Alphabets in Generic Programming

Juha Kärkkäinen

Department of Computer Science, P.O.Box 68 (Gustaf Hällströmin katu 2 B)
FI-00014, University of Helsinki, Finland

e-mail: `Juha.Karkkainen@cs.helsinki.fi`

Abstract. We initiate the design of a software library of algorithms and data structures on strings. The design is based on generic programming, which aims for a single implementation of an abstract algorithm that works in every situation, particularly with any kind of string or sequence, without any disadvantage to a more specific implementation. The design requires a deep understanding of both different algorithms and various types of strings. In this paper, we address one aspect of strings, the alphabet. The main contribution is a novel definition of the concept of an alphabet in a program. The key feature is the recognition of two levels, the level of abstract algorithms and the level of concrete programs, and the establishment of a connection between the levels. Based on the definition, we provide a sketch of a design for alphabet traits, a crucial abstraction layer between algorithms and strings.

1 Introduction

Algorithms and data structures on strings [5, 12] are often practical: implementable with a reasonable effort and usable for real world problems. Indeed, many basic algorithms have been implemented several times in applications or for experimental evaluation, and practical aspects have been an important area of research (see, for example, [11]). However, existing implementations are usually hard to find, of low quality (even incorrect), or difficult to modify for new purposes. Thus, someone needing an implementation faces a lot of work whether implementing from scratch or starting from an existing implementation.

A good software library can significantly ease the task of an implementer as it provides a single source of high quality, well-tested and flexible implementations of algorithms and data structures. There are successful libraries in several areas of algorithmics including fundamental algorithms and data structures (STL [3]), graph algorithms (LEDA [10]), and computational geometry (CGAL [8]). Stringology has been identified as another area that is ripe for a software library and a proposal has been made [7], but nothing comparable to STL, LEDA or CGAL exists, yet.

The purpose of this paper is to initiate the design for a software library of algorithms and data structures on strings. The library design is based on the generic programming paradigm [3], which was established by STL and is also the basis of CGAL. Generic programming strives for simultaneous flexibility and efficiency through implementations that work with as many data types as possible without a loss of efficiency. Ideally, one can use a single generic implementation of an abstract algorithm in every situation without any disadvantage to a specialized implementation. In the case of stringology, generic programming means that the library algorithms should work efficiently with almost any kind of a string or a sequence.

Generic programming achieves its goal of genericity by the means of an abstraction layer between algorithms and the data they operate on, in this case strings. Designing this layer is the crucial step in designing an algorithm library. The layer needs to operate with a large number of different algorithms and a wide variety of string types, and a good design must be based on a deep understanding of both. Full analysis is far beyond the scope of this paper but we will start with one fundamental aspect.

A string can be defined as a sequence of characters, which reveals the two largely orthogonal aspects of strings: the *sequence aspect* and the aspect of individual character, which we will call the *alphabet aspect*. Sequences are central to STL, and there, a deep analysis of sequences and algorithms on sequences has led to the concept of iterators. A good introduction to iterators can be found in [3]. For understanding this paper, it is enough to think iterators as pointers to an array, with a sequence represented by a pair of iterators indicating the beginning and the end of the sequence.

We will concentrate on the alphabet aspect. We start with a motivating example of a simple algorithm illustrating the problem of alphabets in generic programming. We will then go on to analyze and define the concept of an alphabet. The central feature is the recognition of two levels, the level of abstract algorithm design and analysis, and the level of concrete implementations and programs. We establish a formal connection between the levels enabling one to see an alphabet at both levels simultaneously. Finally, we sketch the design of *alphabet traits* that forms a part of the abstraction layer between algorithms and strings.

C++ is the language of LEDA, STL and CGAL, and has the best support for generic programming (out of widely used languages, at least). It is thus the obvious choice of language. The fast development of template metaprogramming techniques in recent years [1, 2, 6, 13] has brought us closer to achieving the ideals of generic programming. Understanding this paper does not require knowledge of these techniques, though some knowledge of C++ may be helpful.

2 Example Algorithm

Consider the following simple algorithm that computes the number of distinct characters in a string.

```
count_distinct(string $S$)
 1   $seen := \emptyset$
 2   **for** each character $c$ of $S$ **do**
 3         $seen := seen \cup \{c\}$
 4   **return** $|seen|$
```

Two points in this algorithm are problematic for a generic implementation. One is the set *seen*, and the other is the iteration over the characters of S. The latter is involved with the sequence aspect of the string and is the kind of thing that iterators were designed for. The former is involved with the alphabet aspect and could be handled using the generic set data structure in STL. This would lead to the following typical STL-style function:[1]

```
template <typename Iterator>
int count_distinct(Iterator begin, Iterator end)  {
  typedef iterator_traits<Iterator>::value_type chartype;
  set<chartype> seen;
  for (Iterator i = begin; i != end; ++i)
    seen.insert(*i);
  return seen.size();
}
```

This is a quite generic implementation, but it is slower than necessary in many cases since the set is implemented with a balanced search tree. In particular, in the most common case of the characters being of type `char`, the following function is significantly faster for a long string.

```
template <typename Iterator>
int count_distinct(Iterator begin, Iterator end)  {
  vector<bool> seen(256,false);
  for (Iterator i = begin; i != end; ++i)
    seen[*i]=true;
  return count(seen.begin(), seen.end(), true);
}
```

Using standard techniques, we could use the latter implementation, when the characters are of type `char` and the former otherwise. However, choosing the optimal data structure for the set is not that simple:

- If the alphabet is a small range of integers, we should use a vector, whatever the character type.
- If the alphabet is a small set of integers from a large range, a hash table might be the choice.
- Even balanced tree is not quite as generic as is possible. It requires order comparisons, which not all C++ types have, and which, even when available, might do the wrong thing (see below). In such cases, we could still implement the set as an unordered list.

Further complexity can be created by an unusual concept of character equality. Consider the following examples:

- With a case insensitive alphabet, an upper case and a lower case letter are considered to be the same character, and are counted as one.

[1]We have simplified the C++ code in this paper by ignoring some quirks of C++: omitted `typename` at places, used `vector<bool>` though it's not the best choice, assumed `char` is unsigned, etc.

- A character in a protein sequence might contain information about secondary or tertiary structure in addition to the amino acid. If we want to count distinct amino acids, however, the extra information should be ignored when comparing characters.
- Two floating point values might be considered the same if they round to the same integer.

All the examples could be handled by creating first a new string using an appropriate character conversion, but at the cost of a time and space overhead. In character counting, the overhead is probably small, but in other cases it could be significant. For example, the Boyer–Moore algorithm [4] usually accesses only a small fraction of characters and converting all of them could be costly.

The above discussion shows that we cannot expect the C++ type of characters to carry all relevant information about the alphabet. A separate entity (a type or an object) is needed for that purpose. In generic programming, such entities are known as *traits* (see `iterator_traits` above). The C++ standard library does, in fact, include something called character traits, but they are more of a relic from time before generic programming. We will call our traits *alphabet traits*.

Let us finally see what an implementation of our counting function using alphabet traits might look like. (A full implementation with a usage example is in the Appendix.)

```
template <typename Iterator, typename Alphabet>
int count_distinct(Iterator begin, Iterator end, Alphabet A)  {
  typedef generate_set<Alphabet>::type charset;
  charset seen(A);
  for (Iterator i = begin; i != end; ++i)
    seen.insert(*i);
  return seen.size();
}
```

Here `Alphabet` is an alphabet traits *type* and `A` an alphabet traits *object*. The *metafunction* `generate_set` chooses the appropriate implementation for the set.

Despite its simplicity, the above algorithm captures a lot of the difficulties with alphabets in generic programming. For example, the problem of implementing a node in a trie or an automaton is closely related to the problem of implementing the set *seen*.

3 Alphabet

Alphabet traits describe the properties of an alphabet, which itself is a more abstract entity. Before designing alphabet traits, we need to define more clearly what an alphabet is. That is the purpose of this section and, indeed, the main purpose of this paper.

When we talk about an alphabet in a generic implementation of an abstract algorithm, we are talking about two different things. One is the *abtract alphabet*, the mathematical set appearing in problem definitions, abstract algorithms and their asymptotic analysis. The other is the *concrete alphabet*, which is a specific representation of an alphabet in a program.

3.1 Abstract Alphabet

An abstract alphabet is the set of all possible characters. The following properties of the set are of interest:

- *ordering*: Does the alphabet have a linear order?
- size: Is it *constant*, σ (*finite*), or *infinite* (unknown)?
- *integrality*: Are the character integers?

One could also specify other properties but these are sufficient for most situations arising in design and analysis of abstract algorithms. Note that we allow infinite and unordered alphabets.

Consider the character counting algorithm from Section 2. The best implementation of the character set and the resulting complexity depend on the properties of the alphabet. For a string of length n, we have the following complexities for various kinds on alphabets:

- infinite: $\mathcal{O}(n^2)$
- finite: $\mathcal{O}(n \min\{n, \sigma\})$
- constant: $\mathcal{O}(n)$
- ordered: $\mathcal{O}(n \log n)$
- finite and ordered: $\mathcal{O}(n \log \min\{n, \sigma\})$
- finite and integral: $\mathcal{O}(n + \sigma)$ deterministic, $\mathcal{O}(n)$ randomized

3.2 Concrete Alphabet

A concrete alphabet is a representation of an abstract alphabet based on the following three principles:

- All character representations are values of a single C++ type `T`.
- Not all values of `T` need to represent a character.
- Multiple values may represent the same character.

Formally, a *concrete alphabet* $\mathcal{A}$ is a triple $(\texttt{T}, C, \sim)$, where

- `T` is a C++ type.
- C is a subset of the possible values of the type `T`.
- $\sim$ is an equivalence relation on C.

The concrete alphabet $\mathcal{A}$ defines an *abstract alphabet* $\widetilde{\mathcal{A}}$ as the set of equivalence classes of C under $\sim$. We will denote by $[a]$ the equivalence class containing a.

Two distinct but equivalent character values are different representations of the same abstract character. The two representations should behave identically in all algorithms. For example, a don't-care character that matches all other characters is distinct from other characters and forms its own equivalence class. Its special matching properties are not part of the alphabet but a separate entity called a matching relation.

3.3 Conversions

The restriction to a single type applies to concrete alphabets but not abstract alphabets as multiple concrete alphabets can represent the same abstract alphabet. Conversions between concrete alphabets are the mechanism to deal with this.

Let $\mathcal{A}$ and $\mathcal{B}$ be two concrete alphabets. A *conversion* from $\mathcal{A}$ to $\mathcal{B}$ is a mapping $f\colon C^{\mathcal{A}} \to C^{\mathcal{B}}$ that is homomorphic w.r.t. $\sim$, i.e., $a \sim a' \Rightarrow f(a) \sim f(a')$ for all $a, a' \in \mathcal{A}$. Then, we can define $\widetilde{f}\colon \widetilde{\mathcal{A}} \to \widetilde{\mathcal{B}}$ by $\widetilde{f}([a]) = [f(a)]$. The following properties of $\widetilde{f}$ are of interest:

- $\widetilde{f}$ is an *embedding* if it is injective (one-to-one), i.e., $[a] \neq [a'] \Rightarrow \widetilde{f}([a]) \neq \widetilde{f}([a'])$.
- $\widetilde{f}$ is an *isomorphism* if it is a surjective embedding, i.e., an embedding satisfying $\widetilde{f}(\widetilde{\mathcal{A}}) = \widetilde{\mathcal{B}}$.

If there is an isomorphism $\widetilde{f}\colon \widetilde{\mathcal{A}} \to \widetilde{\mathcal{B}}$, we can say that $\mathcal{A}$ and $\mathcal{B}$ are two representations of the same abstract alphabet. Similarly, an embedding implies a subset relation.

The mapping $\widetilde{f}$ being an embedding or an isomorphism does not imply that the conversion f is injective or surjective. The following lemmas characterize embeddings and isomorphisms in terms of conversions.

Lemma 1. *$\widetilde{f}$ is an embedding iff $a \not\sim b \Rightarrow f(a) \not\sim f(b)$.*

Lemma 2. *$\widetilde{f}\colon \widetilde{\mathcal{A}} \to \widetilde{\mathcal{B}}$ is an isomorphism and $\widetilde{g}\colon \widetilde{\mathcal{B}} \to \widetilde{\mathcal{A}}$ is its inverse iff $\widetilde{f}$ and $\widetilde{g}$ are embeddings and $g(f(a)) \sim a$ for all $a \in \mathcal{A}$.*

Embedding conversions in particular play a central role in the library as we will see later. Isomorphic conversions come into play when inverse conversions are involved.

3.4 Ordered alphabets

A concrete *ordered* alphabet $\mathcal{A}$ is a quadruple $(\mathtt{T}, C, \sim, <)$, where $\mathtt{T}$, C and $\sim$ are as before and $<$ is a strict order on C satisfying: For all $a, b \in C$, exactly one of $a < b$, $a \sim b$ and $b < a$ is true. (We also define $\lesssim$ in the usual way.) The corresponding abstract ordered alphabet $\widetilde{\mathcal{A}}$ has an order $\preceq$ defined by $[a] \preceq [b]$ if $a \sim b$ or $a < b$.

A mapping $\widetilde{f}\colon \widetilde{\mathcal{A}} \to \widetilde{\mathcal{B}}$ is *order preserving* if it is homomorphic w.r.t. $\preceq$.

Lemma 3. *$\widetilde{f}$ is order-preserving iff f is homomorphic w.r.t. $\lesssim$.*
$\widetilde{f}$ is an order-preserving embedding iff f is homomorphic w.r.t. $<$.

Order preservation is a surprisingly subtle issue. There are common isomorphisms and embeddings that are not order-preserving. The standard conversion from `signed char` to `unsigned char` is an example. Also, order preservation is often not required even when order comparisons are involved. For example, the implementation of a character set using a balanced search tree requires a linear order but what the order is does not matter. A non-order-preserving conversion would not be a problem then. We will therefore not generally require conversions to be order preserving. However, when the *problem* definition involves an order, for example in the case of sorting, the conversions must be order preserving.

3.5 Integral Alphabets

Many algorithmic techniques work only or primarily on integral alphabets. These include using a character as an array index, computing fingerprints or hash values, radix sorting, etc. These techniques can be made available to a wide variety of alphabets through embeddings to proper integral alphabets.

A concrete alphabet $(\mathtt{T}, C, \sim, <)$ is a *primary integral alphabet* if `T` is a built-in integral type (for example `char` or `int`), C is a range of the form $[0, \sigma)$, $\sim$ is the standard `operator==`, and $<$ is the standard `operator<`. Requiring the minimum to be zero simplifies many of the techniques mentioned above.

A concrete alphabet is a *secondary integral alphabet* if there is an embedding conversion f from it to a primary integral alphabet. An integer range with a minimum other than zero is a secondary integral alphabet, too.

Of additional interest is an isomorphic conversion *from* a primary integral alphabet. For example, random generation of characters can be accomplished using it.

4 Alphabet Traits

The character type `T` does not, in general, contain full information about the alphabet. Additional information in a form usable by algorithms is provided by *alphabet traits*. We will not describe the full design of alphabet traits but give a glimpse to their use with examples.

An alphabet traits is partly a C++ class and partly an object of that class. The class contains *static information* about the alphabet, i.e., information that is known at compile time and can be used for compile time optimization. An object of that class may contain additional *dynamic information*. For example, whether an alphabet is integral or not is always static information but the size of the integral range might be dynamic information.

4.1 Writing Generic Algorithms

The example in Section 2 shows the use of alphabet traits in writing generic algorithms at its simplest. Almost all details are hidden inside the metafunction `generate_set`, which is a part of the basic library infrastructure.

Obtaining more detailed information is demonstrated in the following example. Let `Alphabet` be an alphabet traits class and `A` an object of the class. If the alphabet is integral, we can obtain the conversion to a primary integral alphabet as follows:

```
get_char2int<Alphabet>::type char2int = make_char2int(A);
```

Then `char2int(ch)` performs the conversion for the character `ch`. Comparison functions, for example, are obtained similarly.

The above statement would not even compile for a non-integral alphabet. However, there are standard metaprogramming techniques for conditional compilation based on compile time predicates [1]. In this case, we can determine the integrality, at compile time, using the metafunction

```
is_integral<Alphabet>::value
```

As we saw in Section 2, alphabet traits is supplied as an argument to a function. To make things simpler for the caller of the algorithm, the argument should be optional. When no argument is supplied, the *default alphabet traits* for the character type is used instead. In the case of the `count_distinct` function, this is accomplished by providing the following second variant of the function.

```
template <typename Iterator>
int count_distinct(Iterator begin, Iterator end)  {
  typedef iterator_traits<Iterator>::value_type chartype;
  typedef default_alphabet<chartype>::type alphabet;
  return count_distinct(begin, end, alphabet());
}
```

4.2 Creating Alphabets

As mentioned, algorithms typically assume a default alphabet if no alphabet traits is provided by the user. If the default is not correct, the user needs to pass a correct one as an argument to the algorithm. The library will provide a number of alphabet traits for common situations. If none of these is satisfactory, there are metafunctions for creating custom alphabets.

The following example shows one way for creating a case-insensitive alphabet.

```
struct caseless_equal {
  bool operator() (char a, char b) {
    return tolower(a)==tolower(b);
  }
};
typedef construct_alphabet<char,
                  set_equivalence<caseless_equal> >::type
        caseless_alphabet;
```

Now a call such as `count_distinct(begin, end, caseless_alphabet())` would count upper and lower case letters as one.

The above alphabet is not ordered or integral as no order comparison or integral conversion is provided. Therefore, the set in `count_distinct` would be implemented as an unordered list. An order comparison and an integral conversion could be provided as additional arguments to the metafunction, but there is simpler way:

```
struct tolower_conversion {
  char operator() (char c) { return tolower(c); }
};
typedef embedded_alphabet<char, default_alphabet<char>::type,
                    tolower_conversion >::type
        caseless alphabet;
```

Here we create a new alphabet by embedding it to an existing alphabet. Many properties including ordering and integrality are automatically inherited. There is a similar metafunction `isomorphic_alphabet` that also takes the inverse conversion as an argument.

Integral alphabets are common and useful alphabets and there is a separate metafunction for creating alphabet traits for them. For example,

```
integral_alphabet<char, 10, 20>::type
```

creates an alphabet representing the range $[10, 20]$.

All the example alphabet traits here contain no dynamic information. Creating alphabet traits with dynamic information is more complicated and we ignore the details here.

5 Concluding Remarks

The purpose of this paper is to iniate the design of a string algorithms library based on the generic programming paradigm. We have addressed only one fundamental but limited aspect of the library, the alphabet. However, we believe that the design approach based on a careful analysis of concrete examples leading to a definition of the concept of an alphabet and the programming techniques developed for implementing the design provide a good start for the design of further aspects of the library.

The design of the sequence aspect has already been provided to an extent, thanks to the STL iterators and some further work building on them (`http://boost.org/libs/iterator/doc/`, `http://boost.org/doc/html/string_algo/design.html`, and `http://boost.org/libs/range/`). There are still issues remaining, though. For example, in some cases the alphabet and sequence aspects cannot be fully separated without a loss of efficiency [9].

Still more aspects are relevant to a string algorithms library. We have already mentioned one, match relation. Other issues arise, for example, from approximate string matching and other more complex stringology problems.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Addison–Wesley, 2004.

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison–Wesley, 2001.

[3] M. H. Austern. *Generic Programming and the STL.* Addison–Wesley, 1999.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, Oct. 1977.

[5] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison–Wesley, 2000.

[7] A. Czumaj, P. Ferragina, L. Gasieniec, S. Muthukrishnan, and J. L. Träff. The architecture of a software library for string processing. In *Proceedings of Workshop on Algorithm Engineering*, pages 294–305, 1997. Online proceedings at `http://www.dsi.unive.it/~wae97/proceedings/`.

[8] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. *Software — Practice and Experience*, 30(11):1167–1202, 2000.

[9] K. Fredriksson. Faster string matching with super-alphabets. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 2476 of *LNCS*, pages 44–57. Springer, 2002.

[10] K. Mehlhorn and S. Näher. *LEDA — A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[11] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings.* Cambridge University Press, 2002.

[12] B. Smyth. *Computing Patterns in Strings.* Pearson Addison–Wesley, 2003.

[13] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide.* Addison–Wesley, 2002.

A Full Example

Here is the full implementation of the `count_distinct` algorithm.

```
#include"glas/set.hpp"
#include<iterator>

// count the number of distinct characters in a string
// generic form with an alphabet as an argument
template< typename Iterator, typename Alphabet>
int
count_distinct(Iterator begin,
               Iterator end,
               Alphabet A)
{
  typedef typename glas::generate_set<Alphabet>::type charset;
  charset seen;
  for (Iterator i = begin; i != end; ++i) {
      seen.insert(*i);
  }
  return seen.size();
}

// specific form that uses default alphabet
template<typename Iterator>
int
count_distinct(Iterator begin, Iterator end)
{
  typedef typename std::iterator_traits<Iterator>::value_type
      char_type;
  typedef typename glas::default_alphabet<char_type>::type
      alphabet;
  return count_distinct(begin, end, alphabet());
}
```

Below is an program that uses the `count_distinct` function with a case insensitive alphabet.

```
#include "count.hpp"
#include "glas/alphabet_traits.hpp"
#include<string>
#include<iostream>

// case insensitive alphabet
struct tolower_conversion
{
  char operator() (char c) const { return tolower(c); }
};
struct caseless_alphabet
  : glas::embedded_alphabet<
       char,
       glas::default_alphabet<char>::type,
       tolower_conversion
    >::type
{};

int main()
{
  std::string str("ABRACAdabra");
  int cnt = count_distinct(str.begin(), str.end(),
                            caseless_alphabet());
  std::cout << cnt << " distinct characters in "
            << '"' << str << '"' << "\n";
  // prints: 5 distinct characters in "ABRACAdabra"
}
```