

Hungarian Notation

First, a little disclaimer:

This is the version of Hungarian Notation that *I* use. You may not agree with everything I state here. Feel free to check Charles Simonyi's [original document](#), if you prefer. The purpose of this guide is to provide an easier- to- read reference.

What is Hungarian notation?

Hungarian Notation is a naming convention that (in theory) allows the programmer to determine the type and use of an identifier (variable, function, constant, etc.) It is frequently used in Windows programming, so a quick guide (such as this one) may be useful if you're working with Windows code but don't know the naming scheme.

Variables

Variable names are probably the most common type of identifiers. A variable name in Hungarian notation consists of three parts: the *prefix* (or *constructor*), *base type* (or *tag*), and *qualifier*. Not all of these elements will be present in all variable names - - the only part that is really needed is the *base type*.

Base Types (Tags)

The *Tag* is NOT necessarily one of the types directly provided by the programming language; it may be application- defined (for example, a type *dbr* might represent a database record structure). *Tags* are short (usually two or three letters) descriptive reminders about the type of value the variable stores. This type will usually only be useful to someone who knows the application and knows what the basic types the application uses are; for example, a tag *co* could just as easily refer to a coordinate, or a color. Within a given application, however, the *co* would always have a specific meaning - - all *co*'s would refer to the same type of object, and all references to that type of object would use the tag *co*.

Common Tags

These are many basic types that are used in any application. These tags are used for these types.

Tag	Description
f	Boolean flag. The <i>qualifier</i> should be used to describe the condition that causes the flag to be set (for example, <i>fError</i> might be used to indicate a variable that is set when an error condition exists, and clear when there is no error). The actual data representation may be a byte, a word, or even a single bit in a bitfield.
ch	A single- byte character.
	A machine word (16 bits on Win3.1 X86 machines). This is a somewhat

w	ambiguous tag, and it is often better to use a more specific name. (For example, a word that represents a count of characters is better referred to as a <i>cch</i> than as a <i>w</i> . See the <i>prefixes</i> section for an explanation of the <i>cch</i> notation.)
b	A byte (typically 8 bits). See the warnings for <i>w</i> .
l	A long integer (typically 32 bits). See the warnings for <i>w</i> .
dw	A double word (typically 32 bits, but this is NOT necessarily the same thing as a long, depending on the machine architecture!). Again, the same comments about ambiguity apply here as for <i>w</i> .
u	An unsigned value. Usually more accurately used as a prefix with one of the integer types described above. For example, <i>uw</i> is an unsigned word.
r	A single-precision real number (float)
d	A double-precision real number (double)
bit	A single bit. Typically this is used to specify bits within other types; it is usually more appropriate to use <i>f</i> .
v	A void. (Rather C-specific, meaning the type is not specified.) This type will probably never be used without the <i>p</i> prefix. This would usually be used for generic subroutines (such as <i>malloc</i> and <i>free</i>) that work with pointers but don't need to refer to the specific type of data referenced by those pointers.
st	A Pascal-type string. That is, the first byte contains the length of the string, and the remainder contains the actual characters in the string.
sz	A null-terminated (C-style) string.
fn	A function. This will almost always have a <i>p</i> prefix, as about the only useful thing you can do with a function (from a <i>variable's</i> perspective) is take the address of it.

Prefixes (Constructors)

The base types are not usually sufficient to describe a variable, since variables frequently refer to more complex types. For example, you may have a pointer to a database record, or an array of coordinates, or a count of colors. In Hungarian notation, these extended types are described by the variable's prefix. The complete type of a variable is given by the combination of the prefix(es) and base type. Yes, it is possible to have more than one prefix -- for example, you may have a pointer to an array of database records.

Common Constructors

It is not usually necessary to create a new prefix, although it is certainly possible to do so. The list of standard prefixes should be sufficient for most uses, however.

Constructor	Description
p	A pointer.
lp	A long (far) pointer. (Used on machines with a segmented architecture, such as X86's under DOS or Win3.1).
hp	A huge pointer. (Similar to a far pointer, except that it correctly handles crossing segment boundaries during pointer arithmetic.)
	An array. An <i>rgch</i> is an array of characters; a <i>pch</i> could point to a specific element in this array. (The notation comes from viewing an

rg	array as a mathematical function - - the input is the index, and the output is the value at that index. So the entire array is essentially the "range" of that function.)
i	An index (into an array). For example, an ich could be used to index into an rgch. I've also seen this used for resource IDs under Windows (which makes sense if you think about it - - a resource ID is an index into a resource table).
c	A count. cch could be the count of characters in the rgch. (As another example, note that the first byte of an st is the cch of that string.)
d	The difference between two instances of a type. For example, given a type x used to represent an X- coordinate on a graph, a dx could contain the difference on the X- axis of two such coordinates.
h	A handle. Handles are commonly used in Windows programming; they represent resources allocated by the system and handed back to the application. On other systems, a "handle" might be a pointer to a pointer, in which case it might be clearer to use a pp (or lplp if appropriate).
mp	A specific type of array, a mapping. This prefix is followed by two types, rather than one. The array represents a mapping function from the first type to the second. For example, mpwErrisz could be a mapping of error codes (wErr) to indexes of message strings (isz).
v	A global variable (personally I prefer g for this)
s	A static variable
k	A const variable (C++)

Examples

Tags and constructors are both in lower case, with no separating punctuation, so some ambiguity is possible if you are not careful in choosing your representations. For example, you probably shouldn't use pfn to represent a structure you've defined, as it could be taken as a pointer (p) to a function (fn). (Even if you *ARE* careful, some ambiguity is still possible. For example, if you have a handle to a pointer (unlikely, but who knows?), you'd want to represent this as hp, which also means huge pointer. Cases like these should be rare, however, and the true type should still be distinguishable from the context of the code.)

Here are some further examples of constructors + tags:

Variable	Description
pch	A pointer to a character.
ich	An index into an array of characters.
rgsz	An array of null-terminated strings (most likely the values stored in the array are actually pointers to the strings, so you could arguably also use rgpsz).
rgrgx	A two- dimensional array of x's. (An array of arrays of x's.)
pisz	A pointer to an index into an array of null-terminated strings. (Or possibly a pointer to a resource ID of a string - - the real meaning should be clear within the context of the code.)
lpcw	Far pointer to a count of words.

Qualifiers

Although the combination of constructors and tags are enough to specify the type of a variable, it won't be sufficient to *distinguish* the variable from others of the same type. This is where the third (optional) part of a variable name, the *qualifier*, comes in. A qualifier is a short descriptive word or words (or reasonable facsimile) that describes *HOW* the variable is used. Some kind of punctuation should be used to distinguish the qualifier from the constructor + tag portion. Typically this is done by making the first letter of the qualifier (or of each qualifier if you choose to use more than one word) upper-case.

The use of many variables will fall into the same basic categories, so there are several standard qualifiers:

Qualifier	Description
First	The first element in a set. This will often be an index or a pointer (for example, <code>pchFirst</code> or <code>iwFirst</code>).
Last	The last element in a set. Both <code>First</code> and <code>Last</code> refer to valid values that <i>are</i> in a given set, and are often paired, such as in this sample C loop: <pre>for (iw = iwFirst; iw <= iwLast; iw++) { ... }</pre>
Mn	The first element in a set. Similar to <code>First</code> , but <code>Mn</code> always refers to the first actual element in a set, while <code>First</code> may be used to indicate the first element actually dealt with (if you're working with a substring, for example).
Max	The upper limit in a set. This is <i>NOT</i> a valid value; <code>xMax</code> is usually equivalent to <code>xLast + 1</code> . The above example loop could also be written as <pre>for (iw = iwFirst; iw < iwMax; iw++) { ... }</pre> I've also seen <code>Lim</code> used to indicate the limit in much the same manner.
Mac	The <i>current</i> upper limit in a set. This is similar to <code>Max</code> , but is used where the upper limit can vary (for example, a variable length structure).
Sav	A temporary saved value; usually used when temporarily modifying variables that you want to restore later.
T	A temporary variable -- one which will be used quickly in a given context and then disposed of or reused.
Src	A source. This is usually paired with <code>Dest</code> in copy/transfer operations.
Dest	A destination. This is usually paired with <code>Src</code> .

Structures and structure members

Structures are usually by definition their own types, so a given structure usually defines its own tag (for example, the `dbr` I used earlier in this document).

Structure members should simply be named the same way variables are. Since the context is usually only within the structure itself, name conflicts are less likely, so qualifiers are often not as necessary. If you have multiple instances of a variable type within the structure, you'll still need the qualifiers, of course (for example, if you're creating a structure containing name and address string records, you could name them `szName` and `szAddress`). If the language does not support separate contexts for each structure (I think the Microsoft Macro Assembler (MASM) falls into this category, but I haven't worked with it in a few years), then the structure name is appended to the name of the member as a qualifier. (So the examples given above might be named `szNameDbr` and `szAddressDbr` if these fields appeared in the `dbr` structure.)

Procedures

The simple rules for naming variables don't always work quite as well for procedures. This is because specifying what the procedure actually *does* is important, and many procedures won't have a return value. Also, the context for procedures is usually the entire program, so you have more chance for naming conflicts. To handle these problems, a few modifications are made to the rules:

1. Procedure names are distinguished from variable names by using some punctuation - - for example, function names have the first letter capitalized while variable names begin with lowercase letters.
2. If the procedure *explicitly* returns a value (as opposed to implicitly returning one through a variable argument), then the procedure name will begin with the type of value that is returned.
3. If the procedure is a true function (that is, it operates on its parameters and returns a value with no side-effects), then it is typical to name it `XFromYZ...`, where `X` is the type returned and `Y`, `Z`, etc., are the types of the parameters. For example, `DxFromWnd(hwnd)` (or possibly `DxFromHwnd(hwnd)` if you really want to be specific) could be used for a function that returns the width of a window.
4. If the procedure has side-effects, then follow the type (if any) with a few words that describe what the procedure does. Each word should be capitalized. For example, `FTryMove()` could be used to indicate a procedure that checks the validity of a move (in a game, for instance), and returns a boolean value (true/false) to indicate if the move is valid.
5. If the procedure operates on an object, the type of the object should be appended to the name. For example, `InitFoo(pfoo)` could indicate a procedure that initializes a structure `foo` (or more accurately in this case, a structure `foo` that the procedure is given a pointer to).

Macros and constants

Macros are usually handled the same way as procedures. Constants may be handled as variables (such as `fTrue` and `fFalse`), although you'll often see constants defined in all upper-case (`IWFIRST`, for example). Some people will use underscores to separate parts of a constant name if they capitalize them (`I_WFIRST`). If I remember correctly, this capitalization is *NOT* really a part of "proper" Hungarian, but I use it myself to distinguish between constants and variables.

Labels

If you need a label for some reason, it can be considered to be a variation on a procedure - - labels are effectively identifiers specifying a piece of code. Since labels don't take parameters or return a value, no types are specified. `EndLoop` or `OutOfMem` are typical examples.

Credits and thanks

Thanks go especially to Charles Simonyi, who wrote the first document (that I'm aware of, anyway) about Hungarian Notation.

Thanks also to [Samir Ramji](#) for pointing out that I was missing the `dw` type and `s` and `k` prefixes.

Greg Legowski, gregleg@pobox.com
email - - For when it absolutely has to get lost at the speed of light.
[PGP Public Key](#) and [Geek Code 3.12](#) available.

Last modified: Tue Jun 02 13:23:19 1998