Idiomatic Expressions in C

By Adam Petersen <adam@adampetersen.se>

Patterns exist at all levels of scale. At their lowest level they are referred to as idioms, as they depend upon the implementation technology. This article will present a collection of such idioms for the C language.

The Evolution of Idioms

As a language evolves, certain efficient patterns arise. These patterns get used with such a high frequency and naturalness that they almost grow together with the language and generate *an idiomatic usage* for the practitioner of the language. The resulting idiomatic expressions often seem peculiar to a newcomer of the language. This holds equally true for both natural and computer languages; even a programmer not using the idioms has to know them in order to be efficient at reading other peoples' code.

The idioms at the lowest levels are useful in virtually every non-trivial C program. Unfortunately idioms at this level are seldom described in introductory programming books and more advanced literature already expects the reader to be familiar with the idioms. The intention of this article is to capture some of these common idiomatic expressions.

Idiom Description Form

Because of their relative simplicity, devoting an article using a full-blown pattern form to a single one of these idioms would be to complicate it. On the other hand, simply listing the idioms would overemphasize the solution aspect. Therefore, each idiom will be introduced with a code sketch that illustrates a certain problem context, where after the idiom is applied to the code in order to solve the problem. This before-after approach also identifies certain code constructs as candidates for refactoring towards the idiomatic construct.

The idioms presented here are of two, overlapping categories:

- *Idioms for robustness*: The idioms in this category arose to avoid common pitfalls in the C language. For example, the idiom INITIALIZE COMPOUND TYPES WITH {0} provides a tight and portable way to initialize structs and arrays.

- *Idioms for expressiveness*: Writing code that communicates its intent well goes hand in hand with robustness; making the code easier to understand simplifies maintenance and thereby contributes to the long-term robustness of the program. For example, the idiom ASSERTION CONTEXT makes assertions self descriptive.

SIZEOF TO VARIABLES

Problem Context

In the C language, generality is usually spelled **`void*`.** This is reflected in the standard library. For example, the functions for dynamic memory allocation (**`malloc`**, **`calloc`**, and **`realloc`**) return pointers to allocated storage as **`void*`** and do not know anything about the types we are allocating storage for. The client has to provide the required size information to the allocation functions. Here's an example with **`malloc`**:

```
HelloTelegram* telegram = malloc(sizeof(HelloTelegram));
```

Code like this is sneaky to maintain. As long as the **`telegram`** really stays as a **`HelloTelegram`** everything is fine. The sneakiness lies in the fact that the **`malloc`** usage above contains a subtle form of dependency; the size given must match the size of the type on the left side of the assignment. Consider a change in telegram type to a **`GoodByeTelegram`**. With the code above this means a change in two places, which is at least one change too much:

```
/* We must change the type on both sides of the assignment! */
GoodByeTelegram* telegram = malloc(sizeof(GoodByeTelegram));
```

A failure to update both places may have fatal consequences, potentially leaving the code with undefined behaviour.

Applying the Idiom

By following the idiom of SIZEOF TO VARIABLES the dependency is removed. The size follows the pointer type being assigned to and the change is limited to one place. The original example now reads:

```
HelloTelegram* telegram = malloc(sizeof *telegram);
```

But wait! Isn't the code above dereferencing an invalid pointer? No, and the reason that it works is that **`sizeof`** is an operator and not a function; **`sizeof`** doesn't evaluate its argument and the statement **`sizeof *telegram`** is computed at compile time. Better yet, if the type of **`telegram`** is changed, say to a **`GoodByeTelegram`**, the compiler automatically calculates the correct size to allocate for this new type.

As **`telegram`** is of pointer type, the unary operator * is applied to it. The idiom itself is of course not limited to pointers. To illustrate this, consider functions such as **`memset`** and **`memcpy`**. These functions achieve their genericity by using **`void*`** for the pointer arguments. Given only the **`void*`** pointer, there is of course no way for the functions to know the size of the storage to operate on. Exactly as with **`malloc`**, it is left to the client to provide the correct size.

```
uint32_t telegramSize = 0;
memcpy(&telegramSize, binaryDatastream, sizeof telegramSize);
```

With SIZEOF TO VARIABLES applied as above, the size information automatically matches the type given as first argument. For example, consider a change in representation of the telegram size from 32-bits to 16-bits; the **`memcpy`** will still be correct.

INITIALIZE COMPOUND TYPES WITH {0}

Problem Context

Virtually every coding standard bans uninitialized variables and that with very good reasons. Initializing a basic type such as **`int`** or **`char`** is straightforward, but what is the correct way of initializing a compound type like an array or struct? A dangerous but unfortunately common practice is shown below:

```
struct TemperatureNode {
   double todaysAverage;
   struct TemperatureNode* nextTemperature;
};

struct TemperatureNode node;

memset(&node, 0, sizeof node);
```

The problem is that **`memset`**, as its name indicates, sets the bits to the given value and it does so without *any* knowledge of the underlying type. In C, all bits zero do not necessarily represent floating-point zero or a **`NULL`** pointer constant. Initializations using **`memset`** as above result in undefined behaviour for such types.

The alternative of initializing the members of the struct one by one is both cumbersome and risky. Due to its subtle duplication with the declaration of the struct, this approach introduces a maintenance challenge as the initialization code has to be updated every time a member is added or removed.

Applying the Idiom

Luckily, the portable solution provided by the INITIALIZE COMPOUND TYPES WITH {0} does not only ensure correctness; it also requires less typing. The code below guarantees to initialize all members (including floating-points and pointers) of the structure to zero. The compiler guarantees to do so in a portable way by automatically initializing to the correct representation for the platform.

```
struct TemperatureNode node = {0};
```

At the expense of creating a zero-initialized structure, **`memcpy`** may be used to reset an array or members of a structure later. Because it works by copying whole bytes, possible padding included, **`memcpy`** does not suffer from the same problem as **`memset`** and may safely operate on the structure in our example.

```
const struct TemperatureNode zeroNode = {0};
struct TemperatureNode node = {0};

/* Perform some operations on the node. */
...

/* Reset the node (equal to node = zeroNode; ) */
memcpy(&node, &zeroNode, sizeof node);
```

Using **`memcpy`** for zeroing out a struct sure isn't the simplest possible way. After all the last line above could be rewritten as **`node = zeroNode`** while still preserving the same behaviour. Instead the strength of this idiom is brought out when applied to an array. It helps us avoid an explicit loop over all elements in the array as **`memcpy`** now does the hard and admittedly boring task of resetting the array.

```
const double zeroArray[NO_OF_TEMPERATURES] = {0};
double temperatures[NO_OF_TEMPERATURES] = {0};

/* Store some values in the temperatures array. */
...

/* Reset the array. */
memcpy(temperatures, zeroArray, sizeof temperatures);
```

ARRAY SIZE BY DIVISION

Problem Context

The C language itself does not provide much support for handling its built-in arrays. For example, when passing a certain array to a function, the array decays into a pointer to its first element. Without any specific convention and given only the pointer, it simply isn't possible to tell the size of the array. Most APIs leave this book-keeping task to the programmer. One example is the **`poll()`** function in the POSIX API. **`poll()`** is used as an event demultiplexer scanning handles for events. These handles, which refer to platform specific resources like sockets, are stored in an array and passed to **`poll()`**. The array is followed by an argument specifying the number of elements.

```
struct pollfd handles[NO_OF_HANDLES] = {0};

/* Fill the array with handles to poll, code omitted... */

result = poll(handles, NO_OF_HANDLES, INFTIM);
```

The problem with this approach is that there is nothing tying the constant NO_OF_HANDLES to the possible number of elements except the name. Good naming does matter, but it only matters to human readers of the code; the compiler couldn't care less.

Applying the Idiom

By calculating ARRAY SIZE BY DIVISION, we are guaranteed that the calculated size always matches the actual number of elements. The calculation below is done at compile time by the compiler itself.

```
struct pollfd handles[NO_OF_HANDLES] = {0};
const size_t noOfHandles = sizeof handles / sizeof handles[0];
```

This idiom builds upon taking the size of the complete array and dividing it with the size of one of its elements (**`sizeof handles[0]`**).

MAGIC NUMBERS AS VARIABLES

Problem Context

Experienced programmers avoid magic numbers. Magic numbers do not communicate the intent of the code very well and may confuse anyone trying to read or modify it. Traditionally, some numbers like 0, 1, 3.14 and 42 are considered less magic than others. Consider a variable that is initialized to 0. A reader immediately expects this to be a default initialization and the variable will probably be assigned another value later. Similarly, culturally acquainted people know that 42 is the answer to the ultimate question of life, universe, and indeed everything.

The problem with all these not-so-magic numbers is that they build upon assumptions and expectations. These may be broken. One example is **`struct tm`** , which is used to represent the components of a calendar time in standard C. For historical reasons and in grand violation of the principle of least astonishment, assigning 0 to its **`tm_year`** member does not represent year 0; **`tm_year`** holds the years since 1900.Sigh.

Pure magic numbers are of course even worse. Consider the code fragment below:

```
startTimer(10, 0);
```

Despite a good function name, it is far from clear what's going on. What's the resolution - is 10 a value in seconds or milliseconds? Is it a value at all or does it refer to some kind of id? And what about this zero as second parameter?

Applying the Idiom

A step towards self documenting code is to express MAGIC NUMBERS AS VARIABLES. By applying this idiom to the code construct above the questions asked do not even arise; the code is now perfectly clear about its intent.

```
const size_t timeoutInSeconds = 10;
const size_t doNotRescheduleTimer = 0;

startTimer(timeoutInSeconds, doNotRescheduleTimer);
```

Of course the whole approach may be taken even further. By writing

```
startTimer(tenSecondsTimeout, doNotRescheduleTimer);
```

the code gets even more clear. Or does it really? The problem is that to the compiler the variable **`tenSecondsTimeout`** is really just a name. There is no guarantee that it really holds the value 10 as an evil maintenance programmer my have changed the declaration to:

```
const size_t tenSecondsTimeout = 42; /* Original value was 10 */
```

Such things happen and now anyone debugging the program will be unpleasantly surprised about how long ten seconds really are. They may feel like, hmm, well, 42 actually.

An idiom cannot be blindly applied and MAGIC NUMBERS AS VARIABLES is no exception. My recommendation is to use it extensively but avoid coding any values or data types into the names of the variables. Values and types are just too likely to change over time and such code gets unnecessarily hard to maintain.

NAMED PARAMETERS

Problem Context

As we expressed MAGIC NUMBERS AS VARIABLES the code got easier to read. That is, as long as the names of the variables convey meaning and finding good names is hard. To illustrate this, let us return to the previous example where a timer was started and extend it to include the start of two timers. I am rather happy with the name **`timeoutInSeconds`** and would have a hard time finding a second name that helps me remember the purpose of the variable equally well. Instead of going down the dark path of naming by introducing communicative obstacles such as **`timeout1`** and **`timeout2`**, I try to reuse the existing variable by removing its const qualification and re-assign it for the second timer.

```
size_t timeoutInSeconds = 10;
const size_t doNotRescheduleTimer = 0;

notifyClosingDoor = startTimer(timeoutInSeconds, doNotRescheduleTimer);

timeoutInSeconds = 12;
closeDoor = startTimer(timeoutInSeconds, doNotRescheduleTimer);
```

This is a tiny example, yet it's obvious that the code doesn't read as well as before. The extra timer is part of the story, but there's more to it. By re-using the **`timeoutInSeconds`** variable, a reader of the code has to follow the switch of value. As the right-hand sides of the expressions starting the timers look identical, the reader has to span multiple lines in order to get the whole picture.

Applying the Idiom

By naming parameters, it is possible to bind a certain value to a name at the immediate call site. C doesn't have language support for this construct, but it is possible to partly emulate NAMED PARAMETERS.

```
size_t timeoutInSeconds = 0;
const size_t doNotRescheduleTimer = 0;

notifyClosingDoor = startTimer(timeoutInSeconds = 10, doNotRescheduleTimer);

closeDoor = startTimer(timeoutInSeconds = 12, doNotRescheduleTimer);
```

The **`timeoutInSeconds`** variable is still re-used to start the timers, but this time directly in the function call. A reader of the code is freed from having to remember which value the variable currently has, because the expression now reads perfectly from left to right.

As neat as this idiom may seem, I hesitated to include NAMED PARAMETERS in this article. The first time I saw the idiom was in a Java program. It felt rather exciting as I realized it would be possible in C as well. Not only would it make my programs self-documenting; it would also bring me friends, money, and fame. All at once. After the initial excitement had decreased, I looked for examples and possible uses of the idiom. I soon realised that the thing is, it *looks* like a good solution. However, most often it's just deodorant covering a code smell. Most of the examples of its applicability that I could come up with would be better solved by redesigning the code (surprisingly often by making a function more cohesive or simply renaming it). All this suggests that NAMED PARAMETERS are more of a cool trick than a true idiomatic expression.

That said, I still believe NAMED PARAMETERS have a use. There are cases where a redesign isn't possible (third-party code, published API's, etc). As I believe these cases are rare, my recommendation is to rethink the code first and use NAMED PARAMETERS as a last resort. Of course, comments could be used to try to achieve the same.

```
closeDoor = startTimer(/* Timeout in seconds */ 12, /* Do not reschedule */ 0);
```

Because I believe that such a style breaks the flow of the code by obstructing its structure, I would recommend against it.

ASSERTION CONTEXT

Problem Context

Assertions are a powerful programming tool that must not be confused with error handling. Instead they should primarily be used to state something that due to a surrounding context is known to be true. I use `assert` this way to protect the code I write from its worst enemy, the maintenance programmer, which very often turns out to be, well, exactly: myself.

Validating function arguments is simply good programming practice and so is high cohesion. To simplify functions and increase their cohesion I often have them delegate to small, internal functions. When passing around pointers, I validate them once, pass them on and state the fact that I know they are valid with an assertion (please note that compilers prior to C99 take the macro argument to `assert` as an integral constant, which requires programmers to write `assert(NULL != myPointer);` for well-defined behaviour).

```
void sayHelloTo(const Address* aGoodFriend)
{
   if(aGoodFriend) {
      Telegram friendlyGreetings = {0};
      /* Add some kind words to the telegram, omitted here... */
      sendTelegram(&friendlyGreetings, aGoodFriend);
   }
   else {
      error("Empty address not allowed");
   }
}

static void sendTelegram(const Telegram* telegram, const Address* receiver)
{
   /* At this point we know that telegram points to a valid telegram... */
   assert(telegram);
   /* ...and the receiver points to a valid address. */
   assert(receiver);

   /* Code to process the telegram omitted... */
}
```

In the example above **`assert`** is used as a protective mechanism decorated with comments describing the rationale for the assertions. As always with comments, the best ones are the ones you don't have to write because the code is already crystal clear. Specifying the intent of the code is a step towards such clearness and **`assert`** itself proves to be an excellent construct for that.

Applying the Idiom

The idiom ASSERTION CONTEXT increases the expressiveness of the code by merging the comment, describing the assertion context, with the assertion it refers to. This context is added as a string, which always evaluates to true in the assertion expression. The single-line assertion is now self describing in that it communicates its own context:

```
assert(receiver && "Is validated by the caller");
```

Besides its primary purpose, communicating to a human reader of this code that the receiver is valid, ASSERTION CONTEXT also simplifies debugging. As an assertion fires, it writes information about the particular call to the standard error file. The exact format of this information is implementation-defined, but it will include the text of the argument. With carefully-chosen descriptive strings in the assertions it becomes possible, at least for the author of the code, to make a qualified guess about the failure without even look at the source. As it provides rapid feedback, I found this feature particularly useful during short, incremental development cycles driven by unit-tests.

A drawback of ASSERTION CONTEXT is the memory used for the strings. In small embedded applications it may have a noticeable impact on the size of the executable. However, it is important to notice that **`assert`** is pure debug functionality and do not affect a release build; compiling with NDEBUG defined will remove all assertions together with the context strings.

CONSTANTS TO THE LEFT

Problem Context

The C language has a rich flora of operators. The language is also very liberal about their usage. How liberating the slogan “Trust the programmer” may feel, it also means less protection against errors and some errors are more frequent than others. It wouldn’t be too wild a guess that virtually every C programmer in a moment of serious, head aching caffeine abstinence has erroneously written an assignment instead of a comparison.

```
int x = 0;

if(x = 0) {
   /* This will never be true! */
}
```

So, why not simply bane assignments in comparisons? Well, even if I personally avoid it, assignments in comparisons may sometimes actually make some sense.

```
Friend* aGoodFriend = NULL;
...

if(aGoodFriend = findFriendLivingAt(address)) {
   sayHelloTo(aGoodFriend);
}
else {
   printf(“I am not your friend”);
}
```

How is the compiler supposed to differentiate between the first, erroneous case and the second, correct case?

Applying the Idiom

By keeping CONSTANTS TO THE LEFT in comparisons the compiler will catch an erroneous assignment. A statement such as:

```
if(0 = x) {
}
```

is simply not valid C and the compiler is forced to issue a diagnostic. After correcting the if-statement, the code using this idiom looks like:

```
if(0 == x) {
   /* We’ll get here if x is zero – correct! */
}
```

The idiom works equally well in assertions involving pointers.

```
assert(NULL == myNullPointer);
```

Despite its obvious advantage, the CONSTANTS TO THE LEFT idiom is not completely agreed upon. Many experienced programmers argue that it makes the code harder to read and that the compiler will warn for potentially erroneous assignments anyway. There is certainly some truth to this. I would just like to add that it is important to notice that a compiler is *not* required to warn for such usage. And as far as readability concerns, this idiom has been used for years and a C programmer has to know it anyway in order to understand code written by others.

Summary

The collection of idiomatic expressions in this article is by no means complete. There are many more idioms in C and each one of them is solving a different problem.

Idiomatic expressions at this level are really just above language constructs. Thus, the learning curve is flat. Changing coding style towards using them is a small step with, I believe, immediate benefits.

Acknowledgements

Many thanks to Drago Krznaric and André Saitzkoff for their feedback.