The C++ STL: How to Harness Its Power

James Waugh
Software Engineer III

Sometimes, as a software engineer, one has to deal with code we would rather not. "What were they thinking?" is a common sentiment. In my experience, this seems to be more exacerbated in C++, where it's much easier to do the wrong thing and re-implement something ourselves instead of using what's available to us by the standard library, especially the Standard Template Library (STL).

In C++ in particular, it's very easy to fall into the mindset that the STL "has nothing really useful" or "can't do what I want." But in reality, for a number of common, real-world algorithms and expressions, it's a great toolset. In this blog post, I want to give a quick refresher on the STL, and show how proper STL usage would have greatly simplified now-legacy code at Accusoft.

Although I admit that C++ STL isn't as "rich" as other standard libraries such as C#'s, that's no reason to write it off. In some ways, it even provides more power, like std::adjacent_find. I also think the STL is a more unique in that it isn't as intuitive to understand its power. I believe this is because the STL algorithms tend to be very generic and don't always lend themselves to instantly knowing when to use what. So how does one go about effective STL use? I've found it comes down to a couple key sequential thought processes:

1. Become familiar with <algorithm>, <numeric>, and the STL containers in particular. This will allow one to realize the majority of what is available before programming.

2. Before writing code using containers, model the problem down to its core. This is through verbiage such as "all this function is doing is..." and "I just need to check that..." you may find there is either a specific algorithm for what you're doing, such as **std::nth_element** and the

set functions such as **std::set_union**, or the case is commonly handled with things such as **std::all_of**.

3. When writing, avoid raw loops where possible. C++ programmers may be familiar with the term "raw pointers" and to prefer unique_ptr, shared_ptr, and so on, but "raw loops" is a concept that, if you write a C for- or while-loop, there is likely a chance a STL algorithms can do the same for you with less code and less chance to make mistakes.

Not using raw loops has the advantage that STL functions can only do one thing. In many codebases, it seems the temptation to just modify an existing loop to check multiple things is common. If another loop to check something is simply a linear operation, unless you are doing super critical high performance applications, I believe it's worth it to maintain better readability.

The next sections will put these thought processes to work on a number of real code examples with steps on how to improve their effectiveness using the Standard Template Library. Read on!

Example: Very Common Range Element Checks

Here's one thing I see very often: Determining if a range is satisfactory based on properties of its elements. That is, making decisions if one of, any of, or none of the elements supply a certain criteria. For example, consider the following three very similar-looking functions:

```
bool is_valid(int* array,
int size)
{
    for ( int i = 0; i  <
size :  ++i)  {
   if (array[i]  != 5)
return false;
    }
    return true;
}
```

```
bool is_valid(int* array,
int size)
{
    for ( int i = 0; i  <
size ;  ++i)  {
   if (array[i] == 5)
return false;
    }
    return true;
}
```

```
bool is_valid(int* array,
int size)
{
    for ( int i = 0; i  <
size :  ++i)  {
   if (array[i] == 5)
return true;
    }
    return false;
}
```

In a number of codebases I've worked on, a generic name like is_valid is often used. These three functions do very different things. Can you tell which does what?

Modeling the Problem

The first "passes" if all elements are equal to 5
The second if no elements are equal to 5
The third if any are equal to 5

The problem statement is simply that a certain criteria must be matched in elements in a certain way. These are very simple operations, and it's easy to just write them while in the thick of thin things. But it doesn't help readability.

Improvement

These types of queries are so common, that one of my favorite STL additions in C++11 were

std::all_of, std::none_of, and std::any_of do exactly this. It's worth knowing these functions given their frequency, and similar-lookingness when written.

Here's the first function rewritten:

```
bool is_valid(int* array, int size)
{
    return std::all_of(array, array + size, isEqualTo5);
}
```

Example: Copying Text Objects

Here's a verbatim example from a codebase here I've worked on. What does this function do?

```
// Manually check all text blocks for being empty
for (RecognizeObjectIt roIt2 = pRegion->RecognizeObjectsBegin();
    roIt2 != pRegion->RecognizeObjectsEnd(); roIt2++) {
  RecognizeObject*  pRObject2 = *roIt2;

  if (pRObject2->RecognizeType() == TextBlockRecognizeObject) {
    TextBlock*  pTextBlock2 = (TextBlock*)pRObject2;

    if (pTextBlock2->IsEmpty()) {
   // Check if already in the list
   bool found = false;
   for (RecognizeObjectIt removeIt = regionRemoveList.begin();
         removeIt != regionRemoveList.end(); removeIt++) {
           RecognizeObject*  pRemove = *removeIt;

           if (pRemove == pTextBlock2) {
             found = true;
             break;
           }
   }

   if (!found)
        regionRemoveList.push_back(pTextBlock2);
    }
  }
}
```

I don't really want to read this. We have double-nested loops, booleans, breaks, and other things.

Modeling the Problem

At its core, all this code is doing is copying objects satisfying a certain criteria from one container to another. The resulting container shouldn't have duplicate elements.

Improvement

We can already greatly improve this using a range-based for-loop, and **std::find**. Not using **std::find** in this case is exactly like using a manual loop instead of Array.Contains in C#--you wouldn't do that there, right? It only makes sense to use the library to do standard things.

This is also an opportunity to clean up a little bit by preferring exit-early checks over ever-expanding indentation:

```
// Manually check all text blocks for being empty
for (RecognizeObject* ro : pRegion->RecognizeObjects()) {
  if (ro->RecognizeType() != TextBlockRecognizeObject)
   continue;

  if (!ro->IsEmpty())
   continue;

  // Check if already in the list
  if (std::find ( regionRemoveList.begin(), regionRemoveList.end(), ro) ==
regionRemoveList.end()) {
   regionRemoveList.push_back(ro);
  }
}
```

But in fact, we can do better. "// Check if already in the list" is a hint that we don't want duplicates in the list. Placing our minds in terms of the STL, we can remember **std::set**, which by definition only contains unique elements according to a comparison operator. Since we're using RecognizeObject*s, this is only a pointer comparison and is taken care of for us.

The code is then simply reduced to a single call to **std::copy_if**.

```
std::set < RecognizeObject*> regionRemoveSet;

std::copy_if ( pRegion->RecognizeObjectsBegin(),
               pRegion ->RecognizeObjectsEnd(),
               std::inserter(regionRemoveSet, regionRemoveSet.end()),
               [](RecognizeObject* ro) {
                 return ro->RecognizeType() == TextBlockRecognizeObject  &&
ro->IsEmpty();
         });
```

Had the code been written this way from the start, which it very well could have, minus C++11 features, much pain (my pain) could have been saved investigating this area to solve a flaky test years later. It's all about modeling the problem correctly from the start.

Conclusion

That's about it for this article. All-in-all, I believe the STL is unfortunately under-utilized, where often realizing its potential results in much more readable and enjoyable code. Also, the "S" in STL is "Standard" for a reason, meaning that more programmers will be able to understand your code by using what the language already provides.

Although some of this post may seem like common sense, there's something to be said about the STL in particular in its effective use. I hope this post shared some…*pointers*.