The Hungarian Naming Convention

written by Doug Klunder
edited and updated by Cathy Saxton
May 30, 2001

1. Introduction

This document describes a set of naming conventions that go by the name "Hungarian," referring both to the nationality of their original developer, Charles Simonyi, and also to the fact that to an uninitiated programmer they are somewhat confusing. Once you have gained familiarity with Hungarian, however, we believe that you will find that the clarity of code is enhanced. For convenience, this document first describes how to use Hungarian, and then describes why it is useful; the general approach is from a programming viewpoint, rather than a mathematical one. For a more theoretical approach, you are invited to read Chapter 2 of Simonyi's "Meta-Programming" thesis.

2. The Rules

Hungarian is largely language independent; it is equally applicable to a microprocessor assembly language and to a fourth-generation database application language (and has been used in both). There are rules for naming variables, functions, defined constants, and structures. Those rules are explained in the following sections.

This list of rules is comprehensive, and thus covers both common and rare situations. A new user of hungarian can just concentrate on the rules that apply to the specific constructs that he or she is using, referring back to this document as the need arises to determine names for more complicated cases.

2.1. Variables

The most common type of identifier is a variable name. All variable names are composed of four elements: scope, prefixes, base type, and qualifiers. Not all elements are present in all variable names; the only part that is always present is the base type. These elements are described in detail in the following sections.

The base type should not be confused with the types supported directly by the programming language; most base types are application specific (and are based on structures, classes, enumerated values, etc, that are created for that application). For example, the base type *stk* could refer to a class implementation of a stack; a *co* could be a value specifying a color.

As you create enumerated values, structures, and classes, you will create base types for those concepts. That is discussed further in sections 2.2 "Defined Constants" and 2.3 "Structures."

2.1.1. Base Types

Tags should be short (typically two or three letters) and somewhat mnemonic. Because of the brevity, the mnemonic value will be useful only as a reminder to someone who knows the application, and has been told what the basic types are; the name will not be sufficient to inform (by itself) a casual viewer what is being referred to. For example, a *co* could just as easily refer to a geometric coordinate, or to a commanding officer. Within the context of a given application, however, a *co* would always have a specific meaning; all *co*'s would refer to the same type of object, and all references to such an object would use the term *co*.

One should resist the natural first impulse to use a short descriptive generic English term as a type name. This is almost always a mistake. One should not preempt the most useful English phrases for the provincial purposes of any given version of a given program. Chances are that the same generic term could be equally applicable to many more types in the same program. How will we know which is the one with the pretty "logical" name, and which have the more arbitrary variants typically obtained by omitting various vowels or by other disfigurement? Also, in communicating with other programmers, how do we distinguish the generic use of the common term from the reserved technical usage? In practice, it seems best to use some abbreviated form of the generic term, or perhaps an acronym. In speech, the tag may be spelled out, or a pronounceable nickname may be used. In time, the exact derivation of the tag may be forgotten, but its meaning will still be clear.

As is probably obvious from the above, it is essential that all tags used in a given application be clearly documented. This is extremely useful in helping a new programmer learn the code; it not only enables him or her to decode the otherwise cryptic names, but it also serves to describe the underlying concepts of the program, since the data types tend to determine how the program works. It is also worth pointing out that this is not nearly as onerous as it sounds; while there may be tens of thousands of variables in a program, the number of types is likely to be quite small.

Although most types are particular to a given application, there are a few standard ones that appear in many different applications; synonyms for these types should never be used:

f	a flag (boolean, logical). The qualifier (discussed in section 2.1.3 “Qualifiers”) should describe the condition that will cause this flag to be set (e.g. fError would be clear if there were no error, set if one exists). This tag may refer to a single bit, a byte, or a word; often it will be an object of type BOOL (defined by the application, usually as int). Usually the object referred to will contain either 1 (fTrue, TRUE) or 0 (fFalse, FALSE). In some instances, other values may be used, either for efficiency or historical reasons; such a use usually indicates that another type may be more appropriate.
ch	a character.
sz	a zero-terminated string. This is the way strings are represented in C/C++.
st	a Pascal-type string (first byte is count, remainder is the actual characters). Not commonly used in C/C++ programming, but can be more efficient since the length of the string does not need to be calculated repeatedly. Also useful when performing file I/O.
fn	a function. Since about the only thing you can do with a function is take its address, this almost always has a “p” prefix (see section 2.1.2 “Prefixes” below).

There are some more types that appear in many applications; they should only be used for the most generic purposes:

b	a byte (8 bits). For most purposes, this is an incorrect usage, since the base type should be a description of the purpose of the variable, not its size. Correct usages are generally limited to generic subroutines (e.g. sort an array of bytes) that can deal with a number of different (but similarly-sized) base types; another common use is in conjunction with the prefix “c” (see section 2.1.2 “Prefixes” below), to produce a count of bytes (the size) for some object. The exact meaning of b is also somewhat loose; it sometimes means a signed quantity and sometimes unsigned.
w	an int (as defined by the C/C++ compiler -- it does not imply a specific size). The same warnings apply to this as to b.
sw	a short word (16 bits). The same warnings apply to this as to b.
lw	a long word (32 bits). The same warnings apply to this as to b.
llw	a long long word (64-bits). The same warnings apply to this as to b.

2.1.2. Prefixes

Base types are not by themselves sufficient to fully describe the use of a variable, since variables often refer to more complex items. The more complex items are always derived from some combination of simple items, with a few operations. For example, there may be a pointer to a *co*, or an array of them, or a count of *co*’s. These operations are represented in Hungarian by prefixes; the combination of the prefixes and the base type represent the complete type of an entity. Note that a type may consist of multiple prefixes in addition to the base type (e.g. a pointer to a count of *co*’s); the prefixes are read right to left, with each prefix applying to the remainder of the type (see examples below).

In theory, new prefixes can be created, just as new types are routinely created for each application. In practice, very few new prefixes have been created over the years, as the set that already exists is rather comprehensive for operations likely to be applied to types.

The standard prefixes are:

p	a pointer. Note that a pointer is not itself a type, it is an operation applied to a type. For example, a pch is a pointer to a character.
a	an array. (see also mp and dn below). For example, an ach is an array of characters; a pch could point to one of the characters in the array. Note that it is perfectly reasonable to assign an ach to a pch; pch points to the first character in the array.
i	an index into an array. For example, an ich is used to index an ach (with a ch as the result, i.e. ch = ach[ich]).
c	a count. For example, the first byte of an st is a count of characters, or a cch.

d	a difference between two instances of a type. This is often confused with a count, but is in reality quite separate. For example, a cch could refer to the number of characters in a string, whereas a dch could refer to the difference between the values ‘a’ and ‘A’. The confusion arises when dealing with indices; a dich (difference between indices into a character array) is equivalent to a cch (count of characters); which one to use depends on the viewpoint.
h	a handle. This is often a pointer to a pointer (used to allow moveable heap objects). Most commonly used for interface to an operating system. In some systems (e.g. Windows) a handle is not a pointer to a pointer. To avoid confusion it may be best to use pp as the prefix when the application is actually going to do the indirection, and reserve h for instances in which the handle is just passed on to the system.
gr	a group, usually of variable-size objects. Typically used with the “f” base type, meaning a group of flags. It is often implemented as bitfields in a word or long.
mp	an array. This prefix is followed by two types, rather than the standard one, and represents the most general case of an array. From a mathematical viewpoint, an array is simply a function mapping the index to the value stored in the array (hence mp as an abbreviation of map). In the construct mpxy, x is the type of the index and y is the type of the value stored in the array. In many cases, mp provides more flexibility than is required; it is often the case that only the type of the value is important and the index is just an integer with no other meaning. In that case, an a is used; this means that an ax is equivalent to an mpixx. Both will be indexed using an ix and will produce an x.
dn	an array. This is used in the rare case that the important part of the array mapping is the index, not the value. dn is an abbreviation for domain. An example of a plausible use is given in the discussion of e, below.
e	an element of an array. This is used in conjunction with a dn (and is thus just as rare); it is the type of the value stored in a dn. Just as ax is equivalent to mpixx, dnx is equivalent to mpxex. An example of use is in the native code generation part of the CS compiler; there is a type vr (an acronym for virtual register). A vr is just a simple integer, specifying which register to use for various pieces of code output. However, there is quite a bit more information than just a number that is associated with each register. This additional data is stored in a structure called an evr; there is an array of them called dnvr. Thus, the information for a given register (evr) can be found with the expression dnvr[vr].
f	a bit within a type. This is a new prefix that is currently used only by a few projects. It is typically used for overloading an integer type with one or more bit flags, in otherwise unused portions of the integer. This should not be confused with the f type, in which the entire value is used to contain the flag. An example is a scan mode (base type sm), with possible values smForward and smBackwards. Since the basic mode only requires a few bits (in this case only one bit), the remainder of a word can be used to encode other information. One bit is used for fsmWrap, another for fsmCaseInsens. Here the f is a prefix to the sm type, specifying only a single bit is used.
u	a union. This is a rarely used prefix; it is used for variables that can hold one of several types. In practice this becomes unwieldy. An example is a urwcol, which can hold either a rw type or a col type.
w	a wide value. This is used in conjunction with ch and sz for double-byte characters (used for Unicode, for example).
u	an unsigned value. Typically used as a prefix to w, lw, and sw.

2.1.2.1. Some examples

Since the prefixes and base types both appear in lower case, with no separating punctuation, ambiguity can arise. Is *pfc* a tag of its own (e.g. for a private first class), or is it a pointer to an *fc*? Such questions can be answered only if one is familiar with the specific types used in a program. To avoid problems like this it is often wise to avoid creating base type names that begin with any of the common prefixes. In practice, ambiguity does not seem to be a problem. The idea of additional punctuation to remove the ambiguity has been shown to be impractical.

The following list contains both common and rarer usages:

pch	a pointer to a character.
ich	an index into an array of characters.
ach	an array of characters.
cch	a count of characters. Possibly the length of a string.
px	a pointer to an object of type x.
pich	a pointer to an index into a character array. A common use for something like this is passing a pointer as a parameter to a function so that a return value can be stored through the pointer; pich would be extremely unlikely to be used in an expression without indirection (pich += 2 is probably gibberish; (*pich) += 2 may well be meaningful).
en	probably a base type (such as an entry). Conceivably it is an element of an array indexed by an n; only knowledge of the application can tell for certain.
dx	length of a horizontal line (difference between x coordinates).
mpmipfn	an array of pointers to functions, indexed by mi's. For example, an mi could be a menu item, and this array could be used for a command dispatch. Again, context makes the parsing clear, this could equally well be interpreted as an array of fn's (perhaps friendly nukes), indexed by mip's (perhaps missile placements).

2.1.3. Qualifiers

While the prefixes and base type are sufficient to fully specify the type of a variable, this may not be sufficient to distinguish the variable. If there are two variables of the same type within the same context, further specification is required to disambiguate. This is done with qualifiers. A qualifier is a short descriptive word (or facsimile; good English is not required) that indicates what the variable is used for. In some cases, multiple words may be used. Some distinctive punctuation should be used to separate the qualifier from the type; in C/C++ and other languages that support it, this is done by making the first letter of the qualifier upper-case. (If multiple words are used, the first letter of each should be upper-case; the remainder of the name, both type and qualifier, is always lower-case.)

Exactly what constitutes a naming context is language specific; within C/C++ the contexts are individual blocks (compound statements), functions, data structures (for naming fields), or the entire program (globals). As a matter of good programming style, it is not recommended that hiding of names be used; this means that any context should be considered to include all of its subcontexts. (In other words, don't give a local the same name as a global.) If there is no conflict within a given context (only one variable of a given type), it is not necessary to use a qualifier; the type alone serves to identify the variable. In small contexts (data structures or small functions), a qualifier should not be used except in case of conflict; in larger contexts it is often a good idea to use a qualifier even when not necessary, since later modification of the code may make it necessary. In cases of ambiguity, one of the variables may be left with no qualifier; this should only be done if it is clearly more important than the other variables of the same type (no qualifier implies primary usage).

When using qualifiers to clarify a variable's use, it is best to describe the state or use that the variable represents. Avoid using names that describe what happened to set a state or what should happen as a result. For example, a flag set when part of a window has become newly exposed should be *fDirty*, not *fNeedsUpdate* (which doesn't also communicate that the window contents should not be used), or *fUncovered* (since other things might result in the same "dirty" state).

Since many uses of variables fall into the same basic categories, there are several standard qualifiers. If applicable, one of these should be used since they specify meaning with no chance of confusion. In the case of multiple word qualifiers, the order of the words is not crucial, and should be chosen for clarity; if one of the words is a standard qualifier, it should probably come last. The standard qualifiers are:

Min	the first element in a set. This is very similar to First, but typically refers to the actual first element of an array, not just the first to be dealt with. It is also more often used with a pointer than an index, since ixMin tends to be 0.
Max	the upper limit of elements in a set. This is not a valid value; for all valid values of x, x<xMax. Max is typically used for compile-time constants, or variables that are set at load time and not changed. Very often, ixMax is used to specify the number of elements in an ax array.

Mic	the *current* first element in a set. This is very similar to Min, but is used where the low value varies over time (such as for a heap that grows downward in memory). Like Min, this is a valid value. Since few things grow downward, this is not often used.
Most	the *current* last element in a set. Often paired with a Min. Can also be viewed as Mac-1. Typical usage would be: for (pch = pchMin; pch <= pchMost; pch++)
Mac	the *current* upper limit of elements in a set. This is very similar to Max, but is used where the upper limit varies over time (such as for a variable length structure, or a growing heap). This is not a valid value; it is often paired with Min, as in this common loop: for (pch = pchMin; pch < pchMac; pch++)
First	the first element (in a set) *to be dealt with*. This is usually used with an index or a pointer (e.g. pchFirst). The index may be an implied index (as with a rw type in a spreadsheet).
Last	the last element (in a set) *to be dealt with*. This is usually used with an index or a pointer (e.g. pchLast). Both First and Last represent valid values (compare with Lim below); they are often paired, as in this common loop: for (ich = ichFirst; ich <= ichLast; ich++)
Lim	the upper limit of elements (in a set) *to be dealt with*. This is not a valid value; for all valid values of x, x<xLim. xLim is equivalent to xLast+1; xLim-xFirst is the dx which specifies the number of elements in the set. Thus, the following code is typical (cp is a type that is an implied index): for (cp = cpFirst, cpLim = cpFirst + dcp; cp < cpLim; cp++)

Note that the above qualifiers have a strict relationship:

    Min <= Mic <= First <= Last <= Most < Lim <= Mac <= Max

Commonly, only a few of these modifiers are used together in a given context. In many cases picking just one correct modifier from this group will clearly explain the valid bounds of an operation and prevent off-by-one errors.

Sav	a temporary saved value. Often used as part of error recovery, or just when temporarily modifying variables that need to be restored. An example of typical usage:
	rwSav = rwAct;
	for (rwAct = rwFirst; rwAct <= rwLast; rwAct++)
	    ...
	rwAct = rwSav;

Nil	a special illegal value. Typically used with defined constants, this is a value that can be distinguished from all legal values. This is often 0 or -1, but may be something else in some circumstances.
Null	the 0 value. Typically used with defined constants, this value is always 0, typically an illegal value. May or may not be equivalent to Nil. In order to avoid confusion, it is usually best not to have both Nil and Null defined; if both do exist, the differences should be clearly delineated (which turns out to be nearly impossible in practice, so resist using both of these together).
T	a temporary value. This is often convenient to distinguish the second value in a given context. However, unless it is a truly temporary usage, it is often better to use a more descriptive qualifier. (After all, what happens when you add the third variable of the same type?). Some particularly poor usages stack the T's up, to produce variables such as pchTT or pchT2; this is almost certainly an indicator that better qualifiers should be used.
Cur	the current item. Used to distinguish from generic loop variables, Next and Prev items, etc.
Src	a source. Typically paired with Dst and used in transfer operations.
Dst	a destination. Typically paired with Src and used in transfer operations. Sometimes written as Dest.
Next	the next item. Typically a pointer variable used when traversing a linked list.
Prev	the previous item. Typically a pointer variable used when traversing a linked list.

2.1.4. Scope

Most variables are used within the scope of a function. For those that have a broader scope, one of the following scopes is placed at the beginning of the variable name (before any prefixes):

m_	a member of a class. This is used to label the variables (but not methods) in a class. This is used so that member variables can be easily distinguished from local variables in the class methods.
g_	a global. This is applied to variables that are available to numerous functions in multiple files.
s_	a static. This is applied to variables that are available to multiple functions all within the same file, but not available to functions in other files (essentially a global variable, but private to one file). These variables are almost always declared with the "static" keyword, e.g.: static int s_dxWindow;

Is it of course worth mentioning that the use of global (g_) and static (s_) variables is generally discouraged, avoided in favor of other mechanisms that are less error-prone (and thread-safe) such as using member variables and passing variables to the functions that need them.

2.1.5. Structure Members

When possible, structure members are named the same way variables are. Since the context is small (only the structure), conflicts are less likely, and qualifiers are often neither needed nor used.

2.1.6. Arrays

Arrays in C/C++ present an interesting dilemma. An array variable is technically a pointer, so if you have an array of characters, should it be called *ach* or *pach*? In practice, it turns out to be useful to use each of these, depending on the circumstances. One interesting consideration is what happens when you calculate sizeof() for the variable. For a statically declared array (e.g. char achVowels[5];), sizeof() will return the length of the array. For an array that has been allocated or received as a function parameter, sizeof() will return the size of the pointer variable. Thus, it becomes useful to distinguish these cases. Use just the *a* prefix for statically declared arrays, and use the *pa* prefix for dynamically allocated arrays and arrays received as function parameters.

The same logic works for strings, which are really just arrays of characters. A statically declared string would be an *sz* and a string that is allocated or received as a function parameter would be a *psz*.

2.2. Defined Constants

As much as possible, defined constants should look just like variables of the same type. For many types, defined constants will exist for the Nil, Max, Min, and/or Last values. The program text will read exactly as if they are variables. There are three common exceptions, all originating in the mists of time, and unlikely to change soon. NULL is defined to be 0, and is used with all pointer types; TRUE and FALSE are defined to be 1 and 0, and are used with f types (correct Hungarian, practiced by some projects, uses fTrue and fFalse instead of TRUE and FALSE).

There are often types for which each value is a defined constant; these are essentially equivalent to enumeration types supported by some languages (including C/C++). These are typically types used for table-driven algorithms, for specifying options to a function, or specifying possible return values. Note that these types are in fact separate types; they are not all examples of the same type, nor are they values for the w type. Since they are special purpose (you can't pass an option for function x to function y with any meaning), they must be a new type.

Since defined constants are typed quantities, the type must be present in the names; possible values for colors are not RED, BLUE, YELLOW, GREEN, BLACK, etc., but rather could be coRed, coBlue, coYellow, coGreen, coBlack, etc.

In cases where defined constants are used to specify options to a function, you can think of the value as a "mode." It is often convenient to create the type name by using the initials of the function followed by an "m" (watching out for conflicts). For example, if a function DrawBitmap offers options for drawing the bitmap in different locations, you could define values *dbmLeft*, *dbmCenter*, and *dbmRight* that could be passed for a *dbm* ("draw bitmap mode") parameter.

When creating a list of values to indicate the use for a structure, you can think of the values as representing “kinds” of that structure. A type name can easily be created by using the structure name followed by a “k.” For example, if you had a structure *GP* for a game player, you might want a member that indicated whether the player was local, played by someone connected through a network, or controlled by the computer. You could create *gpk* (“game player kind”) values for those possibilities (e.g. *gpkLocal*, *gpkNetwork*, *gpkComputer*).

2.3. Structures

Each structure is almost by definition its own type, and should be named as a type (two or three letters with some possible mnemonic value). By convention in C, the entire type name is capitalized in the definition of the type. When a variable of that structure type is declared, it should take the same type name, with all letters lowercase (e.g. DATE date;). The same rules apply to classes and unions.

Many projects find it convenient to include typedefs for each structure type; this means that the word struct is not included in declarations, and allows the redefinition of the type to an array, or even a simple type, without having to change all the declarations. Typedefs may also be suitable for non-structure types, particularly any that are not simple int’s.

In cases where there is a “base” structure (containing common information) which is included by multiple special-instance structures (containing the base structure followed by instance-specific fields), the instance names should consist of the base structure name plus a character. For example, in Word there is a base type of CHR (character run); special instances are CHRF (formula character run), CHRT (tab character run), and CHRV (vanished character run).

2.4. Functions

The above rules used for variable names do not work as well for functions. Whereas the type of a variable is always quite important, specifying how that variable may be used, the important part of a function is typically what it does; this is especially true for functions that don’t return a value. In addition, the context for functions is usually the entire program, so there is more chance for conflict. To handle these issues, a few modifications are made to the rules:

1) All function names are distinguished from variable names by the use of some standard punctuation; in C/C++ this is done by capitalizing the first letter of the function name (all variable names begin with a lowercase type).
2) If the function returns a value (explicitly, not implicitly through pointer parameters), the function name begins with the type of the value returned; if no value is returned, the name must not begin with a valid type.
3) If the function just determines and returns a value based mainly on its parameters, with few or no side effects, it is standard to name the function AFromBCD..., where A is the type of the value returned, and B, C, D, etc. are the types of the parameters. As a simple example, returning the decimal value of an ASCII string would be done by a function called WFromSz (equivalent to the standard C atoi).
4) If the function does more than just determine a return value based on its parameters, follow the return type (if any) with a few words describing what the function does (a verb followed by an object is usually good). Each word should be capitalized. If the type of a parameter is important, it may be appended as well; in many cases this is unnecessary, and may even be confusing (if the type is not truly important).
5) If the function operates on (modifies) an object, the object’s type should be included in the name of the function using the form VerbType, where “Verb” describes the operation performed. Functions like this are commonly used when programming in a class-like (or object-oriented) manner; typically the first parameter to such a function is a pointer to the object to be manipulated. For example, you could have functions like InitCa(pca, ...), DrawDd(pdd, ...), etc.
6) If the function is defined as static (it can be called only from functions within its file), an underscore (_) should be placed at the beginning of the name.

2.4.1. Macros

Macros should be handled exactly the same way as functions.

2.4.2. Labels

Labels can be considered to be a variant on functions; they are identifiers specifying an entry point to a chunk of code. Within C, they are named similarly to functions; they obviously neither return a value nor take parameters, so no types are specified. The name itself consists of a capital L followed by just a few words specifying what the code it labels does or what condition must be true before reaching the label. Since the context of a label is limited to its function, these can be pretty generic terms; typical examples are LCleanup, LShowError, LBitmapLoaded.

3. Advantages of Hungarian

Before adopting a set of conventions, there are always two questions that must be answered:

Why have conventions at all?
Why use this particular set of conventions?

The two questions are actually very closely related; answering the first will usually give an answer to the second, since knowing the goals allows one to see how closely a solution will meet the goals. For naming conventions, many of the goals are well-known, if not often formalized; most good programmers already attempt to meet the goals in a variety of ways.

3.1. Mnemonic Value

An important need in naming objects in a program is the ability to remember what the name is, so that when the object is used, the programmer can quickly determine the name. Traditionally, this need has been met by using descriptive names for variables; for a given programmer working continually on a given program this is usually adequate. Problems arise, however, when a different programmer works on the project, or when the same programmer returns after a hiatus. What was once descriptive now has to be relearned. Hungarian helps somewhat in this respect, though it is not complete. The first part of a variable name can always be determined with no effort (it is the type), and if it is a standard use, the qualifier can also be determined (since it is one of the standard qualifiers). Non-standard qualifiers and function names can not be immediately determined; however, the situation is certainly no worse than the traditional situation, since the qualifier or function name has as much descriptive value as a traditional name. Furthermore, since there are fewer names that must be remembered (since the standard ones are presumably already committed to memory), it is easier to remember them.

3.2. Suggestive Value

At least as important as being able to go from an object to a name (the mnemonic value) is the ability to go from a name to an object (the suggestive value). This is most important when reading code written by someone else; this affects almost all programs today, either because multiple people are working on them, or because they are outgrowths of earlier programs. Again, the traditional approach has been to use names descriptive in some manner; Hungarian again improves the situation somewhat. For the relatively small cost of learning the types used in a given program, a reader gains a much better understanding of what the program does, since the types used in a statement often help determine the meaning of the statement. This is enhanced even more by the use of standard qualifiers; again, the non-standard qualifiers are at least as clear as the traditional names.

3.3. Consistency

Partially an aesthetic idea (“the code looks better”), this is also an important concept for the readability of code. Just as Americans often have an extremely difficult job following the action in Russian novels since the same character goes by many different names, a programmer will have a difficult time understanding code in which the same object is referred to in unrelated ways. Similarly, it is confusing to find the same name referring to unrelated objects. This is a serious problem in traditional contexts, since English is a rich enough language to have many terms that roughly describe the same concept, and also terms that can describe multiple concepts. This problem is exacerbated when programmers resolve name conflicts by use of abbreviations, variant spellings, or homonyms; all of these methods are prone to accidental misuse, through typographical errors or simple failure to understand subtle differences. Hungarian resolves this problem by the use of detailed rules; since all names are created using the same rules, they are consistent in usage.

3.4. Speed

It is desirable to minimize the amount of time spent on determining names; in a sense this is wasted time, since getting the “right” name doesn’t improve the program’s efficiency or functionality. Since the traditional naming methods rely on good descriptions to meet the above goals, a programmer has to spend a goodly amount of time to in fact invent good descriptions; speedy name decisions are likely to result in unmnemonic, unsuggestive, or inconsistent names. In Hungarian, on the other hand, only a few name selections are likely to be time-consuming; good type names often require some thought, as do some functions. The majority of names can be quickly derived from type information and standard qualifiers.

3.5. Type Checking

Just as it is useful to have compilers perform type checking on objects, it is useful to make that type checking apparent to a reader of the code. Among other things, compilers only type check to the extent that they know about types; in Hungarian there are typically a number of types that are represented in the same format as far as the compiler is concerned. Traditional naming schemes provide little or no help in type checking; since Hungarian is based on types, a strong degree of type checking is inherent in the reading of each statement. For example, the statement

```
    if (co == coRed)
        *mpcopx[coBlue] += dx;
```

can easily be determined to be plausible from a type standpoint; a *co* is compared with another *co*; an array lookup uses a *co* index to produce a *px*; this is dereferenced to produce an *x*, which has a *dx* added to it. If the * were missing in the second line, the statement would be obviously incorrect, since it is not proper to add a *dx* to a *px*. In addition, the variable names can actually help produce the correct code; to go from a *co* to an *x*, it is clear that you can index into *mpcopx*, then dereference; the necessity of each step is also clear.

In addition to standard type checking, the use of standard qualifiers also helps to solve some common programming problems; the most obvious case is the standard off-by-one problem. One should become very suspicious when an expression such as *x<xLast* or *x<=xMac* is used; due to the very specific meaning of these qualifiers, the expressions should almost always be *x<=xLast* or *x<xMac*. Traditional uses of “last” and “max” as descriptive terms are much looser; only further examination of code can determine whether they refer to the last valid value or the first invalid value.

This use of Hungarian for type checking also helps determine what constitutes a “type.” As suggested above, the concept of “type” in this context is determined by the set of operations that can be applied to an object. The test for type equivalence is simple: could the same set of operations be meaningfully applied to each of the objects in question? The concept of “operation” is considered quite generally here; “being the subscript of array A” or “being the second parameter of function Position” are operations on object *x*. The point is that “integers” *x* and *y* are not of the same type if Position(x, y) is legal but Position(y, x) is nonsensical.

Note also that type checking extends beyond expressions; it is also quite helpful in making sure that variable (and function) declarations are correct. In essence a C declaration can be determined by reading the Hungarian type in reverse order (though it is somewhat more complicated when arrays and functions are involved). For example a variable *apnode* would be declared as NODE *apnode[10]; the NODE corresponds to the base type *node*, the * corresponds to the prefix *p*, and the [] corresponds to the prefix *a*.

3.6. Writeability

This is a concept not often considered, and arguable in some of its specifics; nonetheless, it is a concept worth considering. The basic premise is that it is always easier to read and modify one’s own code than that written by someone else. Therefore, if everyone writes in the same manner, it will be equally easy to read and modify code written by anyone. Since one of the key parts of writing code is the names used, if everyone uses the same names, everyone’s code will be similar, and therefore easy to read and modify. Traditional naming schemes are extremely unlikely to reach this goal, since English has far too many ambiguities to expect different individuals to describe things in identical terms. It would be naive to expect that Hungarian will cause all programmers to write code identically, or even to use identical names. The names are likely to be much more similar, however, since they are composed using the same rules, with the same types and standard qualifiers.

4. Conclusion

Hungarian is a useful set of rules used to determine the names used in a program. There is no denying that it takes a little time to become familiar with it; true enlightenment comes only with effort. We strongly believe the results are worth the effort. The Applications Development group at Microsoft has been using Hungarian since its inception in 1981, and people at Xerox PARC were using it even earlier. The consistent use of Hungarian makes the programmer’s job easier; it is both easier to write in Hungarian (there are fewer superfluous choices to make) and easier to read and modify existing code. The set of conventions is sufficient to deal with most current situations by itself; it has also proven adaptable to changes in the programming environment. Perhaps the best testimonial for Hungarian is the fact that a number of programmers have continued to use Hungarian even after leaving the jobs in which they encountered it; they have felt that the advantages were great enough to warrant the effort necessary to promote its use elsewhere. We hope that you will feel this way as well, once you become familiar with Hungarian’s usage.