Identifier character sequence reuse

Discussion and proposed guidelines

Derek M. Jones
derek@knosof.co.uk

1 Introduction

When reading identifiers in source code developers need to remember information about them for later use or to recognise previously read identifiers and recall information about them. Experience shows that developers regularly make incorrect associations between a reference to an identifier and its declaration. The reasons for these incorrect associations include lack of knowledge (i.e., the developer did not know that there was more than one possible association) and various forms of cognitive error (i.e., visually misreading the identifier).

The two obvious guideline recommendations are to require that all identifiers declared in a program be use different character sequences and that all character sequences used be sufficiently different from each other that cognitive errors are minimised.

While guidelines are intended to provide benefits, they are also likely have costs associated with their use. This proposal looks at the issues that are believed to be the root cause of some identifier related faults and discusses their costs and benefits. The issues can be divided into those that are computer language related and those that are derived from how people process character sequences.

- Computer language issues, e.g., scope, namespace, linkage, storage duration.
- Human processing issues, e.g., visual (looks like), sounds like, semantics (closely associated with), mechanical (typing mistakes).

The issues discussed in this proposal are also applicable when choosing the names of sources files.

Some suggested guideline recommendations and deviations from which language specific recommendations can tailored are given.

The material in this proposal has been distilled from that appearing in sentence 787 of "The New C Standard: An economic and cultural commentary" by Derek M. Jones (available via www.knosof.co.uk/cbook/cbook.html).

2 Same character sequence, different identifier

Programming languages provide a range of mechanisms that allow declarations of different identifiers, having the same character sequence, to exist in the source code of a program. The following are three possibilities supported by existing languages:

- Scope. The declaration of an identifier (let's call it the second declaration) causes references that would have referred to the identifier declared in another declaration to refer to this second declaration. The second declaration is said to *hide* the identifier declared in the other declaration.
- Block structure. The syntactic contexts in which two identifier declarations occur is such that the subsets of the source over which the identifiers may be referenced is textually disjoint (i.e., removing either declaration does not cause a reference to refer to an identifier declared in a different declaration).
- The semantic attributes given to an identifier by its respective declaration is such that the localised syntactic context decides which declaration a reference refers to (e.g., in most languages an identifier appearing to the right of a **`goto`** token can only refer to a label).

The term *visibility* is commonly used in developer discussions on where an identifier can and cannot be referenced (i.e., is or is not visible).

Minimising the visibility of an identifier has a number of benefits and programming languages provide various mechanisms that give developers some control over identifier visibility. For instance, one mechanism (which is often given the name *scope*) is using the kind of statement or declaration within which an identifier is declared. For instance, an identifier declared within a *block* is only visible within that block.

2.1 Cost of reusing character sequences

A consequence of identifier visibility being a subset of a program's source code is that it is possible to use the same character sequence to declare identifiers that look the same but are different. Compilers do not have any problem deciding which declaration an occurrence of an identifier refers to, however experience shows that developers are not so perfect; mistakes are made which result in unintended program behavior.
The following are three possible root causes of mistakes that can be made when different identifiers share the same character sequence include:

- Deleting a declaration results in identifiers that used to refer to it now referring to a different declaration. An example is when an identifier declared in an inner scope has the same name as an identifier declared in an outer scope:

```
1  int abc_xyz;
2
3      {
4      int abc_xyz;
5
6      ...
7      abc_xyz=3;
8
9      ...
10     }
```

  When two identifiers with the same name exist in overlapping scopes, it is possible for a small change to the source to result in a completely unexpected change in behavior. For instance, if the identifier definition in the inner scope is deleted, all references that previously referred to that, inner scoped, identifier will now refer to the identifier in the outer scope. Deleting the declaration of an identifier is usually intended to be accompanied by deletions of all references to it. The omission of a deletion will only generate a diagnostic, when the source is translated, if the attributes of the identifier differ in some significant way (e.g., the identifiers denote objects having incompatible types). The term *unexpected visibility* might be used to describe this usage. The minimum requirement for this kind of mistake to go undetected is that the translator successfully translate the source. That is the the attributes associated with the identifier declared in the inner scope must have some degree of compatible with the those of the identifier declared in the outer scope.
- Adding a declaration causes additional identifiers to become visible. These identifiers are sometimes said to have been *injected*. An example is argument dependent lookup in C++...
- A developer recalls information about an identifier that does not apply to a particular reference to an identifier.

```
1  /*
2   * Return true if the point spot has been hit.
3   */
4  Bool hit_the_spot(int spot);
5
6  /*
7   * Return true if 'attempt' will hit the spot
8   */
9  #define HIT_THE_SPOT(attempt) ...
10
11 /* ... */
```

  It is possible that developers will incorrectly swap the actions performed by the above function and macro, resulting in one being invoked when the other had been intended.

When considering language specific guidelines it is necessary to consider all of the ways in which identifier declarations can hide previously referenced declarations...

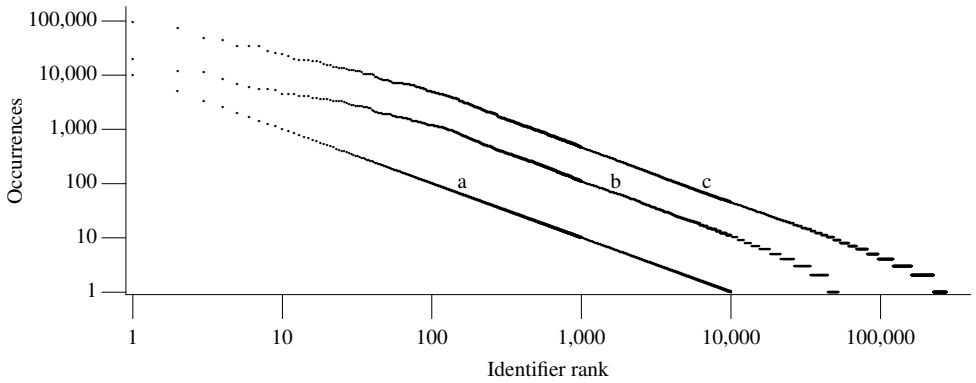

Figure .1: Identifier rank (based on frequency of occurrence of identifiers having a particular spelling) plotted against the number of occurrences of the identifier in the visible source of (b) Mozilla, and (c) Linux 2.4 kernel; (a) is a distribution following Zipf's law with the most common item occurring 10,000 times. Every identifier is represented by a dot.

LangSpec .1 Language specific guidelines shall document all constructs that enable the declaration of an identifier to hide the declaration of another identifier.

LangSpec .2 Language specific guidelines shall document all constructs where the declaration of an identifier may make visible other identifiers that do not appear in that declaration.

2.1.1 How often do these kinds of mistakes occur in practise?

For these mistakes to go undetected it is necessary that they do not render the source containing them untranslatable. So, for instance, if the first declaration of `abc_xyz` in the earlier example had given it a pointer type, then the assignment of the value 3 would have been diagnosed during compilation.

There are some cases where it is not possible to reference different identifiers that share the same character sequence. For instance, a reference to a member of structure is preceded by the name of an object having the appropriate structure type.

No empirical data on the number of mistakes made by developers, when processing different identifiers having the same character sequences, is available and is unlikely to be available in the near future.

2.2 Benefit of reusing character sequences

Developers need to remember information about identifiers in order to comprehend source code. One technique for reducing the effort needed to remember this information is to make use of a developer's existing knowledge of character sequences (i.e., human language words) and ability to recognise and create new character sequences from existing ones (e.g., adding "ed" to specify the past tense, or forming abbreviations from known words).

The semantic associations triggered by an identifier's character sequence is believed to play a significant role in a readers recall of the coding related information associated with that identifier (another factor is the syntactic context in which the identifier occurs)...

To what extent do developers reuse the same character sequence within a single program? Measurements on reuse of the same character sequence in declarations...

Members in different structure types may represent the same kind of information; for instance, a member named `next` might always point to the next element in a linked list, its type being a pointer to the structure type containing its definition. Members named `x_coord` and `y_coord` might occur in several structure types dealing with coordinate systems.

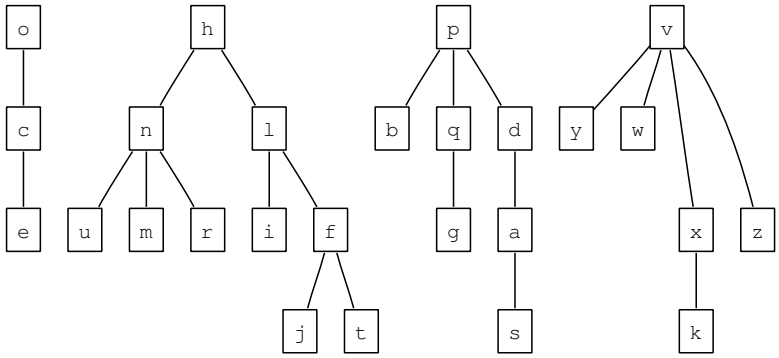

Figure .2: Similarity hierarchy for English letters. Adapted from ###.[?]

3 Different character sequences, same identifier?

Human error can result in a developer incorrectly treating one sequence of characters as the same as a different sequence of characters, or a developer recalling information about an identifier declaration that actually applies to an identifier declared using different character sequence.

While people might make any number of different kinds of error for a wide variety of reasons, it is likely that the root cause of most errors will be traceable to those cognitive processes involved in handling character sequences. The human processing issues considered here are: visual (looks like), sounds like, semantics (closely associated with), and mechanical (typing mistakes).

Detailed data on the kinds of errors made by developers, when processing character sequences, is not available and is unlikely to be available in the near future. The following discussion uses an analysis of the cognitive processes involved in processing character sequences to theoretically deduce the errors that developers are most likely to make. The intent is to use this theory to suggest recommendations that will minimise the character sequence processing errors made by developers. However, different cognitive systems sometimes have conflicting requirements for error minimisation, so tradeoffs have to be made.

Identifiers are not written purely for the benefit of the compiler. It is expected that they will be processed in a variety of ways by the developers who read the source code that contains them. In order to make use of identifiers developers need to store and retrieve information about them, including the character sequence used to denote them. Studies have found that sound and semantics (i.e., meaning) are the two main representations used by humans to store and retrieve information. Use of the written word is a recent invention and there is unlikely to have been any evolutionary pressure favoring the passing on of genes that improved peoples' reading and writing abilities.

People make use of the cognitive abilities they have, that evolved for non-written word tasks, to comprehend source code. These co-opted, existing, cognitive abilities are likely to have their own impact on the kind of mistakes made by developers. For instance, developers don't have a memory system whose operation is directly based on character sequences, but rather make use of one that is based on sound representation (two methods of storing character sequences involve storing the sound of the word they form and storing the sound of individual characters (Kanji???)), therefore it is to be expected that the root cause of some mistakes will be found in the mechanisms used by people to process sound sequences.

3.1 Seeing (reading) character sequences

Visual similarity between different characters can result in a character sequence being treated as identical to a different character sequence. What are the factors that cause a character sequence to be treated as identical to a different character sequence? Research has found that the following are some of the major factors:

- Visible appearance of individual characters. The extent to which two characters have a similar visual appearance is affected by a number of factors, including the orthography of the language, the font used

to display them, and the method of viewing them (e.g., print vs. screen). Examples of two extremes of similarity (based on commonly used fonts) are the characters `1` (one) and `l` (ell), which are very similar, and the characters `T` and `w` which are not visually similar. The greater the visual similarity between two characters, the greater the probability that one of them will be mistakenly treated as the other. One study[4] found that subjects ability to detect spelling errors in text (after adjusting for letter frequency, word frequency, and perceived word similarity) was correlated with measurements of individual letter confusability. Studies of letter similarity measurement have a long history.[3, 10] These studies have used letters only— there are no statistically significant published results that include digits or common mathematical symbols (as occur in programming language source code).

- Word shape. The term *whole word shape* refers to the complete visual appearance of the sequence of letters used to form a word. Some letters have features that stand out above (ascenders— *f* and *t*) and below (descenders — *q* and *p*) the other letters, or are written in uppercase.
- Every natural language has patterns to the way it joins sounds together to form words. This in turn leads to patterns (at least for nonlogographic orthographies) in the character sequences seen in the written form (even in an irregularly spelled language such as English). Proficient users of a language have overlearned these patterns and can effortlessly recognise them. While novice readers may read words a letter at a time, experts make use of their knowledge of commonly occurring character sequences (which may be complete words) to increase their reading rate. The penalty for making use of statistical information is an increased likelihood of making mistakes, particularly when reading character sequences that are not required to be words (e.g., identifiers).

```
1  enum {OO, O0, l1, ll} glob;
```

Word recognition is driven by both bottom-up processes (the visible characters) and top-down processes (reader expectations). Reducing the visual similarity between different character sequences is one technique for reducing the likelihood of a reader mistakenly treating one identifier for another, different, identifier. While the data needed to accurately calculate the visual similarity between two identifiers is not yet available, a guideline recommendation may still be considered worthwhile.

3.2 Hearing (memory for) character sequences

This subsection discusses storage and retrieval of information about identifier character sequences by developers.

When reading source developers convert the character sequence denoting an identifier in to a sequence of sounds. It is this sound sequence whose representation is stored in memory. A memory system that stores information about character sequences using a sound based representation has to make use of algorithms that convert character sequences to sound sequences and sound sequences to character sequences. In many human languages the mapping is not unique. That is there are often many ways of mapping a sound sequence to a character sequence (e.g., `foto`, `photo`, `fotto`, `photoe`, `fotow`); the case of non-unique mappings of a character sequence to a sound sequence is not of interest here.

Various semantic attributes about the occurrence of an identifier may also be noted and stored. Human memory does not remember everything that it is asked to store and the information it holds becomes more difficult to retrieve over time (this may be due to a degradation of the original memory or the paths leading to that memory).

Two of the reasons for needing to retrieve information from human memory include:

- *recall*: this might involve information about an identifier being presented and a developer having to recall the name of the identifier, and
- *recognition*: this might involve the name of an identifier being presented and a developer recognising it as having been seen before. Having recognised an identifier, information about the identifier may be recalled.

and...

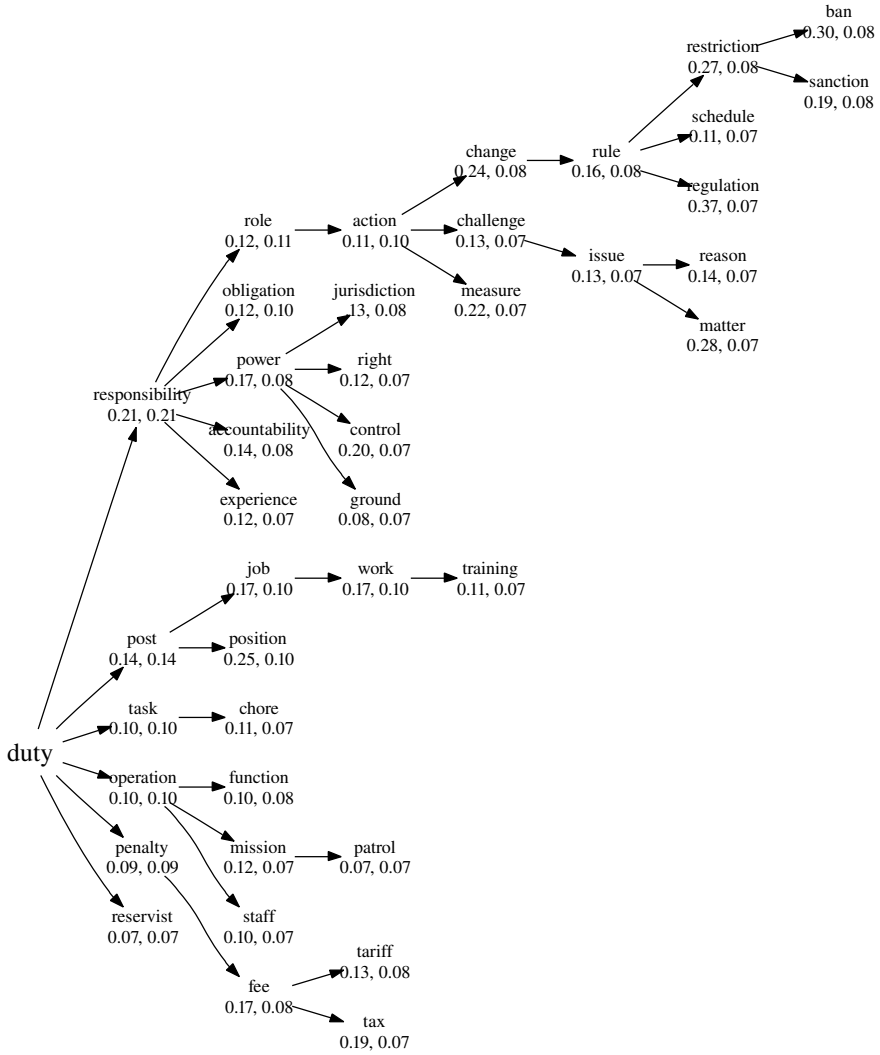

Figure .3: Semantic similarity tree for *duty*. The first value is the computed similarity of the word to its parent (in the tree), the second value its similarity to *duty*. Adapted from Lin.[5]

3.3 Semantic associations

Developers are often exhorted to use "meaningful identifiers". Meaningfulness requires that knowledge already exist in a developer's memory. Making use of existing knowledge reduces the amount of new information a developer needs to remember and reduces the probability that information will be lost (because the semantic associations are already in long term memory and are likely to be recalled in other circumstances {access to memories tends to be impaired if they are not recalled for long periods of time}). The disadvantage of using existing knowledge is that subsequent developers (including the one who selected the original character sequence) may not always assign the same meaning to a character sequence.

The term *same semantic associations* is somewhat ill-defined and subject to interpretation (based on the cultural background and education of the person performing the evaluation, an issue that is discussed in more detail elsewhere).

Some identifiers are formed by concatenating two or more known words, abbreviations, or acronyms (these

subcomponents are called *conceptual units* here). The interpretation given to a sequence of these conceptual units, by a developer, may depend on their relative ordering. For instance, `widget_num` or `num_widget` might be considered, by native English speakers, to denote a number assigned to a particular widget and some count of widgets, respectively. Such an interpretation is dependent on knowledge of English word order and conventions for abbreviating sentences (e.g., "widget number 27" and "number of widgets"). This distinction is subtle and relies on a very fine a point of interpretation which could quite easily be given the opposite interpretation.

To what extent is it permissible for developers to use a semantic interpretation of a character sequence that depends on detailed knowledge of the native language from which the conventions used came???

3.4 Typing performance

A developer may accidentally press the wrong key when entering a character sequence. This mistake may go undetected because the character sequence typed matches that of a visible declared identifier.

The results of typing studies show that the two factors of relevance to identifier guideline recommendations are, 1) hitting a key adjacent to the correct one is the largest single contributor (around 35%) to the number of typing mistakes made by people, and 2) performance has been found to be affected by the characteristics of the typists native written language (e.g., word frequency and morphology).

Researchers studying typing often use skilled typists as subjects (and build models that mimic such people[8]). These subjects are usually asked to make a typewritten copy of various forms of prose (the kind of task frequently performed by professional typists) the time taken and errors made being measured. Software developers are rarely skilled typists and rarely copy from written material. (It is often created in the developer's head on the fly, and a theory of developer typing performance would probably need to consider these two processes separately.) One study,[9] using qualified typists, found that text containing words created using purely random letter sequences had the highest rate of typing mistakes (and the slowest typing rate), almost twice that of text created using the distribution of letters found in English.

These coding guidelines assume that developers' typing mistakes will follow the same pattern as those of typists, although the level of performance may be lower. It is also assumed that the primary input device will be a regular-size keyboard and not one of those found on mobile computers.[7]???

4 Filename character sequences

Just like identifiers, filenames require developers too come up with a character sequence. Measurements have found that these character sequences share many of the characteristics of identifiers (e.g., use of abbreviated words[1] and code performing related tasks being kept in files with names that share related semantic attributes[2]).

Development groups often adopt naming conventions for source file names. Source files associated with implementing particular functionality have related names, for instance:

1. Data manipulation: *db* (database), *str* (string), or *queue*.
2. Algorithms or processes performed: *mon* (monitor), *write*, *free*, *select*, *cnv* (conversion), or *chk* (checking).
3. Program control implemented: *svr* (server), or *mgr* (manager).
4. The time period during which processing occurs: *boot*, *ini* (initialization), *rt* (runtime), *pre* (before some other task), or *post* (after some other task).
5. I/O devices, services or external systems interacted with: *k2*, *sx2000*, (a particular product), *sw* (switch), *f* (fiber), *alarm*.
6. Features implemented: *abrvdial* (abbreviated dialing), *mtce* (maintenance), or *edit* (editor).
7. Names of other applications from where code has been reused.
8. Names of companies, departments, groups or individuals who developed the code.

9. Versions of the files or software (e.g., the number 2 or the word *new* may be added, or the name of target hardware), different versions of a product sold in different countries (e.g., *na* for North America, and *ma* for Malaysia).
10. Miscellaneous abbreviations, for instance: *utl* (utilities), or *lib* (library).

5 Possible recommendations

What is the best way of doing this???

- One guideline + deviations, or
- many guidelines and no deviations.

The recommendations in this subsection are driven by the characteristics of developers, not compilers. This means that all characters in the visible character sequence of an identifier are significant, even if the compiler used would only consider some of them as significant (for the purposes of its own processing).

Cg .3 When performing similarity checks on identifiers, all characters shall be considered significant.

Same rules for identifiers and filenames?
Filenames are a much smaller class of character sequences, compared to identifiers. Does this make it more cost effective for guidelines to have a greater say over the sequences that shall be used? Are the naming benefits the same as for identifiers, if so why spend more on controlling the character sequences (just because we can???)

5.1 Comparing costs and benefits

No empirical data on the number of mistakes made by developers, when processing different identifiers having the different character sequences, is available and is unlikely to be available in the near future.

5.2 Syntactic context

Syntactic context is a source of information available to developers when reading source code. To what extent does syntactic context help resolve the (perhaps) multiple semantic associations that occur in a developer's head when they see a character sequence?
Macros provide a mechanism for hiding the syntactic context from readers of the visible source; for instance, the `offsetof` macro takes a tag name and a member name as arguments. This usage is rare and is not discussed further here.

Table .1: Identifiers appearing immediately to the right of the given token as a percentage of all identifiers appearing in the visible source. An identifier appearing to the left of a `:` could be a label or a **`case`** label. However, C syntax is designed to be parsed from left to right and the presence, or absence, of a **`case`** keyword indicates the entity denoted by an identifier.

Token	`.c` file	`.h` file	Token	`.c` file	`.h` file
`goto` identifier	99.9	100.0	**`struct`** identifier	99.0	88.4
`#define` identifier	99.9	100.0	**`union`** identifier	65.5	75.8
`.` identifier	100.0	99.8	**`enum`** identifier	86.6	53.6
`->` identifier	100.0	95.5	**`case`** identifier	71.3	47.2

In some languages some syntactic contexts are optional. For instance, in Pascal a **`with`** statement can be used to remove the need to specify one or more identifiers in a field selection (e.g., `a.b` might be equivalently written as `b` if it appears in the scope of a `with a`). Cobol allows identifiers to be omitted in a field selection if it is possible for the compiler to unambiguously resolve which object is being referred to.

5.3 Specify permissible character sequences

Why should developers be given the freedom to choose which character sequence should be used to denote an identifier?

Some organizations have decided that an acceptable solution to the confusion and increased cost of sharing code and data caused by having having developers make their individual choice on which character sequence to use for an identifier, is to fix the allowable character sequences in advance (i.e., developers are required to use a character sequence that comes from a controlled vocabulary). For instance, US law enforcement agencies experienced great difficulty in sharing data because different database developers have used different column names to represent the same kind of data. The column names that are to be used across all law enforcement databases is now being decided on (see the Global Justice XML data dictionary at the Justice Standards Clearinghouse it.ojp.gov/jsr/public/).

There is an ISO Standard that deals with naming in a database context: "ISO/IEC 11174 Information technology — Specification and standardization of data elements."

These guidelines cover the use of identifiers within the source of a single program. Developers often work on more than one program and knowledge of character sequences contained within one program is not just available for recall when developers are reading source from that program... Developer knowledge of the same character sequence used in different programs?

5.4 Reuse of the same character sequence

The following are two of the methods that might be used to prevent problems being caused by identifiers unexpectedly being, or becoming, visible:

- Recommend against the use of constructs that have the potential to make identifiers unexpectedly visible,
- Recommend that the character sequence used for an identifier when it is first declared be different from any other identifier visible at the point it is declared.

The following recommendation is the one needed if the approach of single guideline plus deviations is chosen.

Cg .4 A newly declared identifier shall not use the same character sequence as another identifier declared in the same program.

While guideline recommendation wording should normally aim to be precise, it should also try to be concise. It is not clear that the following guideline would be improved by containing more words.

Dev .4 A newly declared member may use the same character sequence as a member in a different structure/union type provided they both share the same semantic associations; and if they both have an arithmetic type, their types are the same.

5.5 Minimum closeness of character sequences

The recommendations made in this subsection apply to character sequences which may be treated as being the same as a different character sequence.

The idea behind the recommendations is to enforce a minimum distance on the similarity of all pairs of character sequences. The similarity being measured being based on either visual, aural, or semantic processes. The algorithm use to compute distance is

There are a number of ways of computing the distance between two points. The Levenstein algorithm is based on counting the number of changes that need to be made to one entity to make it identical to another entity...

5.5.1 Visual similarity

Cg .5 A newly declared identifier shall have a Levenstein distance, based on visual similarity of corresponding characters, of at least two when compared against all identifiers declared in the visible source of a program.

For the purpose of this guideline recommendation, the visual similarity Levenstein distance of two identifiers is defined as the sum, over all pairs of characters, of the visual distance between two characters (one from each identifier) occurring at the same position in the identifier character sequence (a space character is used to pad the shorter identifier). The visual distance between two characters is defined as (until a more accurate metric becomes available):

1. zero if they are the same character,
2. zero if one character represents the letter *O* (uppercase oh) and the other is the digit zero,
3. zero if one character represents the letter *l* (lowercase ell) and the other is the digit one,
4. otherwise, one.

Give characters at the front given greater weight???

5.5.2 Sound similarity

Performing

Memory lookup based on the sound sequence representation of a character sequence can sometimes result in information relating to a similar, previously remembered, sound sequence being returned. Also, if two identifiers are both referenced in a related section of source code, it is possible that both of their sound representations will be held in a reader's phonological loop (audio short-term memory) at the same time.

The following guideline recommendation is intended to reduce the likelihood of interference, in long and short term memory, between two character sequences that are mapped to similar sound sequences.

Rev .6 A newly declared identifier shall have a Levenstein distance, based on the phonemes in its spoken form, of at least two from existing identifiers declared in a program.

This guideline recommendation may be overly restrictive, preventing otherwise acceptable character sequences from being used. The following deviation is based on studies showing, at least for native-English speakers, that the start of a word has greater salience than its middle or end.

Dev .6 A newly declared identifier may have a Levenstein distance, based on phonemes, of one from existing identifiers declared in a program provided the difference occurs in the first phoneme.

5.5.3 Semantic cross-talk

The term *cross-talk* originated in radio communications where the information being sent on one channel leaked through to another channel. In some cases the leaked information is misinterpreted as belonging to the different channel.

The following guideline recommendation is motivated by possible semantic confusion, not by the possibility of typing mistakes. While the general block-edit string matching problem is NP-complete,[6] limiting the comparison to known *conceptual units* significantly reduces the computational cost of checking adherence. Human evaluation is needed to deduce which character subsequences form each conceptual unit.

Rev .7 A newly declared identifier shall have a Levenstein distance, based on individual conceptual units, of at least two from existing identifiers declared in a program.

Dev .7 A newly declared identifier defined in a function definition shall have a Levenstein distance, based on individual conceptual units, of at least two from existing identifiers defined in other function definitions.

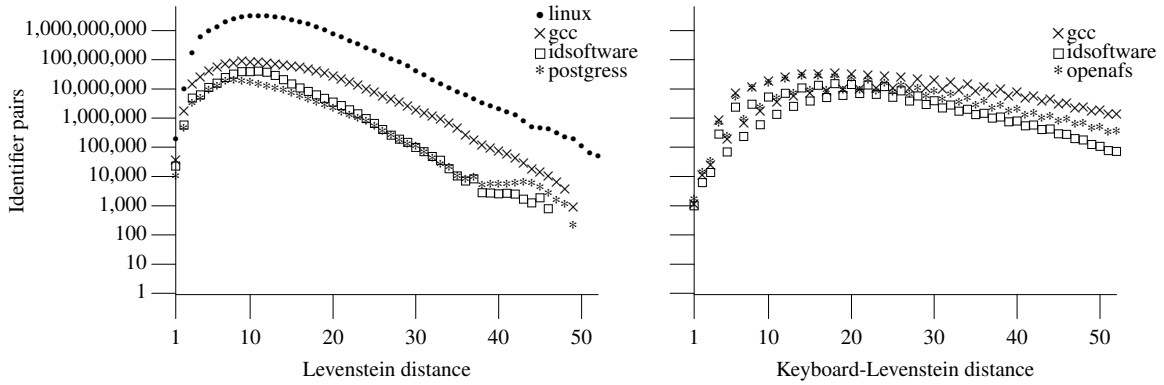

Figure .4: Number of identifiers having a given Levenstein distance from all other identifiers occurring in the visible form of the `.c` files of individual programs (i.e., identifiers in gcc were only compared against other identifier in gcc). The *keyboard-levenstein* distance was calculated using a weight of 1 when comparing characters on immediately adjacent keyboard keys and a weight of 2 for all other cases (the result was normalized to allow comparison against unweighted Levenstein distance values).

5.5.4 Typing mistakes

Cg .8 A new identifier shall have a Levenstein distance, based on individual characters, of at least two from existing identifiers declared in a program.

An identifier may differ by a Levenstein distance of one from another identifier and not be accessible at the point in the source a typing mistake occurs because it is not visible at that point. Requiring a Levenstein distance of two for all new identifiers may be overly restrictive (preventing otherwise acceptable character sequences from being used).

Dev .8 An identifier defined in a function definition may have a Levenstein distance, based on individual characters, of one from existing identifiers defined in other function definitions.

5.6 Enforcing recommendations

The computational resources needed to rapidly check a character sequence against a list of several million character sequences is well within the capabilities of computers used for software development today. However, the cost of maintaining an up to-date list of identifiers currently used within a software product under active development can be a non-trivial task and may result in developers having to wait for a relatively long period of time for a proposed choice of identifier spelling to be checked against existing identifier spellings.

One way of reducing the identifier database maintenance resources required is to reduce the number of identifiers that need to be checked.

5.6.1 Same character sequence, different identifier

All identifiers have attributes other than their character sequence (e.g., they usually have a scope and some have a type), and it might be possible to take advantage of the consequences of an identifier having these attributes; for instance:

- If the identifier `X_1` is not visible at the point in the source where a developer declares and uses the identifier `X_2`, a reference to `X_2` mistyped as `X_1` will result in a translator diagnostic being issued.

- If the identifiers X_1 and X_2 are in different name spaces, a mistyped reference to one can never result in a reference to the other.
- If the identifiers X_1 and X_2 are objects or functions having incompatible types, a mistyped reference to one, which refers to the other, is likely to result in a translator diagnostic being issued.

Dev .4 A newly declared identifier may use the same character sequence as another identifier declared in the same program provided they are both used to denote the same information and both have block scope.

On the basis that the type of objects having scalar type is rarely changed to be a non-scalar type (evidence???).

Dev .4 Identifiers denoting objects, functions, or types need only be compared against other identifiers having, or returning, a scalar type.

5.6.2 Different character sequences, same identifier?

The enforcement problem with looks like, sounds like, and semantic associations is the lack of reliable automated methods of measuring the similarity of the quantities of interest.
Manually checking a new identifier against all existing identifiers is unlikely to be practical. Some form of limited checking could be performed during a code review.

References

Citations added in version 1.0b start at 1449.

1. N. Anquetil and T. Lethbridge. Extracting concepts from file names: A new file clustering criterion. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 84–93. IEEE Computer Society Press/ACM Press, 1998.

2. N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11:201–221, 1999.

3. M. B. Holbrook. A comparison of methods for measuring the interletter similarity between capital letters. *Perception & Psychophysics*, 17(6):532–536, 1975.

4. M. B. Holbrook. Effect of subjective interletter similarity, perceived word similarity, and contextual variables on the recognition of letter substitutions in a proofreading task. *Perceptual and Motor Skills*, 47:251–258, 1978.

5. D. Lin. Automatic retrieval and clustering of similar words. In *Proceedings of Coling/ACL-98*, pages 768–774, 1998.

6. D. P. Lopresti and A. Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 181(1):159–179, 1997.

7. I. S. MacKenzie and R. W. Soukoreff. Text entry for mobile computing: Models and methods, theory and practice. *Human-Computer Interaction*, 17:147–198, 2002.

8. D. E. Rumelhart and D. A. Norman. Similating a skilled typist: A study of skilled cognitive-motor performance. *Cognitive Science*, 6:1–36, 1982.

9. L. H. Shaffer and J. Hardwick. Typing performance as a function of text. *Quarterly Journal of Experimental Psychology*, 20:360–369, 1968.

10. M. A. Tinker. The relative legibility of the letters, the digits, and of certain mathematical signs. *Journal of Generative Psychology*, 1:472–494, 1928.