Alternatives: A New Dimension In Software Quality

Murugan Sundararaj

Murugan_Sundararaj@McAfee.com

Abstract

There are just umpteen numbers of things that go into building ”quality” software. Be it following good design principles, running static code analysis, conducting peer code reviews, testing, etc. But we barely think about “Alternatives”.

Alternative? Yes, finding alternate software code - one that can improve the fundamental quality of the software. It is not necessarily about changing the software implementation language but looking outside the traditional quality radar. Think of adapting to latest programming standard of the existing programming language in which the software is implemented. Or if only a subset of heavy-duty software is used then think of replacing it with lighter ones.

We put more time and energy into staffing up QA team instead of investing in adapting to latest programming standards up front to let developers develop better code. It is time to shed some light on this new approach in building quality software.

Today your software might be of better quality. But improving or introducing new quality processes may alone not sustain the current software quality. The core of software is the programming language in which it is implemented. Replacing the age-old code with simple, efficient and safer new code will boost the quality of the software. It does not stop there. The newer code will reduce the “mean time to repair”.

The out dated programming standard might make a developer write staggering amount of code to achieve a given functionality. But if there are simpler alternatives available in the new programming standard it is good to go for it. Remember bulky codes are one of the major sources of bugs.

This paper explains the various steps involved in finding alternatives and how to successfully apply them to improve the overall quality. We will also look at the challenges involved in multi platform environment.

Biography

Murugan Sundararaj is a principal engineer at McAfee, currently working in the McAfee India center in Bangalore. Over the past eight years he has been developing enterprise software both for windows and non-windows operating systems.

Prior to working for McAfee, Murugan worked as a senior software engineer for Yahoo!

Murugan holds a Bachelor’s degree in Electrical and Electronics Engineering from Periyar University, Tamilnadu, India.

1. Introduction

This technical paper focuses on software written in C++ programming language. But most of the ideas and methods, presented here, can be applied to software written in any programming language.

There are a number of ways to improve the quality of the software. This paper attempts to focus on a few key things that will make a huge impact on the overall quality.

Adopt new programming standards and see if the new facilities (language features) can solve your existing problems in a simple, efficient and safer way.

Stop toiling with the home grown platform abstraction library. Try the standard libraries and you might appreciate integrity of the code.

Software, over a period of time, tends to carry unnecessary baggage. This will result in increased footprint and redundancy. Find ways to get rid of the unused or under used code.

If throughput of the software is vital to your business, then it is time to think about non-blocking programming. The quality of the software might improve when the blocking functional calls are safely replaced by asynchronous ones.

2. Adopting New Programming Standards

We code day in and day out. Find bugs. Fix them. Tweak the performance by replacing dumb algorithms with intelligent and efficient ones. We try to solve problems with limited tools on hand. Run into design issues and often solve them the hard way. Hardly try something outside the traditional methodologies including but not limited to increasing the window of various phases like design, implementation, test, and automation.

Did we ever think adopting new programming standards could improve the quality and maintainability of the software?

The C++ Standards Committee has released a new standard – C++11, which is approved by ISO (The International Standards Organization). The new standard has got lot of useful features to make the software cleaner, safer and faster. If your software is based on older standards (say C++ 2003 or C++98) it is time to think about adopting new standards. Agreed, there are challenges in adoption. Starting from upgrading your compiler to learning what is new in the latest standards and how to apply them. But this transition is vital to see how one can benefit from the new standards.

Even if you have achieved the functionalities provided by the new features in your own way, it is worth considering what the standard library has to offer. Remember standard library is better than handcrafted specialized code.

I have picked up select features, which I feel, is going to be pervasive in the coming days. At first glance, new techniques may look esoteric, but they are simple, efficient, and extremely useful.

Motivating Example -1

Everybody likes “clean” code. It enhances the maintainability as well. Replace lengthy programming statements with short yet understandable constructs.

For example, to store sum of two elements of vector of type ‘U’ and ‘V’ in a variable, say ‘sum’, and use ‘sum’ at some point later, we would write something like this

```
template <class U, class V>
void f(const vector<U>& v1, const vector<V>& v2) {
   vector<U>::const_iterator v1_iter = v1.begin();
   vector<V>::const_iterator v2_iter = v2.begin();
   // ...
   int sum = *v1_iter + *v2_iter; // possible loss of summation result!!
   // ...
   // use 'sum' here.
}
```

If U is of type integer (say 10) and V of type double (say 20.5), then "sum" is 30. The summation result is lost. The type of "sum" should be double to hold the correct result. In other words the type of "sum" should be the one that you get from adding U with V.

Changing the type to "unsigned long" will not work out if either one or both of the vectors contain negative numbers. "long double" may not be desired if none of the vectors contain irrational numbers.

One has to solve this problem the hard way today. This leads to hacks and workarounds.

The new standard has introduced "auto"(The C++ Standards Committee 2013) to solve this problem. "auto" deduces the type of a variable from its initializer. Now the code will look like

```
template <class U, class V>
void f(const vector<U>& v1, const vector<V>& v2) {
   vector<U>::const_iterator v1_iter = v1.begin();
   vector<V>::const_iterator v2_iter = v2.begin();
   // ...
   auto sum = *v1_iter + *v2_iter; // type of sum is what you get from adding U with V
   // ...
   // use 'sum' here.
}
```

As you can see, it has not only solved the problem but also made the code cleaner and understandable.

Note: Type deduction happens at compile time. So, summation might lead to overflow (because the actual value of *v1_iter and *v2_iter are known only at run time), which is not solved by auto.

Few more examples:

```
auto x = 10 + 20;           // x will be of type int
auto x = 10 + 20.5;         // x will be of type double
auto x = INT_MAX + INT_MAX; // overflow!! x will be of type int.
```

Motivating Example -2

We make all possible efforts to save the runtime overhead of a program. Today we use either "assert" macro or #error preprocessor directive for testing software assertions.

For example, if applications were intended to run only on 32-bit processor, we would write

```
assert(sizeof(long) >= 8);
```

But the assert macro tests assertions at runtime, which is far later than would be desired, and which implies a runtime performance cost. The #error preprocessor directive is processed too early to be of much use.

The new standard offers "static_assert" (The C++ Standards Committee 2013) to save the runtime performance cost. The expressions are guaranteed to evaluate at compile time.

```
static_assert(sizeof(long) >=8, "64 bit code generation not supported");
```

Library developers use assertions extensively in their program. Having something that will assert at the compile time will help in reducing the runtime cost.

Motivating Example -3

Creation of temporary objects is one of the pain points every developer is battling with. Temporary objects slow down a program. Temporary objects are optimized away by the compilers (the return value optimization, for example). But this is not always the case, and it can result in expensive object copies. Consider the following

```
template <class T> swap(T& a, T& b) {
   T tmp(a);    // now we have two copies of a
   a = b;       // now we have two copies of b
   b = tmp;     // now we have two copies of tmp (aka a)
}
```

But, we didn't want to have any copies of a or b, we just wanted to swap them. C++11 provides “move” (The C++ Standards Committee 2013) semantics to overcome this.

```
template <class T> swap(T& a, T& b) {
   T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

Move semantics allows us to avoid unnecessary copies when working with temporary objects that are about to evaporate, and whose resources can safely be taken from that temporary object and used by another.

These are some of the features added in the latest standards. There could be lot many which are not covered here but might be useful to your application. Please read the latest standards for more information. The key thing which needs to be stressed upon is, look out for newer standards and try to solve problems that are present due to lack of features. This thinking can be applied to any product, independent of the implementation programming language.

3. Platform Agnostic APIs (Application Programming Interface)

Different OS(Operating system) provide different APIs to acquire and use system resources (creation of threads, for example). Software running on multiple OS has to handle system resource and concurrency in different ways, each varying from other OS. But writing cross-platform software takes time. We will end up writing huge amount of code to build platform abstract library.

Maintaining and updating this library on addition of support to new platforms or on upgrade of target platform may be expensive. Resource allocation or any OS specific operation is outside the boundary of software’s business logic. We should try to keep the direct dependency on OS specific API as minimal as possible.

You could either see if the newer standards have got ways (features) to address this issue or use a third party library that provides cleaner abstraction.

The new C++ standards, C++11, for example, added a memory model and provided ways to help the programmers to create threads and achieve synchronization using facilities such as mutexes, condition variables, and atomics. It will make the application development easier as the OS specific operations are blended with the programming language. It will lead to better error handling and debugging. Programmers may now get rid of their platform abstraction library and focus on the solving the business problem.

Here is an example of using C++11 thread (The C++ Standards Committee 2013) to sum up the elements of a huge vector, in parallel.

```
#include <iostream>
#include <vector>
#include <thread>

using namespace std;

class Accumulator {
private:
    vector<double>::iterator begin_, end_;
    double *sum_;
public:
    Accumulator(vector<double>::iterator begin, vector<double>::iterator end, double *sum) :
begin_(begin), end_(end), sum_(sum) {}
    void operator() () {
        while(begin_ != end_) *sum_ += *begin_++;
    }
};

int main()
{
    double sum_1 = 0.0;
    double sum_2 = 0.0;
    double sum_3 = 0.0;
    vector<double> v;

    for(int radius = 0; radius < 1500000; ++radius)
        v.push_back(3.14*radius*radius);

    std::thread t1{Accumulator(v.begin(), v.begin() + 500000, &sum_1)};
    std::thread t2{Accumulator(v.begin() + 500000, v.begin() + 1000000, &sum_2)};
    std::thread t3{Accumulator(v.begin() + 1000000, v.begin() + 1500000, &sum_3)};

    t1.join();
    t2.join();
    t3.join();
    std::cout << "sum = " << sum_1 + sum_2 + sum_3 << '\n';
}
```

4. Heavy But Under Or Un-used Code

Software when started from the grounds up stays lean and focused on the solving problem at hand. But over a period of time it might tend to attract more functionalities and features than required. This extra baggage goes unnoticed as it usually appears in smaller scale in every software release.

How to identify and safely remove the unnecessary code?

Get the latest design document. Form a team with the people that have had worked on most part of the software. Trawl through the source code. Brainstorm among the team and check with the design document and find the chunks which are no longer needed.

What to do with the unnecessary code?

You can either remove the code or replace it with some lighter ones.

How to verify if the software is still safe to use? In other words, how to verify the sanity of the software, post the above mentioned operation?

It is very important to see that the software's business logic functions the same way before and after carrying out the operation. You can follow the steps (this list not complete but are necessary) detailed below for the same.

1. Test the software thoroughly. Developers should be executing all the unit test cases, including the modules that are not impacted. This is to ensure the integrity of the software still remains intact. If needed add new test cases or modify existing ones around the impacted module.

2. Make sure there are no memory leaks by running the software through static analysis tools. There are various static analysis tools (for example, Coverity and Fortify) which give in-depth analysis of the software ranging from reporting "uninitialized variable" usage, dead code to buffer overflow and memory leaks. It is better to address all the issues reported by the tool.

3. Involve the QA and get all the black box testing (Wikipedia 2013) done. QA must be maintaining a repository of all the test cases executed since the birth of the software, meaning all the test cases created and executed across all the releases of the software. This is usually called Regression testing (Wikipedia 2013)

4. There are few bugs which will surface only under certain conditions, say after few hours or days of run or under heavy load. It will be difficult to unearth these kinds of bugs either in unit testing or black box testing. The software has to be soak tested (Wikipedia 2013).

The initial exercise of finding and removing or replacing the unnecessary code is just a starting point. Spend good time in this phase and make sure your changes do not have bearings on the business logic. The entire operation is said done only after QA gives thumps up.

But to successfully complete this operation we require good amount of time and effort. It is where the software test automation (Wikipedia 2013) gives us a big helping hand. If you have enough automation built around testing the complete functionality of the software, you could save a lot of time and effort.

Here is a use case on how replacing the mammoth code base with lighter and "to the point" one benefits us.

1. How do you store the runtime data?

Most of the software have to deal with the data (could be either input data or runtime data) in one or other way. The way the runtime data is handled will go a long way in improving the software quality, especially if your software churns out a lot of data in the runtime.

Study the kind and amount of data used and produced by the product. Make a wise choice before settling for a particular procedure. For example, using a full-blown sql database is not suggested unless your software produce huge amount of runtime data that requires structured query language for retrieval. You could consider storing them in a flat file (encrypt if necessary).

When to write the data in memory into the disk also matters as the disk I/O are expensive. Writing the data as and when created results in frequent disk I/O but reduces the data loss when the software goes haywire (crash or hung state). Frequent disk I/O hampers the software speed. Delayed writing means minimal disk I/O and good runtime speed, but might increase the probability of data loss. Strike a tradeoff between them to improve the quality of the software.

2. Get rid of unnecessary and accidental complexities:

A real world example from author's software development cycle:

We were developing a task scheduler module in which we formulated a state machine. The state machine captures the state of scheduler and the various tasks held by the scheduler. We started with few states and finished our first sprint (we follow scrum methodology for software development). As the scheduler

evolved, more and more states were added. The resulting code became too big to understand, debug, and maintain. We ended up exceeding the runtime allocated and the footprint of the software. This is a classic example of introducing accidental complexity into the software.

Lesson learned: get rid of the accidental complexity before it is too late.

How?

The key here is the design. It is supposed to be modular and loosely coupled. Lack of which had led to accidental complexity. But achieving such a quality design at each and every level might be challenging and time consuming. The suggestion is to sense the moment a module gets clumsy and invest some time redesigning that particular module and/or the dependent modules. This is process of restructuring an existing body of code is commonly called Code Refactoring (Wikipedia 2013)

One way to go about in this particular use case is to segregate the state machine from the scheduler. The state machine knows what action to be taken for a particular state. Now all the state handling mechanism happens in one place, there is no code duplicity. Apparently, it will be lot more easier to clean up the states that are no longer valid or add new states as the scheduler evolves over the time.

5. Paradigm Shift: Blocking To Non-blocking

Producing responsive software is a challenge. A responsive software is one that let the users to continue submitting requests before the earlier requests were done.

One of the weakest links in the software is the blocking function call. It could be a blocking networking call, file system call, or individual I/O operations that take long time to complete. I/O operations can be extremely slow compared to the processing of data.

Implementation strategies such as thread-per-connection can degrade system performance, due to increased context switching, synchronization and data movement among CPUs. With asynchronous operations it is possible to avoid the cost of context switching by minimizing the number of operating system threads and only activating the logical threads of control that have events to process.

Many Operating systems offer a native asynchronous I/O API for developing high performance network application. These asynchronous mechanisms handle the blocking system calls without requiring programs to use concurrency models based on threads and explicit locking.

Long-duration operations are performed asynchronously by the implementation on behalf of the application. Consequently applications do not need to spawn many threads in order to increase concurrency. Consider using them and see the difference in performance.

The next time you create a socket think about making it a non-blocking one. Windows provide I/O completion port (IOCP) to achieve asynchrony. UNIX based system offer select, poll, kqueue, etc to achieve the same. Boost asio library (Kohlhoff Christopher, 2013) provides a good abstraction over windows and non-windows platform.

6. Conclusion

Adopting new standards may not be a panacea for all the software problems. But one can definitely improve the current quality of the software at various levels. The new standards might provide cleaner, safer, and faster ways to solve your problem. Use caution while adopting the latest standards.

Trawling through the thousands of lines of source code and finding the chunks which are no longer needed warrants careful examination of the source code. Run through all the business case scenarios and check the changes you made do not impact the flow. Revisit your unit test cases. Run static analysis tools to check if there are still dead codes out there in the product. Remember dead codes are revealed only by good unit test cases. Ideally a product should have all its functions covered by the unit test cases.

It is more difficult to develop applications using asynchronous mechanisms due to the separation in time and space between operation initiation and completion. Debugging asynchronous applications may also be harder due to the inverted flow of control. But the overall throughput of the software will rise sharply.

References

The C++ Standards Committee, 2013, http://www.open-std.org/JTC1/SC22/WG21/

Wikipedia, 2013, https://en.wikipedia.org/wiki/Test_automation

Wikipedia, 2013, http://en.wikipedia.org/wiki/Regression_testing

Wikipedia, 2013, https://en.wikipedia.org/wiki/Test_automation

Wikipedia, 2013, http://en.wikipedia.org/wiki/Code_refactoring

Kohlhoff Christopher, 2013, http://www.boost.org/doc/libs/1_54_0/doc/html/boost_asio.html