WIKIPEDIA

Hungarian notation

Hungarian notation is an identifier naming convention in computer programming, in which the name of a variable or function indicates its intention or kind, and in some dialects its type. The original Hungarian notation uses intention or kind in its naming convention and is sometimes called Apps Hungarian as it became popular in the Microsoft Apps division in the development of Word, Excel and other apps. As the Microsoft Windows division adopted the naming convention, they used the actual data type for naming, and this convention became widely spread through the Windows API; this is sometimes called Systems Hungarian notation.

Hungarian notation was designed to be language-independent, and found its first major use with the BCPL programming language. Because BCPL has no data types other than the machine word, nothing in the language itself helps a programmer remember variables' types. Hungarian notation aims to remedy this by providing the programmer with explicit knowledge of each variable's data type.

In Hungarian notation, a variable name starts with a group of lower-case letters which are mnemonics for the type or purpose of that variable, followed by whatever name the programmer has chosen; this last part is sometimes distinguished as the *given name*. The first character of the given name can be capitalized to separate it from the type indicators (see also CamelCase). Otherwise the case of this character denotes scope.

> **Simonyi**: ...BCPL [had] a single type which was a 16-bit word... not that it matters.
>
> **Booch**: Unless you continue the Hungarian notation.
>
> **Simonyi**: Absolutely... we went over to the typed languages too later ... But ... we would look at one name and I would tell you exactly a lot about that...[1]

Contents

History

The original Hungarian notation was invented by Charles Simonyi, a programmer who worked at Xerox PARC circa 1972–1981, and who later became Chief Architect at Microsoft. The name of the notation is a reference to Simonyi's nation of origin, and also, according to Andy Hertzfeld, because it made programs "look like they were written in some inscrutable foreign language".[2] Hungarian people's names are "reversed" compared to most other European names; the family name precedes the given name. For example, the anglicized name "Charles Simonyi" in Hungarian was originally "Simonyi Károly". In the same way, the type name precedes the "given name" in Hungarian notation. The similar Smalltalk "type last" naming style (e.g. aPoint and lastPoint) was common at Xerox PARC during Simonyi's tenure there.

Simonyi's paper on the notation referred to prefixes used to indicate the "type" of information being stored.[3][4] His proposal was largely concerned with decorating identifier names based upon the semantic information of what they store (in other words, the variable's *purpose*). Simonyi's notation came to be called Apps Hungarian, since the convention was used in the applications division of Microsoft. Systems Hungarian developed later in the Microsoft Windows development team. Apps Hungarian is not entirely distinct from what became known as Systems Hungarian, as some of Simonyi's suggested prefixes contain little or no semantic information (see below for examples).[4]

Systems Hungarian vs. Apps Hungarian

Where Systems notation and Apps notation differ is in the purpose of the prefixes.

In Systems Hungarian notation, the prefix encodes the actual data type of the variable. For example:

- `lAccountNum` : variable is a *long integer* (`"l"`);
- `arru8NumberList` : variable is *an **arr**ay of **u**nsigned **8**-bit integers* (`"arru8"`);
- `bReadLine(bPort,&arru8NumberList)` : function with a byte-value return code.
- `strName` : Variable represents a string (`"str"`) containing the name, but does not specify how that string is implemented.

Apps Hungarian notation strives to encode the logical data type rather than the physical data type; in this way, it gives a hint as to what the variable's purpose is, or what it represents.

- `rwPosition` : variable represents a *row* (`"rw"`);
- `usName` : variable represents an *unsafe string* (`"us"`), which needs to be "sanitized" before it is used (e.g. see code injection and cross-site scripting for examples of attacks that can be caused by using raw user input)
- `szName` : variable is a ***z**ero-terminated **s**tring* (`"sz"`); this was one of Simonyi's original suggested prefixes.

Most, but not all, of the prefixes Simonyi suggested are semantic in nature. To modern eyes, some prefixes seem to represent physical data types, such as `sz` for strings. However, such prefixes were still semantic, as Simonyi intended Hungarian notation for languages whose type systems could not distinguish some data types that modern languages take for granted.

The following are examples from the original paper:[3]

- p*X* is a pointer to another type *X*; this contains very little semantic information.
- *d* is a prefix meaning difference between two values; for instance, *dY* might represent a distance along the Y-axis of a graph, while a variable just called *y* might be an absolute position. This is

entirely semantic in nature.

- *sz* is a null- or zero-terminated string. In C, this contains some semantic information because it is not clear whether a variable of type *char** is a pointer to a single character, an array of characters or a zero-terminated string.
- *w* marks a variable that is a word. This contains essentially no semantic information at all, and would probably be considered Systems Hungarian.
- *b* marks a byte, which in contrast to w might have semantic information, because in C the only byte-sized data type is the *char*, so these are sometimes used to hold numeric values. This prefix might clear ambiguity between whether the variable is holding a value that should be treated as a character or a number.

While the notation always uses initial lower-case letters as mnemonics, it does not prescribe the mnemonics themselves. There are several widely used conventions (see examples below), but any set of letters can be used, as long as they are consistent within a given body of code.

It is possible for code using Apps Hungarian notation to sometimes contain Systems Hungarian when describing variables that are defined solely in terms of their type.

Relation to sigils

In some programming languages, a similar notation now called sigils is built into the language and enforced by the compiler. For example, in some forms of BASIC, `name$` names a string and `count%` names an integer. The major difference between Hungarian notation and sigils is that sigils declare the type of the variable in the language, whereas Hungarian notation is purely a naming scheme with no effect on the machine interpretation of the program text.

Examples

- `bBusy` : boolean
- `chInitial` : char
- `cApples` : count of items
- `dwLightYears` : double word (Systems)
- `fBusy` : flag (or float)
- `nSize` : integer (Systems) or count (Apps)
- `iSize` : integer (Systems) or index (Apps)
- `fpPrice` : floating-point
- `decPrice` : decimal
- `dbPi` : double (Systems)
- `pFoo` : pointer
- `rgStudents` : array, or range
- `szLastName` : zero-terminated string
- `u16Identifier` : unsigned 16-bit integer (Systems)
- `u32Identifier` : unsigned 32-bit integer (Systems)
- `stTime` : clock time structure
- `fnFunction` : function name

The mnemonics for pointers and arrays, which are not actual data types, are usually followed by the type of the data element itself:

- `pszOwner` : pointer to zero-terminated string
- `rgfpBalances` : array of floating-point values
- `aulColors` : array of unsigned long (Systems)

While Hungarian notation can be applied to any programming language and environment, it was widely adopted by Microsoft for use with the C language, in particular for Microsoft Windows, and its use remains largely confined to that area. In particular, use of Hungarian notation was widely evangelized by Charles Petzold's *"Programming Windows"*, the original (and for many readers, the definitive) book on Windows API programming. Thus, many commonly seen constructs of Hungarian notation are specific to Windows:

- For programmers who learned Windows programming in C, probably the most memorable examples are the `wParam` (word-size parameter) and `lParam` (long-integer parameter) for the WindowProc() function.
- `hwndFoo` : handle to a window
- `lpszBar` : long pointer to a zero-terminated string

The notation is sometimes extended in C++ to include the scope of a variable, optionally separated by an underscore.[5][6] This extension is often also used without the Hungarian type-specification:

- `g_nWheels` : member of a global namespace, integer
- `m_nWheels` : member of a structure/class, integer
- `m_wheels`, `_wheels` : member of a structure/class
- `s_wheels` : static member of a class
- `c_wheels` : static member of a function

In JavaScript code using jQuery, a `$` prefix is often used to indicate that a variable holds a jQuery object (versus a plain DOM object or some other value).[7]

Advantages

(Some of these apply to Systems Hungarian only.)

Supporters argue that the benefits of Hungarian Notation include:[3]

- The symbol type can be seen from its name. This is useful when looking at the code outside an integrated development environment — like on a code review or printout — or when the symbol declaration is in another file from the point of use, such as a function.
- In a language that uses dynamic typing or that is untyped, the decorations that refer to types cease to be redundant. In such languages variables are typically not declared as holding a particular type of data, so the only clue as to what operations can be done on it are hints given by the programmer, such as a variable naming scheme, documentation and comments. As mentioned above, Hungarian Notation expanded in such a language (BCPL).
- The formatting of variable names may simplify some aspects of code refactoring (while making other aspects more error-prone).

- Multiple variables with similar semantics can be used in a block of code: dwWidth, iWidth, fWidth, dWidth.
- Variable names can be easy to remember from knowing just their types.
- It leads to more consistent variable names.
- Inappropriate type casting and operations using incompatible types can be detected easily while reading code.
- In complex programs with many global objects (VB/Delphi Forms), having a basic prefix notation can ease the work of finding the component inside of the editor. For example, searching for the string `btn` might find all the Button objects.
- Applying Hungarian notation in a narrower way, such as applying only for member variables, helps avoid naming collision.
- Printed code is more clear to the reader in case of datatypes, type conversions, assignments, truncations, etc.

Disadvantages

Most arguments against Hungarian notation are against *Systems* Hungarian notation, not *Apps* Hungarian notation. Some potential issues are:

- The Hungarian notation is redundant when type-checking is done by the compiler. Compilers for languages providing strict type-checking, such as Pascal, ensure the usage of a variable is consistent with its type automatically; checks by eye are redundant and subject to human error.
- Most modern integrated development environments display variable types on demand, and automatically flag operations which use incompatible types, making the notation largely obsolete.
- Hungarian Notation becomes confusing when it is used to represent several properties, as in `a_crszkvc30LastNameCol`: a constant reference argument, holding the contents of a database column `LastName` of type varchar(30) which is part of the table's primary key.
- It may lead to inconsistency when code is modified or ported. If a variable's type is changed, either the decoration on the name of the variable will be inconsistent with the new type, or the variable's name must be changed. A particularly well known example is the standard WPARAM type, and the accompanying wParam formal parameter in many Windows system function declarations. The 'w' stands for 'word', where 'word' is the native word size of the platform's hardware architecture. It was originally a 16 bit type on 16-bit word architectures, but was changed to a 32-bit on 32-bit word architectures, or 64-bit type on 64-bit word architectures in later versions of the operating system while retaining its original name (its true underlying type is UINT_PTR, that is, an unsigned integer large enough to hold a pointer). The semantic impedance, and hence programmer confusion and inconsistency from platform-to-platform, is on the assumption that 'w' stands for a two byte, 16-bit word in those different environments.
- Most of the time, knowing the use of a variable implies knowing its type. Furthermore, if the usage of a variable is not known, it cannot be deduced from its type.
- Hungarian notation reduces the benefits of using code editors that support completion on variable names, for the programmer has to input the type specifier first, which is more likely to collide with other variables than when using other naming schemes.
- It makes code less readable, by obfuscating the purpose of the variable with type and scoping prefixes.[8]
- The additional type information can insufficiently replace more descriptive names. E.g. sDatabase does not tell the reader what it is. databaseName might be a more descriptive name.
- When names are sufficiently descriptive, the additional type information can be redundant. E.g. firstName is most likely a string. So naming it sFirstName only adds clutter to the code.

- It's harder to remember the names.
- Multiple variables with **different** semantics can be used in a block of code with similar names: *dwTmp, iTmp, fTmp, dTmp*.
- Placing data type or intent character identifiers as a prefix to the field or variable's Given name subverts the ability, in some programming environments, to jump to a field or variable name, alphabetically, when the user begins typing the name. FileMaker, [9] for example, is one such programming environment. It may be preferable when using one of these programming environments to instead suffix Given names with such identifying characters.

Notable opinions

- Robert Cecil Martin (against Hungarian notation and all other forms of encoding):

> ... nowadays HN and other forms of type encoding are simply impediments. They make it harder to change the name or type of a variable, function, member or class. They make it harder to read the code. And they create the possibility that the encoding system will mislead the reader.[10]

- Linus Torvalds (against Systems Hungarian):

> Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged—the compiler knows the types anyway and can check those, and it only confuses the programmer.[11]

- Steve McConnell (for Apps Hungarian):

> Although the Hungarian naming convention is no longer in widespread use, the basic idea of standardizing on terse, precise abbreviations continues to have value. Standardized prefixes allow you to check types accurately when you're using abstract data types that your compiler can't necessarily check.[12]

- Bjarne Stroustrup (against Systems Hungarian for C++):

> No I don't recommend 'Hungarian'. I regard 'Hungarian' (embedding an abbreviated version of a type in a variable name) as a technique that can be useful in untyped languages, but is completely unsuitable for a language that supports generic programming and object-oriented programming — both of which emphasize selection of operations based on the type and arguments (known to the language or to the run-time support). In this case, 'building the type of an object into names' simply complicates and minimizes abstraction.[13]

- Joel Spolsky (for Apps Hungarian):

> If you read Simonyi's paper closely, what he was getting at was the same kind of naming convention as I used in my example above where we decided that `us` meant unsafe string and `s` meant safe string. They're both of type `string`. The compiler won't help you if you assign one to the other and Intellisense [an Intelligent code completion system] won't tell you bupkis. But they are semantically different. They need to be interpreted differently and treated differently and some kind of conversion function will need to be called if you assign one to the other or you will have a runtime bug. If you're lucky. There's still a tremendous amount of value to Apps Hungarian, in that it increases collocation in code, which makes the code easier to read, write, debug and maintain, and, most importantly, it makes wrong code look wrong.... (Systems Hungarian) was a subtle but complete misunderstanding of Simonyi's intention and practice.[4]

- Microsoft's Design Guidelines[14] discourage developers from using Systems Hungarian notation when they choose names for the elements in .NET class libraries, although it was common on prior Microsoft development platforms like Visual Basic 6 and earlier. These Design Guidelines are silent on the naming conventions for local variables inside functions.

See also

- Leszynski naming convention, Hungarian notation for database development
- Polish notation
- PascalCase

References

1. "Oral History of Charles Simonyi" (http://archive.computerhistory.org/resources/access/text/2015/06/102702232-05-01-acc.pdf) (PDF). *Archive.computerhistory.org\accessdate=5 August 2018*.
2. Rosenberg, Scott (1 January 2007). "Anything You Can Do, I Can Do Meta" (https://www.technologyreview.com/2007/01/01/227178/anything-you-can-do-i-can-do-meta/). *MIT Technology Review*. Retrieved 21 July 2022.
3. Charles Simonyi (November 1999). "Hungarian Notation" (http://msdn2.microsoft.com/en-us/library/aa260976(VS.60).aspx). *MSDN Library*. Microsoft Corp.
4. Spolsky, Joel (2005-05-11). "Making Wrong Code Look Wrong" (http://www.joelonsoftware.com/articles/Wrong.html). *Joel on Software*. Retrieved 2005-12-13.
5. "Mozilla Coding Style" (https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style#Prefixes). *Developer.mozilla.org*. Retrieved 17 March 2015.
6. "Webkit Coding Style Guidelines" (http://www.webkit.org/coding/coding-style.html#names-data-members). *Webkit.org*. Retrieved 17 March 2015.
7. "Why would a JavaScript variable start with a dollar sign?" (https://stackoverflow.com/questions/205853/why-would-a-javascript-variable-start-with-a-dollar-sign). *Stack Overflow*. Retrieved 12 February 2016.
8. Jones, Derek M. (2009). *The New C Standard: A Cultural and Economic Commentary* (http://www.coding-guidelines.com/cbook/cbook1_2.pdf) (PDF). Addison-Wesley. p. 727. ISBN 978-0-201-70917-9.
9. "Make an app for any task - FileMaker — An Apple Subsidiary" (http://www.filemaker.com). *Filemaker.com*. Retrieved 5 August 2018.

10. Martin, Robert Cecil (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Redmond, WA: Prentice Hall PTR. ISBN 978-0-13-235088-4.
11. "Linux kernel coding style" (https://www.kernel.org/doc/html/v4.10/process/coding-style.html). *Linux kernel documentation*. Retrieved 9 March 2018.
12. McConnell, Steve (2004). *Code Complete* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1967-0.
13. Stroustrup, Bjarne (2007). "Bjarne Stroustrup's C++ Style and Technique FAQ" (http://www.stroustrup.com/bs_faq2.html#Hungarian). Retrieved 15 February 2015.
14. "Design Guidelines for Developing Class Libraries: General Naming Conventions" (http://msdn2.microsoft.com/en-us/library/ms229045.aspx). Retrieved 2008-01-03.

External links

- Meta-Programming: A Software Production Method (https://web.archive.org/web/20180519042122/http://www.parc.com/publication/1940/meta-programming.html) Charles Simonyi, December 1976 (PhD Thesis)
- Hugarian notation - it's my turn now :) (https://blogs.msdn.microsoft.com/larryosterman/2004/06/22/hugarian-notation-its-my-turn-now/) – Larry Osterman's WebLog
- Hungarian Notation (http://msdn.microsoft.com/en-us/library/aa260976%28VS.60%29.aspx) (MSDN)
- HTML version of Doug Klunder's paper (http://www.byteshift.de/msg/hungarian-notation-doug-klunder)
- RVBA Naming Conventions (http://www.xoc.net/standards/rvbanc.asp)
- Coding Style Conventions (http://msdn.microsoft.com/en-us/library/aa378932%28VS.85%29.aspx) (MSDN)