

Chapter 2

Names

“Ah, variable names. Length is not a virtue in a name; clarity of expression is. A global variable rarely used may deserve a long name, `maxphysaddr` say. An array index used on every line of a loop needn't be named any more elaborately than `i`. ... As in all other aspects of readable programming, consistency is important in naming. If you call one variable `maxphysaddr`, don't call its cousin `lowestaddress`. ... I prefer minimum-length but maximum-information names, and then let the context fill in the rest.”

- Rob Pike, “Notes on Programming in C” [Pike89]

2.1 General naming conventions

2.1.1. Use simple names for simple things, more complex names for more complex things. (Recommended)

Rationale

A loop variable used as a counter or an index whose scope is limited to that of its controlling statement doesn't need to have a long, elaborate name. Variables that have a wider scope, or are used in places in code separated by many lines, or variables that convey information of a complex nature could have longer, more descriptive names. Use your best judgment.

Examples

In the following code snippet, the variable `i` has a clear and understandable purpose. However, the purpose of variable `y` is not clear. Can you tell what `y` is for?

```
for (i = 0; i <= maxTimesToRetry(); i++) {
    x = getBoardInfo();
    if (x != ERROR)
        break;
}
// x now contains the information. Do stuff with x.
y = x & 0xff;
```

See how much clearer the last few lines are with more descriptive variable names:

```
for (i = 0; i <= maxTimesToRetry(); i++) {
    boardInfo = getBoardInfo();
    if (boardInfo != ERROR) {
        break;
    }
}
slotNumber = boardInfo & 0xff;
```

Also, following the rules in sections 3.3 and 3.4, the constant `0xff` should be hidden in an abstraction, such as this:

```
slotNumber = getSlotNumber(boardInfo);
```

Reference

[Kernighan1999]

2.1.2. Use a common module prefix for global names. (Recommended)

Rationale

It helps humans to organize and understand code if global-scoped symbols are named with a prefix identifying the module they are part of. This also reduces the opportunity for name collision in the global name space. Document the use of module name prefixes in the principal header file(s) that are associated with the module, and anywhere else that a programmer would expect to find general information about the module.

Example

The prefix “vdr” shows that these names are from the vdr module:

```
vdrStartup()  
vdrContext []  
vdrDebug
```

In C++ an explicit scope resolution can serve to identify the class or namespace to which a symbol belongs, as described in section 4.3 in the discussion about scope of visibility.

2.1.3. Names must be unique in the first 31 characters. (Required)

Rationale

Names that are too short make it hard to read code because they don't convey any information to the reader. On the other extreme, very long names make it hard to read code also. Find a good compromise between names that are so short that they are meaningless, and names that are so long that code verbosity hinders understanding. Instead of using a very long name, try using an abbreviated name, with a comment at one or more places to explain what it is. Also, for maximum portability, choose names that are unique in their first 31 characters. This applies to names of variables, objects, functions, type names, etc.

Why 31 characters and not 29 or 32? Experience has shown us that we have the best results from our tools if identifier names differ in the first 31 characters.

2.1.4. Use embedded capital letters instead of underscores. (Required)

Rationale

Except for preprocessor macros, start the second and subsequent word in an identifier with a capital letter. When an acronym or abbreviation is part of an identifier, capitalize only the first letter of the acronym or abbreviation.

Example

Change:

```
max_time_limit
get_time_delay()
```

To:

```
maxTimeLimit
getTimeDelay()
```

Change:

```
firstVITCpointer
```

To:

```
firstVItcPointer
```

2.1.5. Avoid redundancy in names. (Recommended)

Rationale

Verbosity hinders readability. Names that are longer than needed can result in lines that must wrap onto the next line, which makes code harder to read. Add redundancy to names only if it helps the next programmer understand the code better.

Examples

```
void printText(const char *messageArg); // "Arg" is redundant
typedef txtBuffer<char> textBufferType; // "Type" is redundant
typedef txtBuffer<char> textBuffer; // better
void printText(textBuffer& messageRef); // "Ref" is redundant
while (loopNum < MAX_LOOPS) { // "Num" is redundant
while (loop < MAX_LOOPS) { // better
enum mixerStateEnum { } // "Enum" is redundant
int maxLengthValue; // "Value" is redundant
```

2.1.6. Don't use names reserved for the implementation. (Required)

Rationale

Some names are reserved for use by the operating system and compiler. Honor this convention for two reasons: (1) it helps document which names are part of the operating system or compiler implementation, and (2) it avoids name clashes with identifiers that may be used by the implementation.

“Names starting with an underscore are reserved for special facilities in the implementation and the run-time environment, so such names should not be used in application programs.”

-- [Stroustrup1997], p. 81.

2.2 Macros

2.2.1. If you have to use a preprocessor macro, use all upper case with underscores between words. (Required)

Rationale

It's tradition. It shows that the identifier is a preprocessor macro. Underscore characters help divide the name into parts so that it is more readable. Note that inline functions are preferred to function-like macros; see item 8.1.

Example

Change:

```
#define myprintf
```

To:

```
#define MY_PRINTF
```

2.3 Name adornments

2.3.1. Avoid “Hungarian notation” or other prefixes unless you have a reason to expose the implementation. (Recommended)

Rationale

Much of the so-called Hungarian Notation (and similar naming conventions) is directly opposed to basic principals of object oriented programming. Hungarian notation uses prefixes on identifiers that reveal their implementation, while Object Oriented

Programming 101 tells us that we usually want to hide the implementation. You can't have both. If you want the name of an identifier to reveal that it is specifically a `const` pointer to a global function that returns a class of type `D`, then you can name it `cpgfclMyFunction()`. But now you are stuck. Little is abstracted or encapsulated. For example, you cannot change the function return type from a class to an integer without renaming every occurrence of `cpgfclMyFunction()` to `cpgfiMyFunction()`.

However, we do acknowledge exceptions to the rule. There are times that you want names to distinguish between an object and a pointer to an object, in order to distinguish between value semantics and reference semantics. Do so if you can defend the need to make the distinction.

Also, a data member of a class may begin with “`m_`” to expose the fact that it is a member, if you can give a convincing reason that exposing that fact helps code readability.

Example

Change:

```
const char *pfcszGetErrString(int errorCode);
```

To:

```
const char *getErrString(int errorCode);
```

Examples

```
firstResource    // the information itself
pFirstResource  // must be a pointer to the information
itResourceList  // "it" is a common prefix for container iterators
m_lastPosition  // member data of a class
```

Reference

[Schmidt1996]

2.4 Function names

2.4.1. *Predicate functions should be Boolean type and prefixed with “is”. (Recommended)*

Rationale

A function that returns one of two possible values is usually a Boolean. You can tell that a piece of information is Boolean if it can have one of only two possible values, such as yes/no, or on/off, or true/false. Generally, code is more readable if expressions can be read like natural language. The same can be applied to make code easier to read.

Example

Change:

```
int checkFifoStatus(void);  
if (checkFifoStatus == OK) {  
    // do something  
}
```

To:

```
bool isFifoReady(void);  
if (isFifoReady()) {  
    // do something  
}
```

2.4.2. Use the verbs “set” and “get” for functions that set and get. Use the form “verbNoun” for function names. (Recommended)

Rationale

It’s our style. Note that a function that begins with “get” implies to some readers that it does nothing more than return the current value of something, without changing any values or affecting the state of any objects in any way. Member functions with names that begin with “get” are usually `const` functions.

Example

```
class myClass {  
public:  
    int getChannelNumber(void) const;  
    void setChannelNumber(int chan);  
    // etc. . . .  
}
```

In general, the morphology of a function name should be “verb-noun” instead of “noun-verb.”

Chapter 3

Types

“Abstraction is selective ignorance.”

-- Andrew Koenig and Barbara Moo, [Koenig2000, p. 80]

3.1 Choose the correct information abstraction

3.1.1. *Choose a type that tells you something about the information it holds. (Required)*

Rationale

For trivial loop counters or iterators, the type is not so important. For example, knowing that the array `X` has only two elements and can never have more, it is sufficient to loop through the two-element array with an index of type `int`. Type `int` is the natural size of the machine architecture, and often implies, “this variable is trivial, its usage is well-defined, well-confined, and there is no need to make a point by giving it a type that draws attention to itself.” For values that can only be nonnegative and never used in math expressions, consider using `unsigned int`. It is the natural machine word size in an unsigned flavor. For example, if we have an array known to be very small, and especially if the array index is limited to loop scope, use `size_t` or `unsigned int` as a generic index type that does not draw attention to itself:

```
for (size_t i = 0; i <= 1; i++) {  
    // do something with array xyz[i];  
}
```

Contrast the example above with a loop that uses an uncommon type for the index, such as this:

```
for (UINT16 i = 0; i <= 1; i++) {  
    // do something with array xyz[i]  
}
```

The use of `UINT16` makes us stop and wonder why the original programmer felt it was necessary to use a user-defined type for the index, especially a data type that is specifically 16 bits wide.

For non-trivial variables it is often useful to choose a data type that will help specify the nature of the information that it carries. For example, if a variable specifies time duration, and if the maximum value could be greater than $2^{16}-1$ but not greater than

$2^{32}-1$, and if the value could never be negative, then this information can be conveyed by giving it a type of `unsigned long`.

If a kind of information could be used in math expressions, you may want to give it a signed type so that negative values don't wrap around to large positive values.

Be consistent. When the same kind of information is used in different parts of code, always use the same data type to represent it. In the example above, if that particular kind of time duration information is used in many places, then consistently use type `unsigned long` to hold its values.

Often it is useful to define a special name for a particular kind of data. By abstracting the data type you realize at least three benefits: (1) the user-defined name helps document the type of data you're handling, (2) it encourages you not to mix values of different abstractions, and (3) if the range of values ever becomes different from the data type you're using, you can tweak the data type in the one place where it is defined by the `typedef`. If a `typedef` is used to abstract a data type, be consistent in using the `typedef'd` name.

Reference

[Koenig2000], especially pp. 18, 22.

3.2 Specific-width types

3.2.1. *Only use a specific-width data type when needed.* (Required)

Rationale

In a system where data is shared among different processors or between code compiled by different compilers, it is necessary to write and read data in a portable way. Bits have to be the same order, bytes have to be ordered in the same endianness, and the number of bits in a variable must be the same. There are other requirements for portable data, but this rule concerns the data width. Use user-defined data types to specify if the data must be exactly 8, 16, 32, or 64 bits wide. When it is necessary to store data in an exact number of bits, then use the user-defined types such as `uint16_t` or `UINT16`. If it is not necessary to store the data in a specific number of bits, then use a built-in data type that has a range sufficient for the information. For C and C++, the guaranteed minimum ranges for integer types are as follows (the ranges are inclusive):

<code>char</code>	0 through 127, or -127 through 127
<code>signed char</code>	-127 through 127
<code>unsigned char</code>	0 through 127
<code>short, signed short</code>	-32767 through 32767
<code>unsigned short</code>	0 through 65535
<code>int, signed int</code>	-32767 through 32767
<code>unsigned, unsigned int</code>	0 through 32767

long, signed long	-2147483647 through 2147483647
unsigned long	0 through 4294967295

Example

```
// Resource numbers are small positive integers
const char *convertResourceNumberToString(UINT16 resourceNum);
```

In this example, there is probably no reason for the function to require that the argument be exactly 16 bits wide. Yet, the specific-width type makes us wonder what tricky code is inside the function that makes it require that the argument to be precisely 16 bits wide. Wouldn't a simple `int` or `unsigned int` have worked? The special type draws attention to itself. It tells the world that the value passed to it must be 16 bits wide, not 8, not 32. Either of the following alternatives are better:

```
// This draws attention to the fact that resource numbers
// are non-negative and not used in math expressions, otherwise
// they are unremarkable:

const char *convertResourceNumberToString(unsigned int resourceNum);

// The next function uses a user-defined type to abstract the
// type of a "resource number," hiding its actual machine
// representation.

const char *convertResourceNumberToString(ResourceNum_t resourceNum);
```

3.3 Hard-coded special values

3.3.1. *Don't hard-code constants and magic values.* (Recommended)

Rationale

You may know what 2 or 781 or 4.2 means in your code. But will your successor know? It is best to symbolically define the constant for two reasons: (1) constants are not always constant – some do change as products change, and (2) it helps document what the value means. Also see section 3.4 about encapsulating constants inside functions, and section 8.2 about using functions, function objects, and `const` variables instead of preprocessor macros.

Example

Change:

```
if (trackNum > 312) {
```

To:

```
const int lastReservedTrack = 312; // Tracks 0..312 are reserved
if (trackNum > lastReservedTrack) {
```

Whenever you need to specify the size of a variable or object, use the `sizeof` operator. Your code will usually be more likely to survive longer if you apply the `sizeof` operator to the name of the variable or object, not to its type, because the type of the object is more likely to change in the future than its name.

Example

Change:

```
int checksum(cmdMsg& msgBuf)
{
    long checksum = 0;
    for (size_t i = 0; i <= sizeof(cmdMsg); ++i) {
```

To:

```
    for (size_t i = 0; i <= sizeof msgBuf; ++i) {
```

(The argument to the `sizeof` operator must be enclosed in parentheses if it is the name of a type.)

The same principle applies to code that must know the size of a built-in data type.

Example

Change:

```
    for (int i = 0; i < 4; ++i)        // wrong: what is the 4 for?
```

To:

```
    for (int i = 0; i < sizeof bufMsg; ++i) // best
```

Don't use:

```
    for (int i = 0; i < sizeof(long); ++i) // embedded assumption
```

With a symbolic label or a `sizeof` expression, we can more easily see that the loop counts through the number of bytes in a particular data item. Although using `sizeof(long)` helps us understand the loop, it hard-codes the assumption that the data item we are examining will always be the size of a `long`. This violates the rule in section 5.3 about making assumptions about the width of data items. Instead, write the code so that it says exactly what you intend, *viz.*, loop a number of times equal to the number of bytes in a certain named object, whatever type it happens to be.

Reference

[Kernighan1999], [Lakos1996]

3.4 Constants

3.4.1. *Prefer late binding of “constants.” (Recommended)*

Rationale

The old school used preprocessor defines for constant arithmetic values or strings. However, if you look at the ways that legacy code had to change to keep up with new hardware or new functionality, you will find that many values that once were considered “constant” had to change over time, or had to take different values for different types of hardware, or for different versions of software. What we call “constants” are seldom absolutely constant. If you write

```
#define PI 3.141592653589793238462643383279502886
```

then you probably won’t have to change that value ever again. However, if you write,

```
#define CHANNELS_PER_BOARD 2
```

then what do you think the chances are that someone will eventually have to change that “constant” to accommodate the new capabilities of revised hardware? We regularly invest a lot of labor changing huge sections of code in which certain “constants” have changed.

Much of the pain of changing constants can be eliminated by using run-time evaluation of the “constant” value combined with dynamic allocation, rather than compile-time evaluation and static allocation. This technique is rarely followed, but easy to apply. It adapts easily to changing requirements, and is highly recommended.

Example

Change

```
#define CHANNELS_PER_BOARD 2
struct channelState chanState[CHANNELS_PER_BOARD];
```

To:

```
inline int channelsPerBoard(void)
{
    return 2;
}
struct channelState *pChanState;
. . .
pChanState = new channelState[channelsPerBoard()];
```

Note

In the example above, the inline function that returns a constant will generally cause the compiler to generate identical code as the example using the preprocessor. The advantage is that the so-called “constant” can change by changing the behavior of the `channelsPerBoard()` function. For example, suppose new hardware is invented that must coexist with the older boards, but they have different capabilities. Suppose we want the same binary executable to work with all possible hardware configurations. By making the number of channels per board a run-time expression, we can change the

function to test the board type and return a different “constant” for each board type, something that might have been impossible using preprocessor macro constants.

Reference

[Meyers1998]

3.4.2. *Don't implicitly embed constants in control structures.* (Required)

Rationale

A lot of code is rewritten to remove implicit assumptions of the value of constants. For a recent example in our department, consider how much code was rewritten to change the number of video formats from two (NTSC and PAL) to more than two. That was an extreme example, but a lot of our code contains control structures that assume a particular value for a “constant” that could possibly change.

This differs from the previous rule about hard-coded constants. This rule applies to control structures that are written with assumptions about the problem that they solve; there are no explicit constants involved. The control structure itself is a hard-coded manifestation of some assumed constant.

This rule can be generalized. Code can often be written in such a way that design constraints are not embedded in the structure of the code.

“Don't forget: Design constraints buried in a class's design are as bad as magic constants buried in code.”

-- [Alexandrescu2001], p. 19.

Example

There are many ways to eliminate the implicitly embedded constants from code. For example, suppose that there are two kinds of video formats - PAL and NTSC. Our first attempt might be like this:

```
if (videoFormat == PAL) {
    // do something with PAL
} else {
    // It must be NTSC.    (Wrong assumption!)
}
```

However, this code assumes that there are exactly two cases - if it is not PAL then it must be NTSC. This control structure implicitly embeds the assumption that there are only two cases and no more, thus limiting our ability to add new cases.

Often, a switch statement is a step in the right direction. The example code above can be rewritten like this:

```

switch (videoFormat) {
case PAL:
    // do something with PAL
    break;

case NTSC:
    // do something with NTSC
    break;

// There is room for new cases here in the future.

default:
    logErrorMessage(" Please debug me! ");
    // not implemented... yet.
}

```

The switch statement removes the assumption that the world is divided into only two parts. The switch statement also has the virtue of logging an error if a new case is ever introduced into the design without adding code to support it.

With just a tad more coding you can make robust, expandable code with lookup tables, inheritance, or parameterized objects. For example, you can write the preceding switch statement using a more general lookup table and factoring out the common code. It might look something like this:

```

typedef void * (*workerFunction)();
extern workerFunction funcDispatchTable[VideoFormatNumTypes];
. . .
workerFunction pFunc = funcDispatchTable[videoFormat];
pFunc();

```

Many variations are possible. The key is to identify what information might change in the future, then centralize that information and define it in one prominent place. If new capabilities are ever added to the design, all you have to do is change the lookup table, and all the code remains unchanged. The general design does not embed assumptions about the number of cases to handle, and thus it is expandable without changing the code.

Reference

[Alexandrescu2001], [Carroll1995], [McConnell1993], esp. 12.2.

3.5 Boolean types

3.5.1. *In C++, Boolean information should be type bool. (Required)*

Rationale

The C language was deficient in not having an intrinsic Boolean type. C++ fixed that deficiency. If a piece of information is by nature Boolean, it should be so indicated by being stored as a `bool` type, unless there are compelling reasons not to. If you see a C++ variable or function of type `bool`, you know exactly what values it can have, and you

know something about the nature of the variable or function. If there is a valid reason for not using this intrinsic type for Boolean values, then the reasons must be documented in the code.

Example

```
int checkFifoFull(void);
```

Here we don't know what the function will return. We have to assume it was declared an `int` type for a reason. That may mean that the function can return some range of integers that mean something, but we don't have a clue what the different values mean. We can take a guess that it might return the number of items in the FIFO. Without code comments, we would have to look at the implementation of the function to see what it returns. However, consider this declaration:

```
bool checkFifoFull(void);
```

Immediately we can make a pretty good guess that this function checks the FIFO to see if it is full. Most likely the function returns `true` if full, `false` if not. Since this is a predicate function, we can apply guideline 2.4 to give it a name that clearly shows that it is a predicate function:

```
bool isFifoFull(void);
```

Now, is there any question what the return value means?

There is no reason in C++ to use a user-defined type, such as `BOOL`, instead of the built-in type `bool`. Consider this function:

```
BOOL isFifoFull();
```

The function is named like a predicate, and we can assume it returns something that means yes or no, or true or false. But why didn't the programmer feel that the built-in type `bool` was not sufficient? That stops us and makes us wonder why the built-in type wasn't adequate for the programmer. If there are no comments to give us hints, we have to wonder why the programmer felt that the return type had to be the user-defined type `BOOL` instead of `bool`. Is it a matter of the width of the data object used to store the value? Does the function return values other than 0 and 1? It makes us examine the use of the return value and reverse-engineer the code in order to determine the special characteristics of the return value.

In C code we have traditionally used the user-defined type `BOOL`. It has been defined variously as `short`, `unsigned int`, or `int`. When writing C++ code, values of the built-in type `bool` will automatically convert to and from the user-defined type `BOOL`. Likewise, the values of the built-in values `true` and `false` will correctly and silently convert to and from the user-defined values `TRUE` and `FALSE` - no cast is needed. When writing C++ code we should use the facilities built into the C++ language when possible.

Also See:

Section 8.2 and 8.1.

3.6 Comparisons, compatibility, casts, and conversions

3.6.1. In C++ use `reinterpret_cast<>()` for non-portable casts. (Required)

Rationale

For portable, well-defined casts, use either the C-style cast, such as:

```
struct X *pX = (struct X *)pY;
```

or the C++ cast:

```
struct X *pX = static_cast<struct X *>(pY);
```

If you are not familiar with the C++ style cast operators, they look unwieldy at first. Once you are used to them, they help you by telling you something about what the programmer intended. For casts that are generally portable and well-behaved, the C++ `static_cast<>()` operator works fine.

If you want to perform a cast that is implementation-dependent, non-portable, and therefore by definition, dangerous or suspicious, you can document that you know the risks and your cast is intentional by using the C++ cast operator `reinterpret_cast<>()`.

Specifically, use `reinterpret_cast<>()` when you cast from a pointer to an integer, or an integer to a pointer, or when you cast between unrelated types, such as from a pointer to unsigned long to a pointer to struct, or if you want to avoid the normal rules that allow the compiler to apply user-defined conversions in a cast.

Avoid C-style casts for non-portable conversions. A C-style cast does a `static_cast<>()` if it knows how, otherwise it does a `reinterpret_cast<>()`. It's much better to know which you want to do, and do that.

Example:

Given:

```
void *p;  
unsigned long x;
```

Change:

```
x = (unsigned long)p;
```

To:

```
x = reinterpret_cast<unsigned long>(p);
```

Reference

[Meyers1998]

3.7 cv-consistency

3.7.1. Maintain *const* and *volatile-correctness*. (Required)

Rationale

Every pointer (C and C++) and every reference (C++ only) refers to an object that is either writable or read-only. Those are the only two choices. The “referent” is the thing that the pointer points to or the thing referred to by the reference. The declaration of the pointer or reference specifies whether the referent is writable or read-only. The declaration specifies one or the other. If the referent is qualified by the keyword “*const*”, then the referent is read-only. If the *const* qualifier is omitted, the referent is writable. This is clear and unambiguous. For example,

```
const int *p;
```

declares a pointer named “*p*” that points to a read-only integer. The declaration:

```
int *q;
```

declares a pointer named *q* that points to a writable integer.

The presence of the *const* qualifier is a clear specification that the data is read-only. The absence of the *const* qualifier is a clear specification that the data may be modified.

In general, C and C++ allow pointers or references to writable data to convert silently to pointers or references to read-only data, but not the other way. “Constness” can be added by an implicit conversion, but it takes an explicit cast to remove constness.

In general, if a function takes a pointer or reference to data that is *const*-qualified, that constitutes a contract with the caller, a guarantee that the function will not modify the referent data. Likewise, if a function takes a pointer or reference to data that is not *const*-qualified, that constitutes an agreement with the caller that gives permission for the function to modify the data.

Example

```
// This takes a pointer to writable data; the function may
// change the data:

void f(int errorCode *pErr);

// This takes a pointer to non-writable data; the function will
// not overwrite the data:

void f2(const int errorCode *pErr);
```

Example

```
// A caller passes a pointer to status to function f().
// Function f() does not specify that the argument is const.

f(&status);

// Error: the caller now assumes that status is still the same:
perror(status);
```

Reference

[Carroll1995], [Meyers1998]

3.7.2. In C++ use `const_cast<T>()` to cast off `const` or `volatile`. (Required)

Rationale

Any time you use a cast to get rid of a `const` or `volatile` qualifier, it is suspicious. Using the new style cast operator shows (1) that you intend to form a type cast that changes only the `const` and/or `volatile` qualifiers and not the base type; and (2) it helps bring to the attention of the reader that there is an unusual but intended cast here that removes a cv-qualifier.

Example

Given:

```
int *p1;
const int *p2;
```

Change:

```
p1 = (int *)p2;
```

To:

```
p1 = const_cast<int *>(p2);
```

Example

Given:

```
char *p1;
volatile char *p2;
```

Change:

```
p1 = (char *)p2;
```

To:

```
p1 = const_cast<char *>(p2);
```

Reference

[Meyers1998], [Sutter2000]