The Roof Is Going to Go

by Chuck Allison

GETTY IMAGES

In October 1993, the Associated Press published an article about a German tourist on a US flight who approached the cockpit while the plane was taking off. When a flight attendant asked him to sit down, he said, "No, no, the roof would go!" Naturally, the flight attendant took that as a bomb threat, so the plane landed and the man landed—in jail. It turns out he just needed to use the toilet. According to the article, the phrase "Then the roof goes" is a German idiom for "having to go." It's too bad he spent nine months in jail.

Moral of the story: Idioms don't translate.

It's difficult to learn to speak like a native in a new language, but it is doable and desirable. As programmers, we should use programming languages as they were intended, including understanding and using their idioms.

Speak Like a Native

Back in olden times, FORTRAN did not have an **else** statement or a compound statement construct. To implement an **if-then-else** construct, you had to use GOTOs (see figure 1).

```
      IF (X .LE. 4) GOTO 10
      Y = F(X)
      GOTO 20
10    Y = G(X)
20    CONTINUE
```

Figure 1

That's just how it was done. It was a FORTRAN idiom.

My second language was Algol. You'll never guess how I first rendered the fragment in figure 1 in that language (see figure 2).

```
      if x <= 4 then goto ten;
      y := f(x);
      goto twenty;
ten:  y := g(x);
twenty: …
```

Figure 2

Hey, I'm a math major—what did you expect? Not to worry; I soon saw the light. That is, I learned the Algol way (see figure 3).

Most programming languages have their individual claims to fame, which dictate when to use them and what common idioms apply. My first major project out of school was a flight planning system for the US Air Force. We used COBOL for the I/O and FORTRAN for the numeric processing, because that's where those respective languages excelled.

```
if x > 4 then
    y := f(x)
else
    y := g(x)
```

Figure 3

C and C++ are primarily systems programming languages; therefore, power and efficiency are top priority. The C mantra has always been, "Trust the programmer." It's true that not all programmers deserve that trust—programming outside of a sandbox requires some maturity. Many, for example, find idiomatic C code like that in figure 4 troublesome.

```
void f(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Figure 4

Using the expression `*s++` is a well-entrenched idiom, but for many programmers the condition within the `while` statement should be a comparison rather than an assignment statement. This function is a simplified version of the standard `strpy` function, taken right from Kernighan and Ritchie's *The C Programming Language*. Here's what the authors say about it:

> Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom

should be mastered, because you will see it frequently in C programs.

One of the guiding principles of C++ is, "You don't pay for what you don't use." That's why there is no "mother of all classes" in C++ as there is in other object-oriented languages. Where you might use `java.lang.Object` in Java, you would more likely use templates in C++.

```
template<class T> class Singleton {
  Singleton(const Singleton&);
  Singleton& operator=(const Singleton&);
protected:
  Singleton() {}
  virtual ~Singleton() {}
public:
  static T& instance() {
    static T theInstance;
    return theInstance;
  }
};
```

Figure 5

A common template idiom in C++ is known as CRTP, which stands for "Curiously Recurring Template Pattern." The code in figure 5 defines a base class that will make any of its derived classes a singleton. Deriving from `Singleton<T>` requires CRTP.

The copy constructor and assignment operator are private and unimplemented, so no objects can inadvertently come to life by copy or assignment. The default constructor is protected, so only derived objects can invoke it. The type of the object to be made a singleton is the template parameter `T`. This means that `T` must be a template argument to `Singleton` at its own (i.e., `T`'s) point of definition, as the code in figure 6 illustrates.

```
// A sample class to be made a Singleton
class MyClass : public Singleton<MyClass> {
  int x; //represents the set of all attributes
protected:
  friend class Singleton<MyClass>;
  MyClass() { x = 0; }
public:
  void setValue(int n) { x = n; }
  int getValue() const { return x; }
};
```

Figure 6

The first line may seem odd—defining a class in terms of itself—but this is what is meant by CRTP. This works because the `Singleton` template does not need to know the internals of `T` (its only use of a `T` object is the local static object in `instance()`—since it's static, it's not part of a `Singleton<T>` object, which has size zero). With the `Singleton` template you can make any derived class a singleton by making its constructor non-public and its own instantiation of `Singleton` a friend, so `Singleton::instance` can instantiate it. This particular idiom requires the derived class to have a default constructor.

```
class Singleton(object):
    __singletons = {}
    def __new__(cls, *args, **kwds):
        if not cls.__singletons.has_key(cls):
            cls.__singletons[cls] = object.__new__(cls)
        return cls.__singletons[cls]
```

Figure 7

Python uses a different idiom for a singleton maker. Python has a universal `object` class, and, as in Java, each class in Python has an associated *class object*. When you create an object in Python, the special method `object.__new__()` uses the corresponding class object to build your object, and then calls your constructor (named `__init__`), if you have defined one. You can define your own `__new__` method for any class, which is the key for a `Singleton` superclass (see figure 7).

```
class Foo(Singleton):
    def __init__(self, x):
        self.x = x

print Foo(10) is Foo(20) # True: same object!
```

Figure 8

This `Singleton` class has a (static) dictionary (aka hash table), `__singletons`, which stores a unique object for each class that inherits from `Singleton`. Whenever an object of any subclass is created, `__Singleton__new__` executes. It first checks to see if the class object (`cls`) for the instance to be created is already in `__singletons`. If so, it returns the existing unique object for that class. Otherwise, it creates a new class object/object pairing and stores it in `__singletons`, calling `__object.__new__` to allocate storage. Using it is way cinchy (see figure 8).

Know Your Library

Speaking of COBOL, I once worked at a large organization that was trying to convert its COBOL programmers to C programmers. Talk about verbose code. I took one of their modules

```
ifstream inf("pulse.dat");
vector<int> nums;

copy(istream_iterator<int>(inf), istream_iterator<int>(), back_inserter(nums));
transform(nums.begin(), nums.end(), nums.begin(), bind2nd(minus<int>(), 12));
cout << "sum = " << accumulate(nums.begin(), nums.end(), 0) << endl;
cout << "height = " << *max_element(nums.begin(), nums.end()) << endl;
copy(nums.begin(), nums.end(), ostream_iterator<int>(cout, " "));
```

Figure 9

and rewrote it in idiomatic C—and it came out one-seventh its original size. A key problem was that the COBOL programmers had no notion of a library—to them, everything was just verbs and data structures. Consequently, they reinvented parts of the standard C library over and over again. Not only is that a wasteful practice but it also is error prone, since the programmers weren't disposed to take the time to validate and optimize the little utilities they threw together on the fly. Library developers are specialists at making reliable, efficient, *reusable* code. Use standard libraries.

One incredibly brilliant library that (sadly) is underused is the standard C++ algorithms library. It contains more than fifty commonly needed functions that are generic, type-safe, efficient—and likely to save you a noticeable amount of time as well as code real estate.

As an example, I recently upgraded a client's analysis tool for digitized data. The original system was C-like C++ code full of loops. Evidently the developer, an able engineer, didn't know that it is possible to read in a stream of data, change it, and output it without a single explicit loop. Figure 9 shows how.

The first `copy` reads all the space-delimited integers from the file "pulse.dat" into the sequence `nums`. The call to `transform` subtracts twelve from each element. The `accumulate` algorithm computes the sum of the numbers, and `max_element` returns an iterator (pointer) to the largest number. The final call to `copy` prints all of the numbers to the standard output stream. If you're not familiar with the C++ algorithms library, this may look strange, but this is part of speaking C++ as a native—being succinct and declarative.

Expending the effort to master idioms is good practice. You won't be a native speaker until you do. **{end}**

Chuck Allison developed software for twenty years before becoming a professor of computer science at Utah Valley State College. He was senior editor of the C/C++ Users Journal *and is founding editor of* The C++ Source. *He is also the author of two C++ books and gives onsite training in C++, Python, and Design Patterns.*

It's interesting to consider what has changed and what has stayed the same as programming languages have evolved over the past half century. What features are the most lasting? Which idioms make certain languages your "favorites?" Do you have any new idioms in mind?

Follow the link on the StickyMinds.com homepage to join the conversation.