INSERT ELEMENT: A POSSIBLE PERFORMANCE ISSUE

- If we have a container, like stack of queue, it seems logical that when we add a new element via an insertion function, the type of the element remains the same. However, this is not always true. Consider, for example, this code:

```
Queue<std::string> lQueue;

lQueue.enqueue( "Hello World!" );
```

- Here, the container lQueue holds elements of type std::string, but we have used a string literal, "Hello World!", which is of type **const char**[13].

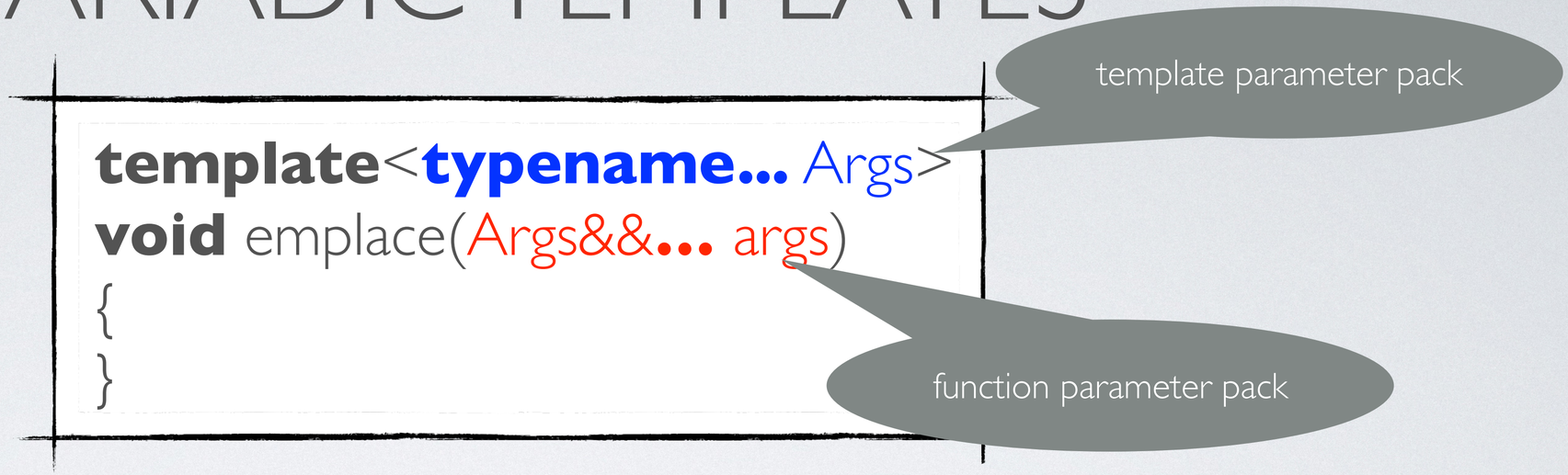
- The compiler sees a mismatch that needs to be resolved. In this case, an implicit type conversion occurs via std::string("Hello World!"). In the context of the enqueue call, the compiler creates a temporary of type std::string and passes it as constant reference to enqueue to be copied into the queue. With respect to performance, it would be better to pass the string literal directly to the queue and construct a std::string object in-place inside the queue.

EMPLACE ELEMENTS

```
template<typename... Args>
void emplace(Args&&... args)                                    // allow for type deduction of arg types
{
    assert( fHead != (fTail + 1) % N );

    fElements[fTail].~T();                                      // free old entry using T's destructor

    new ( &fElements[fTail++]) T( std::forward<Args>(args)... );   // placement new (matching constructor)

    if ( fTail == N )
    {
        fTail = 0;
    }
}
```

- The method emplace() works like enqueue, but the new element inserted is construct into the queue, not copied.
- We define emplace() as a variadic template method and construct the element in-place via a matching element constructor via perfect forwarding of the arguments received by emplace().

VARIADIC TEMPLATES



- A variadic template is a template function or class that can take a varying number of parameters. The varying parameters are known as a parameter pack.

- In a template parameter list, **class...** or **typename...** indicates that the following parameter represents a list of zero or more types (e.g., Args is a list of types).

- The name of a type followed by an ellipsis represents a list of zero or more non-type parameters of the given type (e.g., args is a list of non-type parameters).

STD::FORWARD()

```
void f( const int& p ) { std:: cout << "[l-value]" }
void f ( int&& p) { std:: cout << "[r-value]" }

template<typename T> void f3(T&& p)
{
    f( p );
    f( std::forward<T>( p ) );
    std::cout << std::endl;
}

int a = 2;
f3( a ) ;          // [l-value][l-value]
f3( 4 );           // [l-value][r-value]
```

- The helper function std::forward(arg) allows perfect forwarding on arg.
- No executable code is generated. It is a compile-time function.
- It returns an R-value reference to arg, if arg in not an L-value. If arg is an L-value reference, it returns arg unchanged.

EMPLACE SEQUENCE

```
fElements[fTail].~T();                                          // free old entry using T's destructor

new ( &fElements[fTail++]) T( std::forward<Args>(args)... );    // placement new (matching constructor)
```

- We when use emplace, we first need to free the object that we are overriding. We do not free the object itself, just the resources it uses.

- Next, we use the new in-place operator to construct the new object in-place. The new in-place operator does not acquire new memory. It simply reinitializes existing memory with new data.