The Hash-Trie of Knuth & Liang: A C++11 Implementation

Ştefan Vargyas
stvar@yahoo.com

Oct 31, 2015

Table of Contents

1 The Hash-Trie Program **1**

2 The C++11 Implementation of Hash-Trie **1**
- 2.1 The `Pascal` Implementation of Knuth 2
- 2.2 The Structure of `hash-trie.cpp` Source File 6
- 2.3 The Implementation of `HashTrie<>` Class Template 11

3 The `hash-trie` Binary **21**

4 References **24**

1 The Hash-Trie Program

The **Hash-Trie** program's vocation is that of a C++ tool for exercising the interesting *hash-trie* data structure of Knuth **[Knuth92**, p. 157**]** – a variant of the so called dynamic packed trie data structure of Liang **[Liang83**, p. 32**]**. **Hash-Trie** shall provide an *implementation* for this data structure in modern C++ – that is the new C++11 language **[Strou13]** – and shall encompass the implementation into a framework serving the function of *test bed* for various easily definable *configurations* of hash-tries.

For the purpose of having an *implementation* of the *hash-trie* data structure which facilitates the ease of definition of different *configurations* of the structure, upon a few initial iterations over Knuth's material **[Knuth92**, pp. 151–167**]**, I designed and implemented a set of "modules", of which the leading one – the `HashTrie` module –, being parametrized statically by a handful of *configuration parameters*, gives the means for the birth of as many concrete *hash-trie* structures as needed. For the sake of simplicity all modules were hosted by a single source file, `hash-trie.cpp`.

As a result of compiling the sole source file `hash-trie.cpp` with configuration parameters set according to the needed configuration of *hash-trie*, a binary **`hash-trie`** comes to existence, which will further be used by the *test bed* for testing purposes.

One important driving force behind designing the modules of `hash-trie.cpp` the way they were, was the desire to have a clear-cut separation between the major parts of the program:

module	contents
Sys	utilities close to the undelying system libraries
Ext	extensions of the standard C++ library
HashTrie	everything belonging to the *hash-trie* data structure
Main	the `main` function of **`hash-trie`** along with a few related utilities

2 The C++11 Implementation of Hash-Trie

The first objective – set forth at the beginning of section 1, The **Hash-Trie** Program – is that of providing an C++11 implementation for the *hash-trie* data structure of Knuth & Liang. This data structure in defined and implemented in chapter 6, *Programming Pearls, Continued: Common Words (1986)*, of **[Knuth92**, p. 151**]** as core part of a program solving the following problem proposed by Bentley **[BKM86]**:

> Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

Knuth states in §17, *Dictionary Lookup*, on page 157, of his work, that the *hash-trie* structure is a variant of a data structure due to Liang's Ph.D. work – see section *Dynamic packed tries* of chapter *Pattern generation* in **[Liang83**, p. 32**]**.

2.1 The `Pascal` Implementation of Knuth

The general structure of Knuth's `Pascal` program[1] is revealed in §3 on page 152:

```
program common_words(input, output);
type { Type declarations (17) }
var { Global variables (4) }
{ Procedures for initialization (5) }
{ Procedures for input and output (9) }
{ Procedures for data manipulation (20) }
begin
{ The main program (8) }
end.
```

Upon looking deeper into the implementation, one can see that, the *global variable*s constitute the main mechanism for communication between the parts – i.e. the `procedure`s and the `function`s – of the program, or even, within one part of it only. This is to say that the program does not encapsulate related data into, e.g., `record` structures – thus, nor does he attach specific `procedure`s and `function`s to such structures –, for to have a more clear-cut division between the relevant pieces of functionality of the program.

Another trait of this `Pascal` program is that system dependecies – as these are implied by the potential necessity of running the program on several different supporting platforms – are not dealt with at all. This is of course justifiable by the fact that the `Pascal` language and it's environment – at the time of Knuth's writing – had no concern at all with these kind of issues.[2]

[1] To be more precise, Knuth implemented his solution to Bentley's problem in his own `Web` system of programming (see chapter 4, *Literate Programming*, of **[Knuth92]** for a thorough description of this system). `Web` boils down the source *literate program* given as input to a completely defined `Pascal` source program.

[2] For an interesting and detailed yet debatable analysis of `Pascal`'s shortcomings look upon the well-known report of Kernighan **[Kern81]**. Note that the paper appeared before ISO/IEC 7185 – the `Pascal` standard – was created in 1983.

Nevertheless, Knuth is well aware of the matters of portability and, nota bene, he does have his own solutions to such kind of difficulties in the case of the complex TeX system of typesetting he has been built **[Knuth92**, ch. 4, sec. *Portability*, pp. 122–124**]**.

The third trait of this Pascal program is concerning the issue of handling and reporting the error conditions of at least two kinds: logic errors and run-time errors. (Simply put, the category of logic errors refers to those errors which are due to programming mistakes within the internal logic of the program; the category of run-time errors encompasses the kind of fault conditions due to the outer world of the program – these are either conditions originating from the supporting environment or from the user mishandling the program.) With respect to the category of logic errors, the program relies completely on Pascal's type system: there are no supplemental checks of this kind made in the code.

On the other hand, concerning the run-time errors, the code handles very scarcely the errors which are due to the user misusing the program. For example, look at the code implementing the function read_int in §9 on page 154: there is an insufficient wary about the user providing an invalid numerical input, and there is none with regards to an otherwise valid numerical input that could overflow the capacity of the internal integer type involved:

```
{ 9. Basic input routines. [...] a function
  that reads an optional positive integer,
  returning zero if none is present at the
  beginning of the current line. }
function read_int: integer;
var n: integer; { the accumulated value }
begin
  n := 0;
  if not eof then
  begin
    while (not eoln) and (input^ = '␣') do
      get(input);
    while (input^ >= '0') and (input^ <= '9') do
    begin
      n := 10 * n + ord(input^) - ord('0');
      get(input);
    end;
  end;
  read_int := n;
end;
```

In this context, it is to be noted the interesting remark Knuth made[3] at the beginning of the section which includes the code above (§9, p. 154):

> It will be nice to get the messy details of Pascal input out of the way and off our minds.

The other input procedure of the program is get_word (see it below, on the following page). Similarly to the function read_int above, the code of get_word pays little concern to the run-time errors that might occur due to the faulty input given.

Having in mind the remarks made above, for refining farther the first of the two main objectives stated in section 1, I choose for the C++11 implementation of *hash-trie* to be closely attached to Knuth's own Pascal implementation, but *no closer*.

The point here is that my intention is not to make a one-to-one translation of Pascal code to

[3] Knuth's remark echoes Kernighan's own accounts about Pascal's environment, and, particularly, about Pascal's I/O subsystem. (see section 4 of Kernighan's report.)

C++, but a translation that is made up of choices resulted upon careful deliberation of which `Pascal` construct is transformed to what C++ equivalent construct. An illustrating example might be the following: the implementation in §13 on page 156 defines several things as shown by the code below:

```
define max_word_length = 60 { words shouldn't be longer than this }
buffer: array [1..max_word_length] of 1..26; { the current word }
word_length: 0..max_word_length;
       { the number of active letters currently in buffer }
word_truncated: boolean;
       { was some word longer than max_word_length? }
```

where the nearly surrounding context is as follows:

> **13.** Each new word found in the input will be placed into a `buffer` array. We shall assume that no words are more than 60 letters long; if a longer word appear, it will be truncated to 60 characters, and a warning message will be printed at the end of the run.

From my perspective, it is obvious that the C++ implementation shouldn't impose such an uptight limitation on the length of the input words. On the other hand, yet more importantly, is that the implementation shoudn't define `buffer` to be of type `char[max_word_length+1]`, but to be `char*` – i.e. a null-terminated string – and/or `std::string`. Such an option of transformation from `Pascal` to C++ would render `max_word_length`, `word_length` and `word_truncated` as useless and, consequently, they will not be translated into the implementing C++ code. Moreover, deriving from this choice of translation, the `I/O Pascal` code of §15–16 on pages 156–157 which is listed below:

```
{ 15. We're now ready for the main input routine,
  which puts the next word into the buffer. If no more
  words remain, word_length is set to zero; otherwise
  word_length is set to the length of the new word. }
procedure get_word;
label exit; { enable a quick return }
begin
  word_length := 0;
  if not eof then
  begin
    while lettercode[ord(input^)] = 0 do
      if not eoln then
        get(input)
      else
      begin
         read_ln;
         if eof then return;
      end;
    { Read a word into buffer (16) }
    { 16. At this point lettercode[ord(input^)] > 0,
      hence input^ containts the first letter of a word. }
    repeat
      if word_length = max_word_length then
        word_truncated := true
      else
      begin
        incr(word_length);
        buffer[word_length] := lettercode[ord(input^)];
      end;
```

```
      get(input);
    until lettercode[ord(input^)] = 0;
  end;
exit:
end;
```

will be translated to something simply like the C++-styled `I/O` statements:

```
std::string  input_word;
std::istream input_stream = ...;
std::getline(input_stream, input_word);
```

The string `input_word` will than be *passed by value* to where is needed. Furthermore, is it easily foreseeable that the mapping provided by the `lettercode` table (§11–12 on pages 155–156, see below) should be deferred to a *traits* class – which will become a default template parameter of and it will be used by the class template that will encompass the core functionality of the *hash-trie* data structure.

One more remark concerning the peculiarities of `Pascal` is referring to, in Knuth's words, the following things (§11–12 on page 155):

> **11.** To find words in the *input* file, we want a quick way to distinguish letters from nonletters. `Pascal` has conspired to make this problem somewhat tricky, because it leaves many details of the character set undefined. [...]
>
> If `c` is a value of type `char` that represents the `k`th letter of the alphabet, then `lettercode[ord(c)] = k`; but if `c` is a nonletter, `lettercode[ord(c)] = 0`. We assume $0 \le$ `ord(c)` ≤ 255 whenever `c` is of type `char`.

```
{ Global variables (4) += }
lowercase, uppercase: array [1..26] of char; { the letters }
lettercode: array [0..255] of 0..26; { the input conversion table }
```

> **12.** A somewhat tedious set of assignments is necessary for the definition of `lowercase` and `uppercase`, because letters need not be consecutive in `Pascal`'s character set.

```
{ Set initial values (12) }
lowercase[1] := 'a'; uppercase[1] := 'A';
...
lowercase[26] := 'z'; uppercase[26] := 'Z';
for i := 0 to 255 do
  lettercode[i] := 0;
for i := 1 to 26 do
begin
  lettercode[ord(lowercase[i])] := i;
  lettercode[ord(uppercase[i])] := i;
end;
```

Indeed, `lowercase` and `uppercase` are not necessary in the environment of C++: the implementation will simply use either `tolower` or `towlower` functions. Similarly to the case of `lettercode`, the mapping provided by `lowercase` will be deferred to the traits class associated to the *hash-trie* structure.

For the category of one-to-one translations from `Pascal` to C++, hereafter will follow a couple of examples of places where the `Pascal` implementation matches quite well the translated C++ side. As a matter of fact these places are part of the core algorithms and inner data structures of *hash-trie* – for which I decided that the C++ side be as close as possible to the `Pascal` side as it was laid down by Knuth.

`Pascal`'s type system is known to be of a *strong* kind. Unfortunately, at times this type system seems to be *too strong*: e.g. the size of an array or string is part of its type, thus making certain programming tasks quite difficult.[4] Above was shown that, for certain cases, `array`'s of fixed size in `Pascal` where not translated one-to-one to C++, but an adaptation has been made such that the translation to suite the target style of programming and the surrounding environment.

However, this is not to be generalized completely, because, for a couple of inner structures of *hash-trie* – indeed, `array`'s of fixed size –, the C++ implementation will keep its declarations tightly parallel with the originating `Pascal` ones. These declarations are indicated below, extracted from §17, *Dictionary Lookup*, on page 158:

```
define trie_size = 32767 { the largest pointer value }
pointer = 0..trie_size;
```

from §18 on page 159:

```
define empty_slot = 0
define header = 27
link, sibling: array [pointer] of pointer;
ch: array [pointer] of empty_slot..header;
```

from §24 on page 160:

```
define tolerance = 1000
```

and from §32, *The frequency counts*, on page 162:

```
define max_count = 32767 { count's won't go higher then this }
count: array [pointer] of 0..max_count;
```

Two of the `define`s above – `trie_size` and `tolerance` – will become part of the *configuration parameters* of *hash-trie*'s generic structure: out of specification of each of them, concrete *hash-trie*s will be obtained.

The next two sections to come will introduce a few more of these configuration parameters. Anticipating a bit, but in direct relation with the `Pascal` context of this section, one should observe the necessity of at least two more parameters – let's name them `ARRAY_BOUNDS` and `STRICT_TYPES`.

These parameters associate with two `Pascal` matters which have to be dealt with: arrays indexing and arithmetical expressions. The `Pascal` run-time support ensures that arrays indexing are within the bounds defined for the type involved and that arithmetical expressions on `integer`s do no exceed the limits of the respective types. The C++ code should be flexible enough implementing transparently these kinds of run-time checks, yet, it should be able to let these supplemental checks out of a particular build of *hash-trie* when efficiency is of paramount importance.

2.2 The Structure of `hash-trie.cpp` Source File

The C++ source file of **Hash-Trie** was structured according to the initial design (see the table on page 1). Each of the "modules" `Sys`, `Ext` and `HashTrie` were encompassed within a

[4] Subsection 2.1 of **[Kern81]** accounts for this problem out of the experience the author had rewriting a weighty set of programs in `Pascal`.

namespace with the same name. However, the handful members of `Main` module were defined at the global scope:

name & description	type
`program` and `verdate`	**`const char`**`[]`
The sole global constants of the program. Define the name and the version number and date of the program. The constant `program` is defined by the build parameter `PROGRAM` – which is controlled by **`make`**.	
`global_options_t`	**`struct`**
Encapsulate all global option variables in one structure.	
`globals`	`global_options_t`
The sole global variable. Give access to the global options.	
`options_t`	**`class`**
Parse command line options, update `globals` and provide the action option. These things are obtained upon invoking the sole publicly defined method of the class: `static const options_t options(int argc, char* argv[])`.	
`print_config`	*function*
Print out the configuration parameters of the program.	
`print_op_types<>`	*function template*
Print out the types corresponding to the named operation.	
`print_types`	*function*
Print out the types of *add* and *sub* operations by calling `print_op_types<>`.	
`hash_trie_error`	`printf`-*like variadic function*
Print out formatted error messages and, if told to do so, exit the program.	
`hash_trie_t`	*type alias*
`HashTrie::HashTrie<Sys::char_t>`.	
`print_func_t`	*type alias*
`void (hash_trie_t::*)(Sys::ostream&) const`.	
`exec_hash_trie`	*function*
Load up words read from `stdin` in an instance of `hash_trie_t` and than call on that instance for the named print function, if given. When the `STATISTICS` configuration parameter was enabled at compile-time, if `globals.print_stats` says so, then also print out the statistics information of the instance.	
`main`	*the* `main` *function of* `hash-trie.cpp`
Dispatch the action option obtained from function `options_t::options` by calling the appropriate handler function: `print_config`, `print_types` or `exec_hash_trie`.	

The complete definition of `main` function is as follows:

```
int main(int argc, char* argv[])
{
    const auto opt =
        options_t::options(argc, argv);

    switch (opt.action) {
    case options_t::print_config:
        print_config();
        break;
    case options_t::print_types:
        print_types();
        break;
    case options_t::load_trie_only:
        exec_hash_trie();
        break;
    case options_t::print_trie:
        exec_hash_trie(
            &hash_trie_t::print);
        break;
    case options_t::dump_trie:
        exec_hash_trie(
            &hash_trie_t::dump);
        break;
    default:
        SYS_UNEXPECT_ERR(
            "action='%zu'",
            Ext::size_cast(opt.action));
    }

    return 0;
}
```

The action options relating to `hash_trie_t` are dispatched to the function `exec_hash_trie`:

```
using hash_trie_t = HashTrie::HashTrie<Sys::char_t>;

using print_func_t = void (hash_trie_t::*)(Sys::ostream&) const;

void exec_hash_trie(print_func_t print_func = nullptr)
{
    size_t lno = 1;
    hash_trie_t trie;
    hash_trie_t::string_t str;
    while (Ext::getline(Sys::cin, str)) {
        try {
            if (!trie.put(str) && !globals.quiet)
                hash_trie_error(
                    false, lno,
                    "failed to put '%"
                    SYS_CHAR_TYPE_FMTS
                    "s' in trie",
                    str.c_str());
        }
        catch (const HashTrie::Error& err) {
            hash_trie_error(
                true, lno, "%s", err.what());
        }
        lno ++;
    }

    if (print_func)
        (trie.*print_func)(Sys::cout);

#ifdef CONFIG_HASH_TRIE_STATISTICS
    if (globals.print_stats)
        trie.print_stats(Sys::cout);
#endif
}
```

The namespace Sys contains system-related declarations and definitions:

name & description	type
SYS_CHAR_TYPE_NAME	#define
Specify the type used for character representation throughout the program. Depending on the value of CONFIG_HASH_TRIE_CHAR_TYPE being 0 or 1, is either char or wchar_t.	
SYS_CHAR_TYPE_FMTS	#define
Specify the length modifier of a '%s' format specifier. Depending on the value of CONFIG_HASH_TRIE_CHAR_TYPE being 0 or 1, is either the empty string or "l".	
Sys::char_t	*type alias*
Define the type used for character representation: SYS_CHAR_TYPE_NAME.	
Sys::ostream	*type alias*
Define the I/O streams type used for output: std::basic_ostream<Sys::char_t>.	
Sys::istream	*type alias*
Define the I/O streams type used for input: std::basic_istream<Sys::char_t>.	
Sys::cin	Sys::istream&
Define the I/O streams object associated with stdin. Depending on the value of CONFIG_HASH_TRIE_CHAR_TYPE, it refers to std::cin or to std::wcin.	
Sys::cout	Sys::ostream&
Define the I/O streams object associated with stdout. Depending on the value of CONFIG_HASH_TRIE_CHAR_TYPE, it refers to std::cout or to std::wcout.	
Sys::cerr	Sys::ostream&
Define the I/O streams object associated with stderr. Depending on the value of CONFIG_HASH_TRIE_CHAR_TYPE, it refers to std::cerr or to std::wcerr.	
Sys::die	printf*-like variadic function*
Handle fatal error conditions: print out a formatted error message to stderr and than exit the program. If globals.dump_backtrace says so, print out the backtrace of the program prior to bailing out.	
Sys::assert_failed	*function*
Handle an assertion failed error condition: invoke die with a proper argument list.	
Sys::unexpect_error	printf*-like variadic function*
Handle an unexpected error condition: invoke die with a proper argument list.	
SYS_ASSERT	*preprocessor macro*
Test for the expression provided as argument to be true: if not, raise an assertion failed error by calling in Sys::assert_failed. Nothing is done when DEBUG is not defined.	
SYS_UNEXPECT_ERR	*variadic preprocessor macro*[5]
Raise an unexpected error by calling in Sys::unexpect_error.	

[5] Variadic preprocessor macros via __VA_ARGS__ are by now legal in C++11: ISO/IEC 14882:2011, 16.3 Macro

name & description	type
`Sys::error<>`	`printf`-*like variadic function template*
Format an error string and than throw it via an exception object of type specified.	
`Sys::clocks_t`	**`struct`**
Encapsulate three time counters of microsecond resolution: `real`, `user` and `sys`. Define convenient operators: `+`, `+=`, `-`, `-=` and, for printing out, `<<`. Is compiled in only when defining `CONFIG_HASH_TRIE_STATISTICS`.	
`Sys::clock_t`	**`struct`**
For printing it out, restrict to the specified one of the three time counters of a `clocks_t` object. Is compiled in only when defining `CONFIG_HASH_TRIE_STATISTICS`.	
`Sys::utime_t`	**`class`**
Define a convenient interface to the system API `gettimeofday` and `getrusage` functions for time measurements of microsecond resolution. Return a `clocks_t` object. Is compiled in only when defining `CONFIG_HASH_TRIE_STATISTICS`.	

The namespace `Ext` contains many useful extensions – function templates and class templates – to the standard C++ library. However, the table that follows was shortened for to list only the most important of these extensions:

name & description	type
`Ext::min<>`, `Ext::max<>`	**`constexpr`** *function templates*
Provide handy shortcuts to the corresponding `std::numeric_limits<>` functions.	
`Ext::digits<>`	**`constexpr`** *function template*
Provide a handy shortcut to the `std::numeric_limits<>::digits` constant.	
`Ext::size_cast<>`	*function template*: `size_t(T)`
Cast safely an integer of type narrower or equal width than `size_t` to `size_t`.	
`Ext::array<>`	*function template*: `Sys::array_t<T,N>(T(&)[N])`
Access C-style arrays via a range checking `operator[]`.	
`Ext::array_size<>`	**`constexpr`** *function template*: `size_t(T(&)[N])`
Return the number of elements of C-style arrays.	
`Ext::getline<>`	*function template*
Replace `std::getline<>` on `std::basic_string<>` with an optimized version.[6]	
`Ext::box_t<>`	**`class`** *template*
Provide a wrapper class around integer types for to perform bounds checkings on the assignments and overflow and underflow checkings on the arithmetic operations involved.	

replacement, pts. 4, 5 and 10; 16.3.1 Argument substitution, pt. 2; 16.3.5 Scope of macro definitions, pt. 9.

[6] New in C++11 is that `std::string` of `char`-like objects is required to store contiguously its elements (see ISO/IEC 14882:2011, 21.4.1 pt. 5). A consequence of this is that the growing of a `std::basic_string<>` inside `Ext::getline<>` can be done efficiently in an I/O buffer-like manner.

The namespace HashTrie contains everything related to the *hash-trie* data structure. A short synopsis follows below. The next subsection will delve deeply into the implementation code.

name & description	**type**
HashTrie::Error	**struct**
Define the exception class used for signaling the few run-time error conditions that might occur in HashTrie::HashTrie<> class. It is derived from std::runtime_error.	
HashTrie::types_t	**enum**
Name the four main types used by class HashTrie::HashTrie<>: count_type, pointer_type, letter_type and cell_type.[7]	
HashTrie::types_traits_t	**struct** *template* <typename T, types_t t>
Define the underlying integer type and the associated limits for the named type t.	
HashTrie::add<>	*function template*
Define an addition operation, transparently over the boxed or just plain underlying unsigned integer types used by HashTrie::HashTrie<>. Implement the operation such a way that the type of the resulting value be wide enough to avoid overflowing.	
HashTrie::sub<>	*function template*
Define an subtraction operation, transparently over the boxed or just plain underlying unsigned integer types used by HashTrie::HashTrie<>. Implement the operation such a way that the type of the resulting value be wide enough to avoid underflowing.	
HashTrie::char_traits_t	**struct** *template* <typename C = char>
Define the default traits class template associated to the character type used by HashTrie::HashTrie<>.[8]	
HashTrie::size_traits_t	**struct**
Encompass a few static parameters of class HashTrie::HashTrie<>: max_count, trie_size and tolerance. The last two of these are defined by corresponding CONFIG_HASH_TRIE_* configuration parameters.	
HashTrie::HashTrie<>	**class** *template*
Implement the main functionality of the *hash-trie* data structure of Knuth & Liang.	

2.3 The Implementation of HashTrie<> Class Template

This section is devoted to plunging into the most concrete details of the implementation code of the *hash-trie* structure of Knuth & Liang. The previous section glimpsed over the members of HashTrie namespace which accomodate the class HashTrie<> and its immediate relatives. Below is listed the C++ code for the definitions of class HashTrie<>, of struct template type_traits_t, of struct template char_traits_t, and of struct size_traits_t.

[7] The Pascal types to which the names defined by HashTrie::types_t are referring to were shown by the listings on pages 5–6. More details about the C++ counterparts of these types will be given in the next subsection.

[8] The class template HashTrie::char_traits_t<> relates closely to the issues of tables lowercase, uppercase and lettercode in the Pascal implementation. (see the accounts given on page 5.)

Also is listed the implementation code for the constructor of the `HashTrie<>` class, for its public method `put` and for its most important function, the private method `find`.

The listings were shortened a bit by eliminating certain peripheral inline function code and all static assertions, comment lines and debugging code.[9]

For the implementation code in `hash-trie.cpp`, I took into account the newest corrections of the `Pascal` implementation of *hash-trie* as published by Knuth on his own site.[10] (see the source lines 2259 and 2314 below.)

The `Pascal` type and variable definitions on pages 5–6, as was already emphasized, are part of the core structures of *hash-trie*. These definitions imply four kinds of *integer subrange* types to be used by the C++ implementation: (The subrange type `1..26` is brought forward by the definitions of tables `lowercase`, `uppercase` and `lettercode` on page 5.)

`Pascal` **name**	`Pascal` **subrange type**	C++ `enum` **name**
unnamed	`0..max_count`	`count_type`
`pointer`	`0..trie_size`	`pointer_type`
unnamed	`1..26`	`letter_type`
unnamed	`empty_slot..header`	`cell_type`

The C++ types which correspond to the `Pascal` subrange types above are defined through the `struct` template `types_traits_t`. For each `enum` name there is a specialized version of the `types_traits_t` which defines an inner type `type_t` – the C++ counterpart of the `Pascal` subrange type of the table above. The integral constants needed for these defintions are taken in by `types_traits_t` from the class namespace of its template argument `T`. Further to be seen below is that the template argument `T` will actually name the `HashTrie<>` class itself (see the source lines 2038–2047 below: there is an instance of the well-known C++ idiom of name *Curiously Recurring Template Pattern*, *CRTP*).

```
1696 template<typename T>
1697 struct types_traits_t<T, count_type>
1698 {
1699     typedef unsigned short base_t;
1700 #if defined(CONFIG_HASH_TRIE_STRICT_TYPES)
1701     typedef Ext::box_t<
1702         base_t, types_traits_t<T, count_type>> type_t;
1703 #else
1704     typedef base_t type_t;
1705 #endif
1706
1707     // count_t: [0..max_count]
1708     static constexpr size_t min = 0;
1709     static constexpr size_t max = T::max_count;
 ... ...
1715 };
1716
1717 template<typename T>
1718 struct types_traits_t<T, pointer_type>
1719 {
1720     typedef unsigned short base_t;
1721 #if defined(CONFIG_HASH_TRIE_STRICT_TYPES)
1722     typedef Ext::box_t<
1723         base_t, types_traits_t<T, pointer_type>> type_t;
```

[9] The listing parts of `hash-trie.cpp` seen in this section and in the previous one are annotated with actual line numbers from the source file itself.

[10] **http://www.cs.stanford.edu/~uno/lp.html**: the web page of **[Knuth92]** – contains the errata of the latest printing of the book, the sixth, of 2013;
http://www.cs.stanford.edu/~uno/lp-err.ps.gz: the errata for the first printing of the book.

```
#else
    typedef base_t type_t;
#endif

    // pointer_t: [0..trie_size]
    static constexpr size_t min = 0;
    static constexpr size_t max = T::trie_size;
...
};

template<typename T>
struct types_traits_t<T, letter_type>
{
    typedef unsigned char base_t;
#if defined(CONFIG_HASH_TRIE_STRICT_TYPES)
    typedef Ext::box_t<
        base_t, types_traits_t<T, letter_type>> type_t;
#else
    typedef base_t type_t;
#endif

    // letter_t: [min_letter..max_letter]
    static constexpr size_t min = T::min_letter;
    static constexpr size_t max = T::max_letter;
...
};

template<typename T>
struct types_traits_t<T, cell_type>
{
    typedef unsigned char base_t;
#if defined(CONFIG_HASH_TRIE_STRICT_TYPES)
    typedef Ext::box_t<
        base_t, types_traits_t<T, cell_type>> type_t;
#else
    typedef base_t type_t;
#endif

    // cell_t: [empty_slot..header]
    static constexpr size_t min = T::empty_slot;
    static constexpr size_t max = T::header;
...
};
```

The purpose of template `char_traits_t` is that of alleviating the cumbersome `Pascal`-specific mechanisms embodied by the tables `lowercase`, `uppercase` and `lettercode` of Knuth's implementation (see the accounts on page 5). Designing `HashTrie<>` with a template template parameter `T` framing up the *character representation traits* of the class was a consequence of the following requirement: the class need to avoid hard-coding its dependecies on character representation peculiarities within the implementation code itself. Having the template parameter `T` – with the default value set to the fully fledged `char_traits_t struct` template – enhances the flexibility of the implementation of `HashTrie<>` working transparently with several different types of character representations.

```
template<typename C = char>
struct char_traits_t
{
    typedef
        typename
            std::remove_cv<C>::type
        char_t;
    typedef
        Ext::unsigned_t<char_t>
        uchar_t;
    typedef
        std::char_traits<char_t>
```

```
        traits_t;
    typedef
        Ext::unsigned_t<
            typename traits_t::int_type>
        uint_t;
    typedef
        size_t code_t;
    typedef
        std::basic_string<char_t>
        string_t;
...
    static constexpr uint_t to_int(char_t ch)
    { return traits_t::to_int_type(ch); }

    static constexpr char_t to_char(uint_t uint)
    { return traits_t::to_char_type(uint); }

    static constexpr char_t to_lower(char_t ch);
...
    static constexpr bool is_valid_char(char_t a)
...
    static constexpr bool is_valid_code(code_t a)
...
    static code_t to_code(char_t c)
...
    static char_t from_code(code_t c)
...
    static constexpr code_t min_letter = 1;
    static constexpr code_t max_letter = min_letter +
...
    static constexpr code_t header     = max_letter + 1;
    static constexpr code_t empty_slot = 0;
};
```

The sole role of size_traits_t is that of encapsulating the Pascal defines on pages 5–6:

```
struct size_traits_t
{
    static constexpr size_t max_count = 32767;
    static constexpr size_t trie_size = CONFIG_HASH_TRIE_TRIE_SIZE; // = 32767;
    static constexpr size_t tolerance = CONFIG_HASH_TRIE_TOLERANCE; // = 1000;
};
```

The class template HashTrie<> has three template arguments: C, T and S. C is the type used by the class for character representation; T is the traits class associated to the character representation type of the class; S is the class of which namespace defines statically three size-related constants parametrizing HashTrie<>. Sensible defaults were provided for each of these the template parameters – defaults which were already presented above.

The public interface of HashTrie<> is quite simple: a default constructor, two overloaded methods put, and three printing related methods: print, dump and print_stats. The third of these printing methods is available only when hash-trie.cpp was compiled with the *statistics collecting code* enabled, i.e. when the CONFIG_HASH_TRIE_STATISTICS configuration parameter was #defined.

```
template<
    typename C = char,
    template<typename> class T = char_traits_t,
    typename S = size_traits_t>
class HashTrie :
    private T<C>,
    private S
{
public:
    typedef S size_traits_t;
    typedef T<C> char_traits_t;
```

```
    typedef typename char_traits_t::char_t char_t;
    typedef typename char_traits_t::string_t string_t;

    HashTrie();

    bool put(const char_t*);

    bool put(const string_t& str)
    { return put(str.c_str()); }

    void print(std::basic_ostream<char_t>&) const;
    void dump(std::basic_ostream<char_t>&) const;

#ifdef CONFIG_HASH_TRIE_STATISTICS
    void print_stats(std::basic_ostream<char_t>&) const;
#endif

private:
...
    template<types_t t>
    using types_traits_t = types_traits_t<HashTrie, t>;

    using size_traits_t::max_count;
    using size_traits_t::trie_size;
    using size_traits_t::tolerance;
    using char_traits_t::min_letter;
    using char_traits_t::max_letter;
    using char_traits_t::empty_slot;
    using char_traits_t::header;
...
    template<types_t i, types_t e>
    struct array_of_t
    {
        typedef
            typename types_traits_t<e>::type_t
            elem_t;

        class type_t
        {
        public:
            static constexpr size_t lo =
                types_traits_t<i>::min;
            static constexpr size_t hi =
                types_traits_t<i>::max;
...
        private:
...
            elem_t array[hi - lo + 1];
        };
    };

    typedef typename types_traits_t<count_type>::type_t   count_t;
    typedef typename types_traits_t<pointer_type>::type_t pointer_t;
    typedef typename types_traits_t<letter_type>::type_t  letter_t;
    typedef typename types_traits_t<cell_type>::type_t    cell_t;

    typedef
        typename array_of_t<pointer_type, count_type>::type_t
        count_array_t;
    typedef
        typename array_of_t<pointer_type, pointer_type>::type_t
        pointer_array_t;
    typedef
        typename array_of_t<pointer_type, cell_type>::type_t
        cell_array_t;

    cell_array_t    ch;
    count_array_t   count;
```

```
    pointer_array_t link;
    pointer_array_t sibling;

    // x_n = (alpha * n) % mod_x
    pointer_t x;
...
    static letter_t to_letter(char_t c)
    { return char_traits_t::to_code(c); }

    static char_t to_char(letter_t l)
    { return char_traits_t::from_code(l); }
...
    pointer_t find(const char_t*);

    void print_word(pointer_t, std::basic_ostream<char_t>&, bool) const;
    void dump_word(pointer_t, std::basic_ostream<char_t>&, bool&) const;
    void dump_empty(std::basic_ostream<char_t>&) const;

    static constexpr size_t make_alpha(size_t trie_size, size_t max_letter)
...
    { return std::ceil(0.61803 * (trie_size - 2 * max_letter)); }
...
    static constexpr size_t alpha = make_alpha(trie_size, max_letter);
    static constexpr size_t mod_x = trie_size - 2 * max_letter;
    static constexpr size_t max_h = mod_x + max_letter;
    static constexpr size_t max_x = mod_x - alpha;
...
#ifdef CONFIG_HASH_TRIE_STATISTICS
    struct stats_t
    {
        size_t loaded_words = 0;
        size_t failed_words = 0;
        size_t duplicated_words = 0;
        size_t empty_slots = 0;

        Sys::clocks_t trie_time = {};
...
    };

    mutable stats_t stats;
#endif
};
```

The struct template HashTrie<>::array_of_t<> and its inner defined class type_t is very important to the internal economy of class HashTrie<>. This is so because the class array_of_t<>::type_t is implementing the C++ counterparts of the Pascal arrays seen on pages 5–6. This class provides two oveloaded methods (which were cut off from the listing above), namely operator[] and assign. Each of these methods are to be seen at work in the constructor of class HashTrie<> and in its find method.

The method templates operator[] are offering transparently range-checked or range-unchecked access to an underlying C-style array. The dimension of this C-array is determined by the first template argument of array_of_t<>, while the second template argument of array_of_t<> decides the type of the elements of this array. Yet an important feature of the operator[] methods of array_of_t<>::type_t is the following: these methods model the semantics of Pascal arrays – that is that indexing within the structure doesn't start at 0, but at the position defined by the respective type declaration.

A third significant trait of these operator[] methods is that they accept transparently boxed or just plain integer typed values as indices. HashTrie<> uses boxed integers within its internal workings if and only if the configuration parameter CONFIG_HASH_TRIE_STRICT_TYPES is #defined at the time of its compilation.

The alternative of checking or not the indexed access into such an *array* structure is arbitrated by the configuration parameter CONFIG_HASH_TRIE_ARRAY_BOUNDS: indexing is range-checked if and only if this parameter is #defined at the time of compilation of hash-trie.cpp.

The methods ``void assign(size_t first, size_t last, *type* value)'' do assign value to each element of the underlying array in the range [first, last]. Similarly to the operator[] methods, these assign methods do range checking if and only if the configuration parameter CONFIG_HASH_TRIE_ARRAY_BOUNDS was #defined prior to initiate the compilation of hash-trie.cpp.

At this moment, one can easily examine the veracity of the forecast made in subsection 2.1 about the core Pascal structures of *hash-trie* defined on pages 5–6 to be tightly parallel with the corresponding C++ structures. The definitions of link, sibling, ch and count, as shown by the Pascal source lines 1–9 on page 6, and their parallel C++ definitions shown by source lines 2210–2228 on pages 15–16, are identical indeed.

One final note about the declaration of HashTrie<> concerns its static function make_alpha: even though make_alpha makes use of the standard non-constexpr function std::ceil, its declaration as constexpr is, nota bene!, correct.[11]

The constructor of class HashTrie<> is very simple: it initializes the inner structures and variables of the class exactly as Knuth does **[Knuth92**, §19, p. 159 and §23, p. 160**]**:

```
template<
    typename C,
    template<typename> class T,
    typename S>
HashTrie<C, T, S>::HashTrie()
{
    ch.assign(header, trie_size, empty_slot);
    link.assign(1, max_letter, 0);
    count.assign(1, max_letter, 0);

    for (pointer_t i = 1; i <= max_letter; ++ i) {
        ch[i] = i;
        sibling[i] = i - 1;
    }

    link[0] = 0;
    count[0] = 0;
    ch[0] = header;
    sibling[0] = header - 1;

    x = 0;
}
```

The put methods of HashTrie<> are little more than just public front-ends to the hard-working private function find. Nevertheless, the put method shown below does certain housekeeping: it checks for valid user input[12], increments the counter associated to the input word, if any, and, when CONFIG_HASH_TRIE_STATISTICS is #defined, updates its stats structure:

```
template<
    typename C,
    template<typename> class T,
    typename S>
bool HashTrie<C, T, S>::put(const char_t* str)
{
    if (*str == '\0')
        Sys::error<Error>("hash tries cannot contain empty words");

```

[11] By the C++11 standard, ISO/IEC 14882:2011, the function std::ceil doesn't have to be constexpr, yet **gcc** of version at least 4.8.0, in the header file cmath, boils std::ceil down to the undocumented built-ins __builtin_ceilf and __builtin_ceill.

[12] The *trie* structures **[Knuth98**, §6.3**]** – in particular the *hash-tries* –, by definition, cannot contain empty words.

```
#ifdef CONFIG_HASH_TRIE_STATISTICS
    Sys::utime_t time;
#endif

    // 34. Input the text, maintaining a dictionary with frequency count
    auto p = find(str);

#ifdef CONFIG_HASH_TRIE_STATISTICS
    stats.trie_time += time();

    if (p)
        stats.loaded_words ++;
    else
        stats.failed_words ++;
    if (p && count[p])
        stats.duplicated_words ++;
#endif

    if (!p) return false;
    count[p] ++;
    return true;
}
```

The method `find` of `HashTrie<>` class constitutes the central part of Knuth's *hash-trie* data structure. It substantiates the core algoritm of this data structure – it is the C++ counterpart of Knuth's `Pascal` function `find_buffer` **[Knuth92**, §20–21 and §24–31 on pp. 159–162**]**:

```
template<
    typename C,
    template<typename> class T,
    typename S>
typename
    HashTrie<C, T, S>::pointer_t
    HashTrie<C, T, S>::find(const char_t* str)
{
...
    const pointer_t tolerance2 = tolerance;
    // trial header location
    pointer_t h;
    // the final one to try
    pointer_t last_h; // INT: int last_h;

    const auto get_set_for_computing_header_locations = [&]() {
        // 24. Get set for computing header locations
...
        if (x >= max_x)
            x -= max_x;
        else
            x += alpha;

        h = x + max_letter + 1; // now max_letter < h <= trie_size - max_letter
...
        if (h > max_h - tolerance) {
...
            last_h = add(h, tolerance2) - mod_x;
        }
        else {
...
            last_h = h + tolerance;
        }
    };

    const auto compute_the_next_trial_header_location = [&]() {
        // 25. Compute the next trial header location h, or abort find
...
        if (h == last_h)
            return false;
```

```
        if (h == max_h)
            h = max_letter + 1;
        else
            h ++;
        return true;
    };

    // the current word position
    pointer_t p
        = to_letter(*str ++);
    while (*str) {
        // current letter code
        letter_t c
            = to_letter(*str ++);
        // 21. Advance p to its child number c
        if (link[p] == 0) {
            // 27. Insert the firstborn child of p and move to it, or abort find
            get_set_for_computing_header_locations();
...
            do {
                if (!compute_the_next_trial_header_location())
                    return 0;
...
            } while (ch[h] != empty_slot || ch[h + c] != empty_slot);
...
            link[p] = h;
            link[h] = p;
            p = h + c;
            ch[h] = header;
            ch[p] = c;
            sibling[h] = p;
            sibling[p] = h;
            count[h] = 0;
            count[p] = 0;
            link[p] = 0;
        }
        else {
            // the next word position
            pointer_t q
                = link[p] + c;
            if (ch[q] != c) {
                if (ch[q] != empty_slot) {
                    // 29. Move p's family to a place where child c will fit,
                    //     or abort find
                    // family member to be moved
                    pointer_t r;
                    // amount of motion
                    // INT: int delta;
                    // have we found a new homestead?
                    bool slot_found;
                    // 31. Find a suitable place h to move, or abort find
                    slot_found = false;
                    get_set_for_computing_header_locations();
...
                    do {
                        if (!compute_the_next_trial_header_location())
                            return 0;
...
                        if (ch[h + c] == empty_slot) {
                            r = link[p];
                            auto delta = sub(h, r); // INT: delta = h - r;
                            while (ch[r + delta] == empty_slot &&
                                    sibling[r] != link[p])
                                r = sibling[r];
                            slot_found = ch[r + delta] == empty_slot;
                        }
                    } while (!slot_found);
...
```

```
2615                    q = h + c;
2616                    r = link[p];
2617                    auto delta = sub(h, r); // INT: delta = h - r;
2618                    do {
2619                        sibling[r + delta] = sibling[r] + delta;
2620                        ch[r + delta] = ch[r];
2621                        ch[r] = empty_slot;
2622                        count[r + delta] = count[r];
2623                        link[r + delta] = link[r];
2624                        if (link[r]) link[link[r]] = r + delta;
2625                        r = sibling[r];
2626                    } while (ch[r] != empty_slot);
2627                }
2628                // 28. Insert child c into p's family
2629                h = link[p];
2630                while (sibling[h] > q)
2631                    h = sibling[h];
2632                sibling[q] = sibling[h];
2633                sibling[h] = q;
2634                ch[q] = c;
2635                count[q] = 0;
2636                link[q] = 0;
2637            }
2638            p = q;
2639        }
2640    }
2641    return p;
2642 }
```

Due to the infrastructure already built, the above implementation is almost one hunderd percent a direct translation of Knuth's `Pascal` code to C++. The few minute yet important differences between the two implementations are listed by the table below. The second column of the table contains references relative to **[Knuth92]** and the third column – references to the C++ source code from above.

name & description	`Pascal` **source**	**C++ source**
declaration of `last_h`	p. 161, §26, #3	p. 18, #2449
The `Pascal` code declares the type of `last_h` to be `integer`. The C++ code declares this variable to be of type `pointer_t`.		
assignment to `last_h`	p. 161, §24, #7	p. 18, #2523
The `Pascal` code assigns to `last_h` by the expression "`h + tolerance - mod_x`". The C++ code does that by the expression "`add(h, tolerance2)- mod_x`".		
declaration of `delta`	p. 162, §30, #3	p. 19, #2594
The `Pascal` code declares `delta` an `integer` and assigns to it twice. The C++ code removes the shared declaration and replaces each assignment with a complete definition.		
1st assignment to `delta`	p. 162, §31, #5	p. 19, #2607
The `Pascal` code assigns to `delta` by the statement "`delta = h - r`". The C++ code replaces this assignment with "**`auto`** `delta = sub(h, r)`".		
2nd assignment to `delta`	p. 162, §29, #3	p. 20, #2617
The same difference as in the case of the 1st assignment to `delta`.		

The difference of the two implementations stems from the fact that the `Ext::box_t<>` class template, which accomplishes the boxing of the integral types used by `HashTrie<>` – i.e. the integer types of name base_t defined inside the class namespaces of each of the four instances of the class template types_traits_t<> above –, is quite strict with regards to the operations applied to values of its type. The class template Ext::box_t<> is supposed to simulate in C++ the semantics of subrange types of Pascal, that is that it has to do bounds checking on assignments on values of its type. Ext::box_t<> is, of course, doing that and, more so, it checks the bounds on each of its arithmetical operations too.

The replacement of expression "h + tolerance - mod_x" with the expression "add(h, tolerance2) - mod_x" had to be done, because, as it is easily provable, under cetain conditions, the subexpression "h + tolerance" is exceeding the upper bound trie_size of the boxed integer pointer_t. It is worthy of notice the readily provable fact that both assignments to last_h on lines 2523 and 2533 are correct: each of the expression on the right side of these assignments do not exceed upon evaluation the bounds of pointer_t.

The two assignments to delta shown above had to be adjusted because of two reasons. The first one is that upon subtracting two pointer_ts h and r, the result would be of type pointer_t too. However, the original intent of delta included the possibility of it being a negative quantity.

The other reason for the code adjustement applied to delta is obvious once noticing that the boxed integral types used by HashTrie<> are each parametrized. Thus one cannot hard-code the type of delta without eventually getting into trouble. The function template Ext::sub<> is designed such a way that it decides the type of its resulting value based on the type of its arguments: the resulting type is choosen such that to avoid the occurence of undeflow errors upon the subtraction of the two input values.

3 The hash-trie Binary

The **Hash-Trie**'s main component is the binary program **hash-trie** which results upon compiling and linking the source file hash-trie.cpp. The command line options of **hash-trie** program are as follows:

```
$ ./hash-trie --help
usage: hash-trie [OPTION]...
where the options are:
  -P|--[print-]config  action: print out the config parameters
  -T|--[print-]types   action: print out the add/sub types table
  -L|--load-only       action: only load the input into the hash trie
  -p|--print[-trie]    action: print out all <word, count> pairs from
                         the hash trie (default)
  -d|--dump[-trie]     action: dump out the complete hash trie structure
  -b|--[no-][dump-]backtrace
                       dump a backtrace of the program on fatal error
                         or otherwise do not (default not)
  -c|--[no-][print-]chars
                       print out characters instead of codes on structure
                         dumps or otherwise do not (default not)
  -w|--[no-][print-]words
                       print out words when dumping out the hash trie
                         or otherwise do not (default do)
  -s|--[no-][print-]stats
                       print out some statistics information
                         or otherwise do not (default not)
     --time-type=TYPE  use the specified time type when printing out
                         timings; TYPE can be one of: real, user or sys;
                         the default is real
```

```
      --debug=WHAT      print out some debugging information; WHAT can
                          be one of: probing
      --no-debug        do not print debugging info at all (default)
   -q|--[no-]quiet      be quiet or otherwise do not (default not)
   -v|--version         print version numbers and exit
   -?|--help            display this help info and exit
```

The **hash-trie** program takes input from standard input `stdin` – when its action option is one of '`-L|--load-only`', '`-p|--print-trie`' or '`-d|--dump-trie`'. When the action option is either '`-P|--print-config`' or '`-T|--print-types`', **hash-trie** is not taking input at all – it is only printing out something as requested. If input is to be read in, it is supposed to be formed by words – i.e. non-empty sequences of characters in range `[a-zA-Z]` – each placed on a separate line.

As already known from the previous section, any given build of **hash-trie** is governed by the following six configuration parameters:

```
$ grep -E -how 'CONFIG_[A-Z0-9_]+' hash-trie.cpp|sort -u
CONFIG_HASH_TRIE_ARRAY_BOUNDS
CONFIG_HASH_TRIE_CHAR_TYPE
CONFIG_HASH_TRIE_STATISTICS
CONFIG_HASH_TRIE_STRICT_TYPES
CONFIG_HASH_TRIE_TOLERANCE
CONFIG_HASH_TRIE_TRIE_SIZE
```

Also worthy of notice is that there exists the seventh governing parameter of **hash-trie**: namely `DEBUG`, which is defined by default and is not defined when compiling `hash-trie.cpp` with code optimizations enabled. The **gcc** compiler is told to enable the level NUM of code optimizations when invoking **make** with option '`OPT=`NUM'. The options '`--debug=WHAT`' and '`--no-debug`' of **hash-trie** are available for usage only when `DEBUG` was defined at the time of compilation of the program.

As a result of this condition, is easy to establish whether a given **hash-trie** binary was build with debugging code enabled or not: the count obtained from the **grep** command below is non-zero if and only if `DEBUG` is enabled:

```
$ make -B hash-trie OPT=0
gcc -std=gnu++11 -Wall -Wextra -DPROGRAM=hash-trie -O0 hash-trie.cpp -o hash-trie ↩
↪ -lstdc++
$ ./hash-trie -?|grep -c debug
0
$
```

All of the above `CONFIG_HASH_TRIE_*` parameters are orthogonal to each other, with only two exceptions:

- `STRICT_TYPES` implies `ARRAY_BOUNDS`;
- `TOLERANCE` and `TRIE_SIZE` are constraining each other by the internal logic of class `HashTrie<>` in the source file `hash-trie.cpp`. (Look-up the source code for compile-time assertions `CXX_ASSERT` applied to expressions containing or relating to the static constants `tolerance` and `trie_size`.)

The two parameters `STRICT_TYPES` and `ARRAY_BOUNDS` imply certain run-time supplemental checks on the types involved in the class `HashTrie<>`, each of these checks being made of run-time assertions of `SYS_ASSERT` kind. `SYS_ASSERT` is C++ preprocessing macro, which, however, is ruled out when the `DEBUG` parameter is not defined. Consequently, not defining `DEBUG` voids out all the supplemental restrictions the `STRICT_TYPES` and `ARRAY_BOUNDS` parameters are enforcing. There is also a positive side of this situation of `DEBUG` not being defined: the **`hash-trie`** program will improve significantly its runtime speed.

If **`hash-trie`** was built with all its configuration parameters having their default values, it would produce the following output when invoked for the action option '`-P|--print-config`':

```
$ make -B hash-trie
gcc -std=gnu++11 -Wall -Wextra -DDEBUG -DPROGRAM=hash-trie -g -gdwarf-3 hash-trie.cpp -o↩
↪ hash-trie -lstdc++
$ ./hash-trie -P
TOLERANCE:    1000
TRIE_SIZE:    32767
CHAR_TYPE:    char
STRICT_TYPES: no
ARRAY_BOUNDS: no
STATISTICS:   no
$
```

On a GNU/Linux 64-bit Intel x86 CPU machine, the output of **`hash-trie`** for action option '`-T|--print-types`' would look like:

```
$ ./hash-trie -T
op   operand         operator      wider               result
add  char            int           int                 int
add  signed char     int           int                 int
add  unsigned char   int           int                 int
add  short           int           int                 int
add  unsigned short  int           int                 int
add  int             int           long long           long long
add  unsigned int    unsigned int  unsigned long long  long long
sub  char            int           int                 int
sub  signed char     int           int                 int
sub  unsigned char   int           int                 int
sub  short           int           int                 int
sub  unsigned short  int           int                 int
sub  int             int           long long           long long
sub  unsigned int    unsigned int  unsigned long long  long long
```

Feeding **`hash-trie`** in with two words `a` and `ab`, it would print out the following text tables when the action option is '`-p|--print-trie`' and, respectively, '`-d|--dump-trie`':

```
$ print() { printf '%s\n' "$@"; }
$ print a ab|./hash-trie -p
a   1
ab  1
$ print a ab|./hash-trie -d
p      link[p]  ch[p]   sibling[p]  count[p]  word
0      0        header  26          0
1      20247    1       0           1         a
2      0        2       1           0         b
...    ...      ...     ...         ...       ...
25     0        25      24          0         y
26     0        26      25          0         z
...    .        .       .           .         .
20247  1        header  20249       0
...    .        .       .           .         .
20249  0        2       20247       1         ab
```

```
...    .        .        .            .         .
$
```

When invoked with action option `-L|--load-only`, **hash-trie** would suppress its normal output, printing out only error messages. This behaviour is useful for example in conjunction with options `-s|--print-stats` and `-q|--quiet`. The options `-s|--print-stats`, `--no-print-stats` and `--time-type` are available to the user of **hash-trie** only when the program was built with the STATISTICS configuration parameter enabled (as in the **make** invocation below).

```
$ make -B hash-trie CONFIG+=TOLERANCE=1 CONFIG+=TRIE_SIZE=55 CONFIG+=STATISTICS
gcc -std=gnu++11 -Wall -Wextra -DDEBUG -DCONFIG_HASH_TRIE_TOLERANCE=1 ↩
↪-DCONFIG_HASH_TRIE_TRIE_SIZE=55 -DCONFIG_HASH_TRIE_STATISTICS -DPROGRAM=hash-trie -g ↩
↪-gdwarf-3 hash-trie.cpp -o hash-trie -lstdc++
$ print|./hash-trie -L
hash-trie: error:1: hash tries cannot contain empty words
$ print _|./hash-trie -L
hash-trie: error:1: invalid input char '\x5f'
$ print a ab ac acd|./hash-trie -L
hash-trie: error:4: failed to put 'acd' in trie
$ print a ab ac acd|./hash-trie -L -s
hash-trie: error:4: failed to put 'acd' in trie
loaded-words:     3
failed-words:     1
duplicated-words: 0
empty-slots:      26
trie-time:        16
$ print a ab ac acd|./hash-trie -L -s -q
loaded-words:     3
failed-words:     1
duplicated-words: 0
empty-slots:      26
trie-time:        17
$
```

Note that the sole role of option `-q|--quiet` is to inhibit the error messages of type:

> ``hash-trie: error:LINE: failed to put 'WORD' in trie.''

4 References

[Kern81] Brian W. Kernighan. 1981. *Why Pascal is Not My Favorite Programming Language*. Technical Report 100, April, 1981. Computing Science, AT&T Bell Laboratories, Murray Hill, NJ, USA.

http://www.lysator.liu.se/c/bwk-on-pascal.html

[Liang83] Franklin Mark Liang. 1983. *Word hyphenation by computer*. Ph.D. Thesis, Technical Report STA-CS-83-977. Stanford University, Department of Computer Science, Stanford, CA, USA.

http://www.tug.org/docs/liang/liang-thesis.pdf

[BKM86] Jon Bentley, Donald E. Knuth and Doug McIlroy. 1986. Programming Pearls: a literate program. *Communications of the Association of Computing Machinery* 29, 6 (June 1986), 471-483.

[Knuth92] Donald E. Knuth. 1992. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University, Center for the Study of Language and Information, Stanford, CA, USA.

[Knuth98] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[Strou13] Bjarne Stroustrup. 2013. *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.