be used to construct a new element directly in the container's storage, thereby avoiding unnecessary copies or even moves:

```
void g(std::vector<std::string>& v)
{
    v.emplace_back("hello world");
        // invokes only the std::string::string(const char*) constructor
}
```

Calling `std::vector<std::string>::emplace_back` with a **`const char*`** argument results in a new `std::string` object being created in-place in the next empty spot of the vector's storage. Internally, `std::allocator_traits::construct` is invoked, which typically employs **placement `new`** to construct the object in raw dynamically allocated memory. As previously mentioned, `emplace_back` makes use of both variadic templates and forwarding references; it accepts any number of forwarding references and internally *perfectly forwards* them to the constructor of `T` via `std::forward`:

```
template <typename T>
template <typename... Args>
void std::vector<T>::emplace_back(Args&&... args)
{
    // ...
    (void) new (d_data_p[d_size]) T(std::forward<Args>(args)...);  // pseudocode
    // ...
}
```

Emplacement operations remove the need for copy or move operations when inserting elements into containers, potentially increasing the performance of a program and sometimes, depending on the container, even allowing even noncopyable or nonmovable objects to be stored in a container.

As previously mentioned, declaring without defining the *copy* or *move* constructor of a noncopyable or nonmovable type to be private is often a way to guarantee that a C++11/14 compiler constructs an object in place. Containers that might need to move elements around for other operations, such as `std::vector` or `std::deque`, will still need movable elements, while node-based containers that never move the elements themselves after initial construction, such as `std::list` or `std::map`, can use emplace along with noncopyable or nonmovable objects.

Decomposing complex expressions

Many modern C++ libraries have adopted a more functional style of programming, chaining the output of one function as the arguments of another function to produce complex expressions that accomplish a great deal in relatively concise fashion. Consider a function that reads a file, does some spell-checking for every unique word in the file, and gives us a