

Hungarian Notation

Charles Simonyi
Microsoft Corporation

Reprinted November 1999

Summary: Charles Simonyi's explication of the Hungarian notation identifier naming convention. (10 printed pages)

A note from Dr. GUI: Long, long ago in the early days of DOS, Microsoft's Chief Architect Dr. Charles Simonyi introduced an identifier naming convention that adds a prefix to the identifier name to indicate the functional type of the identifier.

This system became widely used inside Microsoft. It came to be known as "Hungarian notation" because the prefixes make the variable names look a bit as though they're written in some non-English language and because Simonyi is originally from Hungary.

As it turns out, the Hungarian naming convention is quite useful—it's one technique among many that helps programmers produce better code faster. Since most of the headers and documentation Microsoft has published over the last 15 years have used Hungarian notation names for identifiers, many programmers outside of Microsoft have adopted one variation or another of this scheme for naming their identifiers.

Perhaps the most important publication that encouraged the use of Hungarian notation was the first book read by almost every Windows programmer: Charles Petzold's *Programming Windows*. It used a dialect of Hungarian notation throughout and briefly described the notation in its first chapter.

MSDN is pleased to publish here the original edition of Simonyi's historic work.

Program Identifier Naming Conventions

This monograph is intended to give you the flavor of the major ideas behind the conventions.

When confronted with the need for a new name in a program, a good programmer will generally consider the following factors to reach a decision:

1. Mnemonic value—so that the programmer can remember the name.
2. Suggestive value—so that others can read the code.
3. "Consistency"—this is often viewed as an aesthetic idea, yet it also has to do with the information efficiency of the program text. Roughly speaking, we want similar names for similar quantities.
4. Speed of the decision—we cannot spend too much time pondering the name of a single quantity, nor is there time for typing and editing extremely long variable names.

All in all, name selection can be a frustrating and time-consuming subtask. Often, a name that satisfies some of the above criteria will contradict the others. Maintaining consistency can be especially difficult.

Advantages of the Conventions

The following naming conventions provide a very convenient framework for generating names that satisfy the above criteria. The basic idea is to name all quantities by their types. This simple statement requires considerable elaboration. (What is meant by "types"? What happens if "types" are not unique?) However, once we can agree on the framework, the benefits readily follow. The following are examples:

1. The names will be mnemonic in a very specific sense: If someone remembers the type of a quantity or how it is constructed from other types, the name will be readily apparent.
2. The names will be suggestive as well: We will be able to map any name into the type of the quantity, hence obtaining information about the shape and the use of the quantity.
3. The names will be consistent because they will have been produced by the same rules.

Page Options

Average rating:
6 out of 9

 [Rate this page](#)

 [Print this page](#)

 [E-mail this page](#)

 [Add to Favorites](#)

4. The decision on the name will be mechanical, thus speedy.
5. Expressions in the program can be subjected to consistency checks that are very similar to the "dimension" checks in physics.

Type Calculus

As suggested above, the concept of "type" in this context is determined by the set of operations that can be applied to a quantity. The test for type equivalence is simple: could the same set of operations be meaningfully applied to the quantities in questions? If so, the types are thought to be the same. If there are operations that apply to a quantity in exclusion of others, the type of the quantity is different.

The concept of "operation" is considered quite generally here; "being the subscript of array *A*" or "being the second parameter of procedure *Position*" are operations on quantity *x* (and *A* or *Position* as well). 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. Here we can also sense how the concepts of type and name merge: *x* is so named because it is an *x*-coordinate, and it seems that its type is also an *x*-coordinate. Most programmers probably would have named such a quantity *x*. In this instance, the conventions merely codify and clarify what has been widespread programming practice.

Note that the above definition of type (which, incidentally, is suggested by languages such as SIMULA and Smalltalk) is a superset of the more common definition, which takes only the quantity's representation into account. Naturally, if the representations of *x* and *y* are different, there will exist some operations that could be applied to *x* but not *y*, or the reverse.

Let us not forget that we are talking about conventions that are to be used by humans for the benefit of humans. Capabilities or restrictions of the programming environment are not at issue here. The exact determination of what constitutes a "type" is not critical, either. If a quantity is incorrectly classified, we have a style problem, not a bug.

Naming Rules

My thesis discusses in detail the following specific naming rules:

1. Quantities are named by their type possibly followed by a qualifier. A convenient (and legal) punctuation is recommended to separate the type and qualifier part of a name. (In C, we use a capital initial for the qualifier as in *rowFirst*: *row* is the type; *First* is the qualifier.)
2. Qualifiers distinguish quantities that are of the same type and that exist within the same naming context. Note that contexts may include the whole system, a block, a procedure, or a data structure (for fields), depending on the programming environment. If one of the "standard qualifiers" is applicable, it should be used. Otherwise, the programmer can choose the qualifier. The choice should be simple to make, because the qualifier needs to be unique only within the type and within the scope—a set that is expected to be small in most cases. In rare instances more than one qualifier may appear in a name. Standard qualifiers and their associated semantics are listed below. An example is worthwhile: *rowLast* is a type *row* value; that is, the last element in an interval. The definition of *Last* states that the interval is "closed"; that is, a loop through the interval should include *rowLast* as its last value.
3. Simple types are named by short tags that are chosen by the programmer. The recommendation that the tags be small is startling to many programmers. The essential reason for short tags is to make the implementation of rule 4 realistic. Other reasons are listed below.
4. Names of constructed types should be constructed from the names of the constituent types. A number of standard schemes for constructing pointer, array, and different types exist. Other constructions may be defined as required. For example, the prefix *p* is used to construct pointers. *prowLast* is then the name of a particular pointer to a row type value that defines the end of a closed interval. The standard type constructions are also listed below.

In principle, the conventions can be enriched by new type construction schemes. However, the standard constructions have proved to be sufficient in years of use. It is worth noting that the types for data structures are generally not constructed from the tags of their fields. First of all, constructions with over two components would be unwieldy. More important, the invariant property of data structure, the set of operations in which the constructions participate, seems to be largely independent of the fields of the structure that determine only the representation. We all have had numerous experiences with changes in data structures that left the operations (but not the implementation of the operations) unchanged. Consequently, I recommend the use of a new tag for every new data structure. The tag with some punctuation (upper case initial or all upper case) should also be used as the structure

name in the program. New tags should also be used if the constructions accumulate to the point where readability suffers.

In my experience, tags are more difficult to choose than qualifiers. When a new tag is needed, the first impulse is to use a short, descriptive, common, and generic English term as the 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 the programmer, how do we distinguish the generic use of the common term from the reserved technical usage? By inflection? In the long run, an acronym that is not an English word may work out the best for tags. Related types may then share some of the letters of the acronym. In speech, the acronym may be spelled out, or a pronounceable nickname may be used. When hearing the special names, the informed listener will know that the special technical meaning should be understood. Generic terms should remain free for generic use.

For example, a color graphics program may have a set of internal values that denote colors. What should one call the manifest value for the color red? The obvious choice (which is "wrong" here) is *RED*. The problem with *RED* is that it does not identify its type. Is it a label or a procedure that turns objects *RED*? Even if we know that it is a constant (because it is spelled all caps, for example), there might be several color-related types. Of which one is *RED* an instance? If I see a procedure defined as `paint (color)`, may I call it with *RED* as an argument? Has the word *RED* been used for any other purpose within the program? So we decide to find a tag for the color type and use the word *Red* as a qualifier.

Note that the obvious choice for the qualifier is in fact that the "correct" one! This is because the use of qualifiers are not hampered by any of the above difficulties. Qualifiers are not "exclusive" (or rather they are exclusive only within a smaller set), so we essentially need not take into account the possibility of other uses of the term *Red*. The technical use of the term will be clear to everyone when the qualifier is paired with an obviously technical type tag. Since qualifiers (usually) do not participate in type construction, there is no inherent reason why they would need to be especially short.

Conversely, the tag for the type of the color value should not be "color." Just consider all the other color-related types that may appear in the graphics program (or in a future variant): hardware encoding of color, color map entry number, absolute pointer to color map entry, color values in alternate color mapping mode, hue-brightness-saturation triples, other color values in external interfaces; printers, plotters, interacting external software, and so on. Furthermore, the tag will have to appear in names with constructed types and qualifiers.

A typical arbitrary choice could be *co* (pronounced *see-oh*). Or, if *co* was already taken, *cv*, *cl*, *kl*, and so on. Note that the mnemonic value of the tags is just about average: not too bad, but not too good either. The conventions cannot help with creating names that are inherently mnemonic. Instead, they identify, compress, and contain those parts of the program that are truly individual, thus arbitrary. The lack of inherent meaning should be compensated by ample comments whenever a new tag is introduced. This is a reasonable suggestion because the number of basic tags remains very small, even in a large system.

In conclusion, the name of our quantity would be *coRed*, provided that the color type *co* is properly documented. The value of the name will show later in program segments such as the following:

```
if co == coRed then *mpcpx[coRed]+=dx ...
```

At a glance we can see that the variable *co* is compared with a quantity of its own kind; *coRed* is also used as a subscript to an array whose domain is of the correct type. Furthermore, as we will see, the color is mapped into a pointer to *x*, which is de-referenced (by the ***operator in this example) to yield an *x* type value, which is then incremented by a "delta *x*" type value. Such "dimensional analysis" does not guarantee that the program is completely free from bugs, but it does help to eliminate the most common kinds. It also lends a certain rhythm to the writing of the code: "Let's see, I have a *co* in hand and I need an *x*; do I have a *mpcox*? No, but there is a *mpcpx* that will give me a *px*; **px* will get me the *x*..."

Naming for "Writability"

A good yardstick for choosing a name is to try to imagine that there is an extraordinary reward for two programmers if they can independently come up with the same program text for the same problem. Both programmers know the reward, but cannot otherwise communicate. Such an experiment would be futile, of course, for any sizable problem, but it is a neat goal. The reward of real life is that a program written by someone else, which is identical to what one's own program would have been, is extremely readable and modifiable. By the proper use of the conventions, the idea can be approached very closely, give or take a relatively few tags and possibly some qualifiers. The leverage of the tags is enormous. If they are communicated, are agreed on beforehand, or come from a common source, the goal becomes reachable and the reward may be reaped. This makes the documentation of the tags all the more important.

An example of such a consideration is the discretionary use of qualifiers in small scopes where a quantity's type is likely to be unique, for example in small procedures with a few parameters and locals or in data structures which typically have only a few fields. One might prefer to attach a qualifier even to a quantity with a unique type of "writability," the ability for someone else to come up with the name without hesitation. As many textbooks point out, the "someone else" can be the same programmer sometime in the future revisiting the long-forgotten code.

Conclusion Do not use qualifiers when not needed, even if they seem valuable.

Naming Rules for Procedures

Unfortunately, the simple notion of qualified typed tags does not work well for procedure names. Some procedures do not take parameters or do not return values. The scopes of procedure names tend to be large. The following set of special rules for procedures has worked quite satisfactorily:

1. Distinguish procedure names from other names by punctuation, for example by always starting with a capital letter (typed tags of other quantities are in lower case). This alleviates the problem caused by the large scope.
2. Start the name with the tag of the value that is returned, if any.
3. Express the action of the procedure in one or two words, typically transitive verbs. The words should be punctuated for easy parsing by the reader (a common legal method of punctuation is the use of capital initials for every word).
4. Append the list of tags of some or all of the formal parameters if it seems appropriate to do so.

The last point is contrary to the earlier remarks on data structure naming. When the parameters to a procedure are changed, typically all uses of the procedure will need to be updated. There is an opportunity during the update to change the name as well. In fact, the name change can serve as a useful check that all occurrences have been found. With data structures, the addition or change of a field will not have an effect on all uses of the changed structure type. Typically, if a procedure has only one or two parameters, the inclusion of the parameter tags will really simplify the choice of procedure name.

Table 1. Some examples for procedure names

Name	Description
InitSy	Takes an <i>sy</i> as its argument and initializes it.
OpenFn	<i>fn</i> is the argument. The procedure will "open" the <i>fn</i> . No value is returned.
FcFromBnRn	Returns the <i>fc</i> corresponding to the <i>bn, rn</i> pair given. (The names cannot tell us what the types <i>sy, fn, fc</i> , and so on, are.)

The following is a list of standard type constructions. (*X* and *Y* stand for arbitrary tags. According to standard punctuation, the actual tags are lowercase.)

Table 2. Standard type constructions

pX	Pointer to <i>X</i> .
dX	Difference between two instances of type <i>X</i> . $X + dX$ is of type <i>X</i> .
cX	Count of instances of type <i>X</i> .
mpXY	An array of <i>Ys</i> indexed by <i>X</i> . Read as "map from <i>X</i> to <i>Y</i> ."
rgX	An array of <i>Xs</i> . Read as "range <i>X</i> ." The indices of the array are called: iX

	index of the array <i>rgX</i> .
dnX	(rare) An array indexed by type <i>X</i> . The elements of the array are called: eX (rare) Element of the array <i>dnX</i> .
grpX	A group of <i>Xs</i> stored one after another in storage. Used when the <i>X</i> elements are of variable size and standard array indexing would not apply. Elements of the group must be referenced by means other than direct indexing. A storage allocation zone, for example, is a <i>grp</i> of blocks.
bX	Relative offset to a type <i>X</i> . This is used for field displacements in a data structure with variable size fields. The offset may be given in terms of bytes or words, depending on the base pointer from which the offset is measured.
cbX	Size of instances of <i>X</i> in bytes.
cwX	Size of instances of <i>X</i> in words.

Where it matters, quantities named *mp*, *rg*, *dn*, or *grp* are actually pointers to the structures described above.

One obvious problem with the constructions is that they make the parsing of the types ambiguous. Is *pfC* a tag of its own or is it a pointer to an *fC*? Such questions can be answered only if one is familiar with the specific tags that are used in a program.

The following are standard qualifiers. (The letter *X* stands for any type tag. Actual type tags are in lowercase.)

Table 3. Standard qualifiers

XFirst	The first element in an ordered set (interval) of <i>X</i> values.
XLast	The last element in an ordered set of <i>X</i> values. <i>XLast</i> is the upper limit of a closed interval, hence the loop continuation condition should be: $X \leq XLast$.
XLim	The strict upper limit of an ordered set of <i>X</i> values. Loop continuation should be: $X < XLim$.
XMax	Strict upper limit for all <i>X</i> values (excepting <i>Max</i> , <i>Mac</i> , and <i>Nil</i>) for all other <i>X</i> : $X < XMax$. If <i>X</i> values start with $X=0$, <i>XMax</i> is equal to the number of different <i>X</i> values. The allocated length of a <i>dnx</i> vector, for example, will be typically <i>XMax</i> .
XMac	The current (as opposed to constant or allocated) upper limit for all <i>X</i> values. If <i>X</i> values start with 0, <i>XMac</i> is the current number of <i>X</i> values. To iterate through a <i>dnx</i> array, for example: for $x=0$ step 1 to $xMac-1$ do ... <i>dnx</i> [x] ... or for $ix=0$ step 1 to $ixMac-1$ do ... <i>rgx</i> [ix] ...
XNil	A distinguished <i>Nil</i> value of type <i>X</i> . The value may or may not be 0 or -1.
XT	Temporary <i>X</i> . An easy way to qualify the second quantity of a given type in a scope.

Table 4. Some common primitive types

f	Flag (Boolean, logical). If qualifier is used, it should describe the true state of the flag. Exception: the constants fTrue and fFalse .
w	Word with arbitrary contents.
ch	Character, usually in ASCII text.
b	Byte, not necessarily holding a coded character, more akin to <i>w</i> . Distinguished from the <i>b</i> constructor by the capital letter of the qualifier in immediately following.
sz	Pointer to first character of a zero terminated string.
st	Pointer to a string. First byte is the count of characters <i>cch</i> .
h	<i>pp</i> (in heap).

The following partial example of an actual symbol table routine illustrates the use of the conventions in a "real life" situation. The purpose of this example is not to make any claims about the code itself, but to show how the conventions can help us learn about the code. In fact, some of the names in this routine are standard.

```

1  #include "sy.h"
2  extern int *rgwDic;
3  extern int bsyMac;
4  struct SY *PsySz(char sz[])
5  {
6      {
7          char *pch;
8          int cch;
9          struct SY *psy, *PsyCreate();

```

```

10     int *pbsy;
11     int cwSz;
12     unsigned wHash=0;
13     pch=sz;
14     while (*pch!=0
15         wHash=(wHash<>11+*pch++);
16     cch=pch-sz;
17     pbsy=&rgbsyHash[(wHash&077777)%cwHash];
18     for (; *pbsy!=0; pbsy = &psy->bsyNext)
19     {
20         char *szSy;
21         szSy= (psy=(struct SY*) &rgwDic[*pbsy])->sz;
22         pch=sz;
23         while (*pch==*szSy++)
24         {
25             if (*pch++==0)
26                 return (psy);
27         }
28     }
29     cwSz=0;
30     if (cch>=2)
31         cwSz=(cch-2/sizeof(int)+1;
32     *pbsy=(int *) (psy=PsyCreate(cwSY+cwSz))-rgwDic;
33     Zero((int *)psy,cwSY);
34     bltbyte(sz, psy->sz, cch+1);
35     return (psy);
36 }

```

The tag *SY* is the only product specific type in this routine. The definition of *SY* is found in the include file *sy.h* (fair enough). The type name itself is in all capitals, a common convention.

[Line 2](#)—says that there is an array of words, which is called *Dic*(tionary). Remember that since *Dic* is a qualifier, it is named traditionally.

[Line 3](#)—is the offset pointing beyond the last *sy* (see *b* constructor + *Mac* standard qualifier.) One has to guess at this time that this is used for allocating new *sy*'s. The "base" of the offset would also have to be guessed to be *rgwDic*. Actually, the name *grpsy* would have been better instead of *rgwDic*, from this local perspective. In the real program, the *rgwDic* area is used for a number of different purposes, hence the "neutral" name.

[Line 4](#)—is a procedure declaration. Procedure returns a pointer to an *SY* as the result. The parameter must be a zero-terminated string.

[Lines 7-12](#)—declare quantities. The usages should be clear from the names. For example, *cwSz* is the number of words in some string (probably the argument), *pbsy* is a pointer to an offset of an *sy* (*p* constructor + *b* constructor). The only qualifier used here is in *wHash*—the hash code.

[Line 13](#)—*pch* will be a pointer to the first character of *sz*.

[Line 16](#)—*cch* is the count of characters (*c* constructor) ostensibly, in *sz*.

[Line 17](#)—*cwHash* is the number of words in the hash table (I would have called it *ibsyMax*). In a way, the qualifier on *rgbsyHash* could be omitted, but it helps identify the hash table in external contexts.

[Lines 17-18](#)—note the opportunities for dimensional checking:

```

pbsy = &rgbsy[...] follows from pX = &rgX[...]
pbsy = &psy->bsyNext follows from pX=&pY->X; or pX = &Y.X

```

So even the use of *->* instead of *.* follows from local context. The *p* on the left hand side signals the need for the *&* on the right.

[Line 20](#)—introduces a new *sz*, qualified to distinguish it from the argument. The qualifier, very appropriately, is the source of the datum, *Sy*.


[Line 23](#)—given the use of *szSy* in this line, the name *pchSy* would have been a little more appropriate. No harm done, however.

[Lines 29-31](#)—this strange code has to do with the fact that the declaration of *SY* includes 2 bytes of *sz*, so that *cwSz* is really the number of words in the *sz-2* bytes! This should deserve a comment or at least a qualifier *M2* (minus 2) or the like. *cwSY* is the length of the *SY* structure in words. The all caps qualifier is not strictly standard, but it helps to associate the quantity with the declaration of *SY*, rather than with any random *sy* instance.

PsyCreate is a good procedure name; *PsyCreateCw* would have been even better. In line 32 we can also see an example of dimensional checking: While we have a *psy* inside the parenthesis, we need a *bsy* for the left side (**pbsy=bsy!*) so we subtract the "base" of the *bsy* from the *psy*.

$bX + base = pX$; hence: $bX = pX - base$.

In closing, it is evident that the conventions participated in making the code more correct, easier to write, and easier to read. Naming conventions cannot guarantee *good* code, however; only the skill of the programmer can.

 Print  E-Mail  Add to Favorites

How would you rate the quality of this content?

1 2 3 4 5 6 7 8 9
 Poor Outstanding

Tell us why you rated the content this way. (optional)

Submit

Average rating:
6 out of 9



407 people have rated this page

[Contact Us](#) | [E-Mail this Page](#) | [MSDN Flash Newsletter](#) | [Legal](#)

© 2003 Microsoft Corporation. All rights reserved. [Terms of Use](#) [Privacy Statement](#) [Accessibility](#)