only Pascal and C/C++. The lack of a standard library for FreePascal has directly influenced the choice of past

competition tasks. For instance, tasks solvable using balanced binary trees were used in the competition only if there was an alternate solution in Pascal without balanced trees, but with the same asymptotic time complexity. (E.g., solutions that use various interval trees fall into this category.) In addition to leveling the playing field, this library should broaden the spectrum of suitable competition tasks for the IOI.

In the IOI Syllabus (Verhoeff *et al.*, 2006; Verhoeff *et al.*, 2012) most algorithms and data structures from standard libraries have the status "not for task description", meaning that such concepts should not be discussed in the task statements but using them may be necessary to solve some of the competition tasks. Here we see a direction in which the Syllabus ought to be improved in the future: At the moment, there is no clean distinction between *understanding* such data structures, *using* their library implementation and *implementing* them from scratch. The general consensus is that the first two are surely necessary; further discussion about requiring their implementation would be helpful.

About FCL-STL

The core of the FCL-STL FreePascal library was implemented in 2010 and 2011 by Vladimír Boža as a part of his Bachelor Thesis at Comenius University, Slovakia (Boža, 2011). The supervisor of this Thesis was Michal Forišek. At the moment, the library is available in the FreePascal development snapshot (Free Pascal team, 2012) in `fpc/packages/fcl-stl/`.

Outline of the Paper

Sections 2 and 3 list the algorithms and data structures currently implemented in FCL-STL. In Section 4 we provide a comprehensive table that compares the main features to C++ STL. In Section 5 we present data from practical benchmarks of FCL-STL.

2. Library Contents – Algorithms

In this section we list and briefly describe algorithms implemented as a part of the FCL-STL FreePascal library. If unclear about the semantics of a particular algorithm, please refer to (Cormen *et al.*, 2009) or an equivalent algorithm textbook. The same applies to the next section.

Sort

`Sort` is a generic array sort function. Internally the implementation uses IntroSort (Musser, 1997) – which is basically a QuickSort with a fallback to HeapSort in order to achieve a guaranteed $O(n \log n)$ worst case time complexity.

Random Shuffle

`RandomShuffle` randomly shuffles the elements in an array. The time complexity is worst case $O(n)$, and the random distribution is uniform assuming a uniform internal random function.

Next Permutation

`NextPermutation` rearranges the elements in an array to obtain the next permutation in lexicographic order and returns `true`. The only exception: if the input is an array with elements in descending order, the array is reversed (to obtain the lexicographically smallest permutation) and `NextPermutation` returns `false`.

The time complexity of a single call is $O(n)$. More precisely, the time complexity of a single call is linear in the number of necessary changes. Hence a full cycle iterating over all permutations of a given n-element array runs in $O(n!)$.

3. Library Contents – Data Structures

Vector

`TVector` is an array that can be resized at the end efficiently (doubling storage size when necessary). Most important operations:

- indexing using `[]` in $O(1)$;
- `PushBack` in amortized $O(1)$.

Stack

`TStack` is a traditional stack data structure. Most important operations:[1]

- `Push` in amortized $O(1)$;
- `Top` and `Pop` in $O(1)$.

Queue

`TQueue` is a traditional queue data structure. Most important operations:

- `Push` in amortized $O(1)$;
- `Front` and `Pop` in $O(1)$.

Deque

`TDeque` is a deque (usually pronounced [deck], also called a double-ended queue) – a self-resizing array that supports indexing and fast element addition/removal at both ends. Most important operations:

- `PushFront` and `PushBack` in amortized $O(1)$;
- indexing using `[]` in $O(1)$;
- `PopFront` and `PopBack` in $O(1)$.

[1]The implementations of `TStack`, `TQueue` and `TDeque` all use `TVectors` for internal data storage, hence the amortized complexity bounds on insertions. Although there are possible implementations of stacks and queues with guaranteed constant time complexity, this is the industry standard in other languages. For most uses in practice the amortized time complexity does not matter, and in the remaining cases the stacks/queues can easily be implemented as linked lists.

Priority Queue

`TPriorityQueue` is a priority queue that allows insertion of ordered elements and fast extraction of the maximal element. Internally the priority queue is implemented as a binary heap. Most important operations:

- `Push` in amortized $O(\log n)$;
- `Top` in $O(1)$;
- `Pop` in $O(\log n)$.

Ordered Sets and Maps

`TSet` is an ordered container of unique elements. `TMap` is an ordered associative array. Internally, sets are implemented using left-leaning red-black trees (Sedgewick, 2008), and maps are implemented as sets of pairs (key,value).

Most important operations:

- `Insert`, `Find`, `Delete` in worst-case $O(\log n)$;
- `Min`, `Max` in worst-case $O(\log n)$;
- `FindLess[Equal]`, `FindGreater[Equal]` in worst-case $O(\log n)$;
- iteration over all elements in worst-case $O(n)$;
- maps: indexing using `[]` in worst-case $O(\log n)$.

Unordered Sets and Maps

`THashSet` is an unordered container of unique elements. `THashMap` is an unordered associative array. Internally, unordered sets and maps are implemented as self-resizing hash tables, with collisions resolved by chaining.

Most important operations (assuming a good hash function is used):

- `Insert`, `Contains`, `Delete` in expected $O(1)$;
- iteration over all elements in worst-case $O(n)$;
- maps: indexing using `[]` in expected $O(1)$.

(At the moment there are no pre-written hash functions, the programmer has to provide one when using a data structure with a hash table. Default hash functions for basic types are a possible addition in the future.)

4. Comparison of Contents to C++ STL

Table 1 summarizes the correspondences between FPC-STL and the Standard Template Library for C++. We also highlighted some algorithms and data structures that are available in C++, but are not a part of FCL-STL yet.

Table 1
Correspondence between FCL-STL and its C++ counterpart

FreePascal FCL-STL	**C++ STL equivalent**
`Sort`	`sort`
`RandomShuffle`	`random_shuffle`
`NextPermutation`	`next_permutation`
`--`	`stable_sort, nth_element`
`TVector`	`vector`
`TStack, TQueue`	`stack, queue`
`TDeque`	`deque`
`TPriorityQueue`	`priority_queue`
`TSet, Tmap`	`set, map`
`THashSet, THashMap`	`unordered_set, unordered_map`
`--`	`list`
`--`	`bitset`
`--`	`(unordered) multiset`

5. Performance Tests

We made several benchmarks to test the efficiency of our implementation and to compare it to the implementation of STL in C++. The benchmarks were of two different types, focusing on two different topics:

- benchmarks measuring the efficiency of individual library components;
- benchmarks focusing on solving entire tasks from programming contests.

Together, the chosen benchmarks cover all relevant parts of FCL-STL. Some details on the benchmarking environment:

- AMD Athlon(tm) II X4 640 Processor (single core used), 4 GB RAM;
- Linux version 3.2.0-24-generic (Ubuntu/Linaro 4.6.3-1ubuntu5);
- gcc/g++ version 4.6.3, switches -std=gnu++0x, -O2;
- fpc version 2.4.4-3.1 with FCL-STL, switch -O2;
- All measured times are processor times (i.e., time actually spent running the application and executing its system calls);
- Each test was executed 10 times. The main plotted values are averages. All plots also have error bars showing the minimum and maximum, but as the measurements are pretty accurate, the error bars are usually invisible.

Benchmark #1: Push Back, Shuffle and Sort

In this benchmark the program creates an empty vector, inserts values 0 through $n - 1$ (using the push back method), randomly shuffles the vector and then sorts it. Obviously,

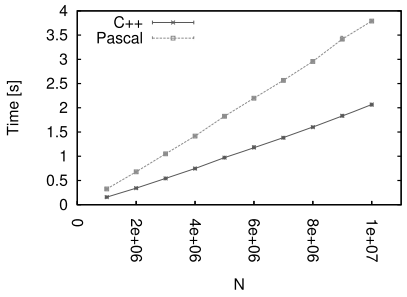

Fig. 1. Vectors and sorting.

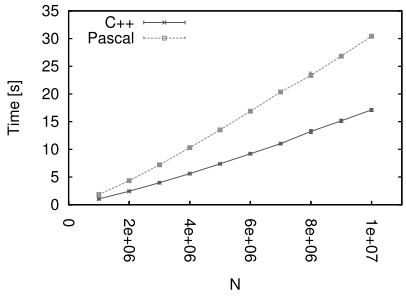

Fig. 2. Sets and iterators.

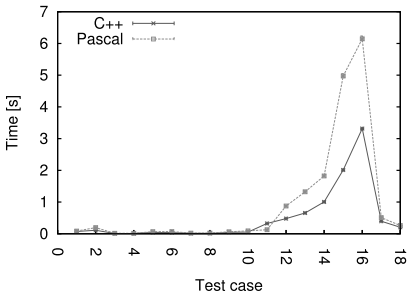

Fig. 3. The task “poet”.

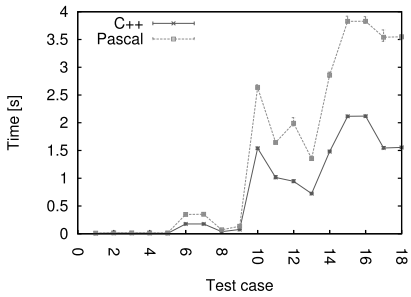

Fig. 4. The task “asphalt”.

most running time is spent in sorting the random permutation. The Pascal implementation is shown in the Appendix. Results of this benchmark for various n are plotted in Fig. 1.

Benchmark #2: Sets

In this benchmark the program creates an empty vector and pushes back n random values, each between 0 and $n - 1$, inclusive. Then all of these values are inserted into a set (effectively sorting them and discarding duplicates). Finally, the set is traversed using an iterator and the sorted values are copied into a new vector. This benchmark tests the efficiency of sets. The Pascal implementation is shown in the Appendix. Results of this benchmark for various n are plotted in Fig. 2.

Benchmark #3: Task “Poet”

For this benchmark we used the task “poet” (Slovak OI, national round 2011). We want make a poem of n couplets (two-line stanzas). For each of the n couplets we are given multiple options. We need to pick the couplets in such a way that for each i, line 2 of couplet i rhymes with line 1 of couplet $i + 1$. Also, the last line of the poem must rhyme with the first line, making the rhymes cyclic.

The solution is to search the state space: for each possible ending of the first line of the poem, inductively construct sets of all possible endings of last lines for couplets 1

through n. To store these sets, our implementations use unordered associative arrays (i.e., hash sets). Results are plotted in Fig. 3.

Benchmark #4: Task "Asphalt"

For this benchmark we used the task "asphalt" (Slovak OI, national round 2011). The goal is to find a shortest path in a 2D landscape by building roads, bridges and tunnels. The solution is a modified version of Dijkstra's shortest paths algorithm, using a priority queue to achieve a better time complexity. The data structures used are: a priority queue, vectors (and vectors of vectors), and stacks. Results are plotted in Fig. 4.

6. Conclusions

The FCL-STL library aims to become a standard algorithm and data structure library for FreePascal. It is worth noting that nowadays this is as far as a Pascal library can go. There is no standard for modern Pascal, and different compilers (e.g., Delphi) choose their own syntax for all language extensions such as generics.

The benchmarks quite consistently show that the Pascal implementations tend to be slower than their C++ counterparts approximately by a factor of 2. In our opinion, this makes the library usable enough for many practical uses, including programming contests.

The main plan for the near future is to have this library included into a stable FreePascal release.

To conclude this paper, the authors would like to thank two anonymous referees for their helpful comments and remarks.

References

Boost C++ Libraries. Available online at `http://www.boost.org/`.

Bož̆a, V. (2011). Knižnica štandardných algoritmov pre kompilátor FreePascal. Comenius University Bratislava, Bachelor thesis, Comenius University Bratislava, Slovak. Available online at `http://oldwww.dcs.fmph.uniba.sk/bakalarky/obhajene/getfile.php/boza11306397896881.pdf?id=159&fid=315&type=application%2Fpdf`.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to Algorithms*, 3rd edn., The MIT Press.

Free Pascal team. (2012). Free Pascal 2.7.x Daily Source Snapshot of Development Tree. Available online at `ftp://ftp.freepascal.org/pub/fpc/snapshot/trunk/source/fpc.zip`.

Musser, D.R. (1997). Introspective sorting and selection algorithms. *Software Practice and Experience*, 27, 983–993.

Musser, D.R., Stepanov, A.A. (1988). Generic programming. In: *Symbolic and Algebraic Computation: International Symposium ISSAC*, 13–25.

Sedgewick, R. (2008). Left-leaning red-black trees. *Workshop on Analysis of Algorithms*, Maresias, Brazil. Available online at `http://www.cs.princeton.edu/ rs/talks/LLRB/RedBlack.pdf`.

Verhoeff, T. Horváth, G., Diks, K., Cormack, G. (2006). A proposal for an IOI syllabus. *Teaching Mathematics and Computer Science*, 4(1), 193–216.

Verhoeff, T., Horváth, G., Diks, K., Cormack, G., Forišek, M. (2012). IOI Syllabus. Available online at `http://ksp.sk/ misof/ioi-syllabus/`.

Appendix

Usage Examples

Below we show a few Pascal source codes that use the FCL-STL library. The first source code is the code used for benchmark #1: random shuffling and then sorting a vector.

```
uses gvector, garrayutils, gutil;

type iLess     = specialize TLess<longint>;
     iVector   = specialize TVector<longint>;
     iOrdUtils = specialize TOrderingArrayUtils<iVector, longint, iLess>;
     iUtils    = specialize TArrayUtils<iVector, longint>;

var  V : iVector;
     n, i : longint;

begin
    read(n);
    V := iVector.Create;
    for i := 0 to n-1 do V.PushBack(i);
    iUtils.RandomShuffle(V,n);
    iOrdUtils.Sort(V,n);
end.
```

The second example is benchmark #2: sets and set iterators.

```
uses gvector, gset, gutil;

type iLess   = specialize TLess<longint>;
     iVector = specialize TVector<longint>;
     iSet    = specialize TSet<longint,iLess>;

var V : iVector;
    S : iSet;
    N, i : longint;
    it : iSet.TIterator;

begin
    read(N);
    V := iVector.Create();
    for i:=1 to N do V.PushBack(random(N));
    S := iSet.Create();
    for i:=0 to N-1 do S.Insert(V[i]);
    V.Clear();
    it := S.Min();
    repeat V.PushBack( it.GetData() ); until not it.Next();
end.
```

V. Bož̆a is a master's degree student at the Comenius University in Slovakia. He has multiple medals from IOIs and IPhOs and now he is an active organizer of various national and international programming contests. In the last two years he also worked as an intern in Google Zurich. His master's thesis is in the area of cryptography.

M. Forišek is an assistant professor at the Comenius University in Slovakia. Since 2006 he serves as an elected member of the International Scientific Committee (ISC) of the IOI. He is also the head organizer of the Internet Problem Solving Contest (IPSC). His research interests include theoretical computer science (hard problems, computability, complexity) and computer science education.