

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 <code>hash-trie.cpp</code> Source File	6
2.3	The Implementation of <code>HashTrie<></code> Class Template	11
3	The <code>hash-trie</code> 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 underlying system libraries
Ext	extensions of the standard C++ library
HashTrie	everything belonging to the <i>hash-trie</i> data structure
Main	the <code>main</code> 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 is 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 program¹ is revealed in §3 on page 152:

```

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

Upon looking deeper into the implementation, one can see that, the *global variables* constitute the main mechanism for communication between the parts – i.e. the *procedures* and the *functions* – 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 *procedures* and *functions* 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 dependencies – 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 its 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 **T_EX** 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 way 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:

```
1 { 9. Basic input routines. [...] a function
2   that reads an optional positive integer,
3   returning zero if none is present at the
4   beginning of the current line. }
5 function read_int: integer;
6 var n: integer; { the accumulated value }
7 begin
8   n := 0;
9   if not eof then
10    begin
11      while (not eoln) and (input^ = ' ') do
12        get(input);
13      while (input^ >= '0') and (input^ <= '9') do
14        begin
15          n := 10 * n + ord(input^) - ord('0');
16          get(input);
17        end;
18      end;
19      read_int := n;
20    end;
```

In this context, it is to be noted the interesting remark Knuth made³ 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:

```

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

```

1 { 15. We're now ready for the main input routine,
2   which puts the next word into the buffer. If no more
3   words remain, word_length is set to zero; otherwise
4   word_length is set to the length of the new word. }
5 procedure get_word;
6 label exit; { enable a quick return }
7 begin
8   word_length := 0;
9   if not eof then
10    begin
11      while lettercode[ord(input^)] = 0 do
12        if not eoln then
13          get(input)
14        else
15          begin
16            read_ln;
17            if eof then return;
18          end;
19      { Read a word into buffer (16) }
20      { 16. At this point lettercode[ord(input^)] > 0,
21        hence input^ contains the first letter of a word. }
22      repeat
23        if word_length = max_word_length then
24          word_truncated := true
25        else
26          begin
27            incr(word_length);
28            buffer[word_length] := lettercode[ord(input^)];
29          end;

```

```

30     get(input);
31     until lettercode[ord(input^)] = 0;
32 end;
33 exit:
34 end;

```

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

```

1 std::string input_word;
2 std::istream input_stream = ...;
3 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 $0 \leq \text{ord}(c) \leq 255$ whenever `c` is of type `char`.

```

1 { Global variables (4) += }
2 lowercase, uppercase: array [1..26] of char; { the letters }
3 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.

```

4 { Set initial values (12) }
5 lowercase[1] := 'a'; uppercase[1] := 'A';
6 ...
7 lowercase[26] := 'z'; uppercase[26] := 'Z';
8 for i := 0 to 255 do
9   lettercode[i] := 0;
10 for i := 1 to 26 do
11 begin
12   lettercode[ord(lowercase[i])] := i;
13   lettercode[ord(uppercase[i])] := i;
14 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 were 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:

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

from §18 on page 159:

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

from §24 on page 160:

```
7 define tolerance = 1000
```

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

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

Two of the defines 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-tries* 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 integers do not 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
<code>program</code> and <code>verdate</code> The sole global constants of the program. Define the name and the version number and date of the program. The constant <code>program</code> is defined by the build parameter <code>PROGRAM</code> – which is controlled by make .	const char[]
<code>global_options_t</code> Encapsulate all global option variables in one structure.	struct
<code>globals</code> The sole global variable. Give access to the global options.	<code>global_options_t</code>
<code>options_t</code> Parse command line options, update <code>globals</code> and provide the action option. These things are obtained upon invoking the sole publicly defined method of the class: static const <code>options_t options(int argc, char* argv[])</code> .	class
<code>print_config</code> Print out the configuration parameters of the program.	<i>function</i>
<code>print_op_types<></code> Print out the types corresponding to the named operation.	<i>function template</i>
<code>print_types</code> Print out the types of <i>add</i> and <i>sub</i> operations by calling <code>print_op_types<></code> .	<i>function</i>
<code>hash_trie_error</code> Print out formatted error messages and, if told to do so, exit the program.	<i>printf-like variadic function</i>
<code>hash_trie_t</code> <code>HashTrie::HashTrie<Sys::char_t></code> .	<i>type alias</i>
<code>print_func_t</code> <code>void (hash_trie_t::*)(Sys::ostream&) const.</code>	<i>type alias</i>
<code>exec_hash_trie</code> Load up words read from <code>stdin</code> in an instance of <code>hash_trie_t</code> and then call on that instance for the named print function, if given. When the <code>STATISTICS</code> configuration parameter was enabled at compile-time, if <code>globals.print_stats</code> says so, then also print out the statistics information of the instance.	<i>function</i>
<code>main</code> Dispatch the action option obtained from function <code>options_t::options</code> by calling the appropriate handler function: <code>print_config</code> , <code>print_types</code> or <code>exec_hash_trie</code> .	<i>the main function of hash-trie.cpp</i>

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

```

3429 int main(int argc, char* argv[])
3430 {
3431     const auto opt =
3432         options_t::options(argc, argv);
3433
3434     switch (opt.action) {
3435     case options_t::print_config:
3436         print_config();
3437         break;
3438     case options_t::print_types:
3439         print_types();
3440         break;
3441     case options_t::load_trie_only:
3442         exec_hash_trie();
3443         break;
3444     case options_t::print_trie:
3445         exec_hash_trie(
3446             &hash_trie_t::print);
3447         break;
3448     case options_t::dump_trie:
3449         exec_hash_trie(
3450             &hash_trie_t::dump);
3451         break;
3452     default:
3453         SYS_UNEXPECT_ERR(
3454             "action='%zu'",
3455             Ext::size_cast(opt.action));
3456     }
3457
3458     return 0;
3459 }

```

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

```

3394 using hash_trie_t = HashTrie::HashTrie<Sys::char_t>;
3395
3396 using print_func_t = void (hash_trie_t::*)(Sys::ostream&) const;
3397
3398 void exec_hash_trie(print_func_t print_func = nullptr)
3399 {
3400     size_t lno = 1;
3401     hash_trie_t trie;
3402     hash_trie_t::string_t str;
3403     while (Ext::getline(Sys::cin, str)) {
3404         try {
3405             if (!trie.put(str) && !globals.quiet)
3406                 hash_trie_error(
3407                     false, lno,
3408                     "failed to put '%"
3409                     SYS_CHAR_TYPE_FMTS
3410                     "s' in trie",
3411                     str.c_str());
3412         }
3413         catch (const HashTrie::Error& err) {
3414             hash_trie_error(
3415                 true, lno, "%s", err.what());
3416         }
3417         lno++;
3418     }
3419
3420     if (print_func)
3421         (trie.*print_func)(Sys::cout);
3422
3423 #ifdef CONFIG_HASH_TRIE_STATISTICS
3424     if (globals.print_stats)
3425         trie.print_stats(Sys::cout);
3426 #endif
3427 }

```


The namespace `Sys` contains system-related declarations and definitions:

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

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

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

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
<code>Ext::min<></code> , <code>Ext::max<></code> Provide handy shortcuts to the corresponding <code>std::numeric_limits<></code> functions.	constexpr function templates
<code>Ext::digits<></code> Provide a handy shortcut to the <code>std::numeric_limits<>::digits</code> constant.	constexpr function template
<code>Ext::size_cast<></code> Cast safely an integer of type narrower or equal width than <code>size_t</code> to <code>size_t</code> .	<i>function template: <code>size_t(T)</code></i>
<code>Ext::array<></code> Access C-style arrays via a range checking operator <code>[]</code> .	<i>function template: <code>Sys::array_t<T,N>(T(&)[N])</code></i>
<code>Ext::array_size<></code> Return the number of elements of C-style arrays.	constexpr function template: <code>size_t(T(&)[N])</code>
<code>Ext::getline<></code> Replace <code>std::getline<></code> on <code>std::basic_string<></code> with an optimized version. ⁶	<i>function template</i>
<code>Ext::box_t<></code> 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.	class template

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
<code>HashTrie::Error</code>	struct
Define the exception class used for signaling the few run-time error conditions that might occur in <code>HashTrie::HashTrie<></code> class. It is derived from <code>std::runtime_error</code> .	
<code>HashTrie::types_t</code>	enum
Name the four main types used by class <code>HashTrie::HashTrie<></code> : <code>count_type</code> , <code>pointer_type</code> , <code>letter_type</code> and <code>cell_type</code> . ⁷	
<code>HashTrie::types_traits_t</code>	struct template <code><typename T, types_t t></code>
Define the underlying integer type and the associated limits for the named type <code>t</code> .	
<code>HashTrie::add<></code>	function template
Define an addition operation, transparently over the boxed or just plain underlying unsigned integer types used by <code>HashTrie::HashTrie<></code> . Implement the operation such a way that the type of the resulting value be wide enough to avoid overflowing.	
<code>HashTrie::sub<></code>	function template
Define an subtraction operation, transparently over the boxed or just plain underlying unsigned integer types used by <code>HashTrie::HashTrie<></code> . Implement the operation such a way that the type of the resulting value be wide enough to avoid underflowing.	
<code>HashTrie::char_traits_t</code>	struct template <code><typename C = char></code>
Define the default traits class template associated to the character type used by <code>HashTrie::HashTrie<></code> . ⁸	
<code>HashTrie::size_traits_t</code>	struct
Encompass a few static parameters of class <code>HashTrie::HashTrie<></code> : <code>max_count</code> , <code>trie_size</code> and <code>tolerance</code> . The last two of these are defined by corresponding <code>CONFIG_HASH_TRIE_*</code> configuration parameters.	
<code>HashTrie::HashTrie<></code>	class template
Implement the main functionality of the <i>hash-trie</i> 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 `types_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
<i>unnamed</i>	<code>0..max_count</code>	<code>count_type</code>
<code>pointer</code>	<code>0..trie_size</code>	<code>pointer_type</code>
<i>unnamed</i>	<code>1..26</code>	<code>letter_type</code>
<i>unnamed</i>	<code>empty_slot..header</code>	<code>cell_type</code>

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 definitions 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*, *C RTP*).

```

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;
1710     ...
1711 };
1712
1713 template<typename T>
1714 struct types_traits_t<T, pointer_type>
1715 {
1716     typedef unsigned short base_t;
1717     #if defined(CONFIG_HASH_TRIE_STRICT_TYPES)
1718         typedef Ext::box_t<
1719             base_t, types_traits_t<T, pointer_type>> type_t;
1720     #else
1721         typedef base_t type_t;
1722     #endif
1723 
```

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.

```

1724 #else
1725     typedef base_t type_t;
1726 #endif
1727
1728     // pointer_t: [0..trie_size]
1729     static constexpr size_t min = 0;
1730     static constexpr size_t max = T::trie_size;
1731     ...
1736 };
1737
1738 template<typename T>
1739 struct types_traits_t<T, letter_type>
1740 {
1741     typedef unsigned char base_t;
1742     #if defined(CONFIG_HASH_TRIE_STRICT_TYPES)
1743         typedef Ext::box_t<
1744             base_t, types_traits_t<T, letter_type>> type_t;
1745     #else
1746         typedef base_t type_t;
1747     #endif
1748
1749     // letter_t: [min_letter..max_letter]
1750     static constexpr size_t min = T::min_letter;
1751     static constexpr size_t max = T::max_letter;
1752     ...
1757 };
1758
1759 template<typename T>
1760 struct types_traits_t<T, cell_type>
1761 {
1762     typedef unsigned char base_t;
1763     #if defined(CONFIG_HASH_TRIE_STRICT_TYPES)
1764         typedef Ext::box_t<
1765             base_t, types_traits_t<T, cell_type>> type_t;
1766     #else
1767         typedef base_t type_t;
1768     #endif
1769
1770     // cell_t: [empty_slot..header]
1771     static constexpr size_t min = T::empty_slot;
1772     static constexpr size_t max = T::header;
1773     ...
1778 };

```

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 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 dependencies 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.

```

1899 template<typename C = char>
1900 struct char_traits_t
1901 {
1902     typedef
1903         typename
1904             std::remove_cv<C>::type
1905         char_t;
1906     typedef
1907         Ext::unsigned_t<char_t>
1908         uchar_t;
1909     typedef
1910         std::char_traits<char_t>

```

```

1911     traits_t;
1912     typedef
1913         Ext::unsigned_t<
1914             typename traits_t::int_type>
1915         uint_t;
1916     typedef
1917         size_t code_t;
1918     typedef
1919         std::basic_string<char_t>
1920         string_t;
...
1930     static constexpr uint_t to_int(char_t ch)
1931     { return traits_t::to_int_type(ch); }
1932
1933     static constexpr char_t to_char(uint_t uint)
1934     { return traits_t::to_char_type(uint); }
1935
1936     static constexpr char_t to_lower(char_t ch);
...
1943     static constexpr bool is_valid_char(char_t a)
...
1948     static constexpr bool is_valid_code(code_t a)
...
1953     static code_t to_code(char_t c)
...
1967     static char_t from_code(code_t c)
...
1977     static constexpr code_t min_letter = 1;
1978     static constexpr code_t max_letter = min_letter +
...
1981     static constexpr code_t header      = max_letter + 1;
1982     static constexpr code_t empty_slot = 0;
1983 };

```

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

```

1999 struct size_traits_t
2000 {
2001     static constexpr size_t max_count = 32767;
2002     static constexpr size_t trie_size = CONFIG_HASH_TRIE_TRIE_SIZE; // = 32767;
2003     static constexpr size_t tolerance = CONFIG_HASH_TRIE_TOLERANCE; // = 1000;
2004 };

```

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`.

```

2006 template<
2007     typename C = char,
2008     template<typename> class T = char_traits_t,
2009     typename S = size_traits_t>
2010 class HashTrie :
2011     private T<C>,
2012     private S
2013 {
2014 public:
2015     typedef S size_traits_t;
2016     typedef T<C> char_traits_t;

```

```

2017     typedef typename char_traits_t::char_t char_t;
2018     typedef typename char_traits_t::string_t string_t;
2019
2020     HashTrie();
2021
2022     bool put(const char_t*);
2023
2024     bool put(const string_t& str)
2025     { return put(str.c_str()); }
2026
2027     void print(std::basic_ostream<char_t>&) const;
2028     void dump(std::basic_ostream<char_t>&) const;
2029
2030 #ifdef CONFIG_HASH_TRIE_STATISTICS
2031     void print_stats(std::basic_ostream<char_t>&) const;
2032 #endif
2033
2034 private:
2035     ...
2036     ...
2037     template<types_t t>
2038     using types_traits_t = types_traits_t<HashTrie, t>;
2039
2040     using size_traits_t::max_count;
2041     using size_traits_t::trie_size;
2042     using size_traits_t::tolerance;
2043     using char_traits_t::min_letter;
2044     using char_traits_t::max_letter;
2045     using char_traits_t::empty_slot;
2046     using char_traits_t::header;
2047
2048     ...
2049     ...
2106     template<types_t i, types_t e>
2107     struct array_of_t
2108     {
2109         typedef
2110             typename types_traits_t<e>::type_t
2111             elem_t;
2112
2113         class type_t
2114         {
2115         public:
2116             static constexpr size_t lo =
2117                 types_traits_t<i>::min;
2118             static constexpr size_t hi =
2119                 types_traits_t<i>::max;
2120
2121         private:
2122             ...
2123             ...
2124             elem_t array[hi - lo + 1];
2125         };
2126     };
2127
2128     typedef typename types_traits_t<count_type>::type_t count_t;
2129     typedef typename types_traits_t<pointer_type>::type_t pointer_t;
2130     typedef typename types_traits_t<letter_type>::type_t letter_t;
2131     typedef typename types_traits_t<cell_type>::type_t cell_t;
2132
2133     typedef
2134         typename array_of_t<pointer_type, count_type>::type_t
2135         count_array_t;
2136     typedef
2137         typename array_of_t<pointer_type, pointer_type>::type_t
2138         pointer_array_t;
2139     typedef
2140         typename array_of_t<pointer_type, cell_type>::type_t
2141         cell_array_t;
2142
2143     cell_array_t ch;
2144     count_array_t count;

```

```

2227     pointer_array_t link;
2228     pointer_array_t sibling;
2229
2230     //  $x_n = (\alpha * n) \% \text{mod\_x}$ 
2231     pointer_t x;
2232 ...
2240     static letter_t to_letter(char_t c)
2241     { return char_traits_t::to_code(c); }
2242
2243     static char_t to_char(letter_t l)
2244     { return char_traits_t::from_code(l); }
2245 ...
2249     pointer_t find(const char_t*);
2250
2251     void print_word(pointer_t, std::basic_ostream<char_t>&, bool) const;
2252     void dump_word(pointer_t, std::basic_ostream<char_t>&, bool&) const;
2253     void dump_empty(std::basic_ostream<char_t>&) const;
2254
2255     static constexpr size_t make_alpha(size_t trie_size, size_t max_letter)
2256     ...
2259     { return std::ceil(0.61803 * (trie_size - 2 * max_letter)); }
2260     ...
2266     static constexpr size_t alpha = make_alpha(trie_size, max_letter);
2267     static constexpr size_t mod_x = trie_size - 2 * max_letter;
2268     static constexpr size_t max_h = mod_x + max_letter;
2269     static constexpr size_t max_x = mod_x - alpha;
2270 ...
2281 #ifdef CONFIG_HASH_TRIE_STATISTICS
2282     struct stats_t
2283     {
2284         size_t loaded_words = 0;
2285         size_t failed_words = 0;
2286         size_t duplicated_words = 0;
2287         size_t empty_slots = 0;
2288
2289         Sys::clocks_t trie_time = {};
2290     ...
2292     };
2293
2294     mutable stats_t stats;
2295 #endif
2296 };

```

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 overloaded 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 `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]:

```
2298 template<
2299     typename C,
2300     template<typename> class T,
2301     typename S>
2302 HashTrie<C, T, S>::HashTrie()
2303 {
2304     ch.assign(header, trie_size, empty_slot);
2305     link.assign(1, max_letter, 0);
2306     count.assign(1, max_letter, 0);
2307
2308     for (pointer_t i = 1; i <= max_letter; ++ i) {
2309         ch[i] = i;
2310         sibling[i] = i - 1;
2311     }
2312
2313     link[0] = 0;
2314     count[0] = 0;
2315     ch[0] = header;
2316     sibling[0] = header - 1;
2317
2318     x = 0;
2319 }
```

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 input¹², increments the counter associated to the input word, if any, and, when `CONFIG_HASH_TRIE_STATISTICS` is `#defined`, updates its `stats` structure:

```
2644 template<
2645     typename C,
2646     template<typename> class T,
2647     typename S>
2648 bool HashTrie<C, T, S>::put(const char_t* str)
2649 {
2650     if (*str == '\0')
2651         Sys::error<Error>("hash tries cannot contain empty words");
2652 }
```

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_ceil`.

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

```

2653 #ifndef CONFIG_HASH_TRIE_STATISTICS
2654     Sys::utime_t time;
2655 #endif
2656
2657     // 34. Input the text, maintaining a dictionary with frequency count
2658     auto p = find(str);
2659
2660 #ifndef CONFIG_HASH_TRIE_STATISTICS
2661     stats.trie_time += time();
2662
2663     if (p)
2664         stats.loaded_words ++;
2665     else
2666         stats.failed_words ++;
2667     if (p && count[p])
2668         stats.duplicated_words ++;
2669 #endif
2670
2671     if (!p) return false;
2672     count[p] ++;
2673     return true;
2674 }

```

The method `find` of `HashTrie<>` class constitutes the central part of Knuth's *hash-trie* data structure. It substantiates the core algorithm 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]:

```

2321 template<
2322     typename C,
2323     template<typename> class T,
2324     typename S>
2325 typename
2326     HashTrie<C, T, S>::pointer_t
2327     HashTrie<C, T, S>::find(const char_t* str)
2328 {
2329     ...
2330     ...
2445     const pointer_t tolerance2 = tolerance;
2446     // trial header location
2447     pointer_t h;
2448     // the final one to try
2449     pointer_t last_h; // INT: int last_h;
2450
2451     const auto get_set_for_computing_header_locations = [&]() {
2452         // 24. Get set for computing header locations
2453         ...
2454         ...
2477         if (x >= max_x)
2478             x -= max_x;
2479         else
2480             x += alpha;
2481
2482         h = x + max_letter + 1; // now max_letter < h <= trie_size - max_letter
2483         ...
2484         ...
2506         if (h > max_h - tolerance) {
2507             ...
2508             ...
2523             last_h = add(h, tolerance2) - mod_x;
2524         }
2525         else {
2526             ...
2527             ...
2533             last_h = h + tolerance;
2534         }
2535     };
2536
2537     const auto compute_the_next_trial_header_location = [&]() {
2538         // 25. Compute the next trial header location h, or abort find
2539         ...
2540         ...
2545         if (h == last_h)
2546             return false;

```

```

2547         if (h == max_h)
2548             h = max_letter + 1;
2549         else
2550             h ++;
2551         return true;
2552     };
2553
2554     // the current word position
2555     pointer_t p
2556         = to_letter(*str ++);
2557     while (*str) {
2558         // current letter code
2559         letter_t c
2560             = to_letter(*str ++);
2561         // 21. Advance p to its child number c
2562         if (link[p] == 0) {
2563             // 27. Insert the firstborn child of p and move to it, or abort find
2564             get_set_for_computing_header_locations();
2565             ...
2566             do {
2567                 if (!compute_the_next_trial_header_location())
2568                     return 0;
2569                 ...
2570             } while (ch[h] != empty_slot || ch[h + c] != empty_slot);
2571             ...
2572             link[p] = h;
2573             link[h] = p;
2574             p = h + c;
2575             ch[h] = header;
2576             ch[p] = c;
2577             sibling[h] = p;
2578             sibling[p] = h;
2579             count[h] = 0;
2580             count[p] = 0;
2581             link[p] = 0;
2582         }
2583         else {
2584             // the next word position
2585             pointer_t q
2586                 = link[p] + c;
2587             if (ch[q] != c) {
2588                 if (ch[q] != empty_slot) {
2589                     // 29. Move p's family to a place where child c will fit,
2590                     //      or abort find
2591                     // family member to be moved
2592                     pointer_t r;
2593                     // amount of motion
2594                     // INT: int delta;
2595                     // have we found a new homestead?
2596                     bool slot_found;
2597                     // 31. Find a suitable place h to move, or abort find
2598                     slot_found = false;
2599                     get_set_for_computing_header_locations();
2600                     ...
2601                     do {
2602                         if (!compute_the_next_trial_header_location())
2603                             return 0;
2604                         ...
2605                         if (ch[h + c] == empty_slot) {
2606                             r = link[p];
2607                             auto delta = sub(h, r); // INT: delta = h - r;
2608                             while (ch[r + delta] == empty_slot &&
2609                                 sibling[r] != link[p])
2610                                 r = sibling[r];
2611                             slot_found = ch[r + delta] == empty_slot;
2612                         }
2613                     } while (!slot_found);
2614                     ...

```

```

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 hundred 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 The Pascal code declares the type of last_h to be integer. The C++ code declares this variable to be of type pointer_t.	p. 161, §26, #3	p. 18, #2449
assignment to last_h 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".	p. 161, §24, #7	p. 18, #2523
declaration of delta 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.	p. 162, §30, #3	p. 19, #2594
1st assignment to delta The Pascal code assigns to delta by the statement "delta = h - r". The C++ code replaces this assignment with " auto delta = sub(h, r)".	p. 162, §31, #5	p. 19, #2607
2nd assignment to delta The same difference as in the case of the 1st assignment to delta.	p. 162, §29, #3	p. 20, #2617

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 certain 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_t`s `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 adjustment 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 chosen such that to avoid the occurrence of underflow 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          a
· 1      20247    1       0          1          b
· 2      0        2       1          0          y
· ...    ...      ...      ...         ...         z
· 25     0        25      24         0          .
· 26     0        26      25         0          .
· ...    .        .       .           .           .
· 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.

Copyright © 2016 **Ştefan Vargyas**

This file is part of **Hash-Trie**.

Hash-Trie is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Hash-Trie is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with **Hash-Trie**. If not, see <http://www.gnu.org/licenses/>.