KDAB

Modernizing C++

Kevin Funk

March 18

Code modernization is an essential part of the software development discipline. If you neglect modernizing your code, your project could miss out on improvements that help readability and maintainability, and lose out on optimizations that improve performance.

Your ability to spot errors at compile-time may also suffer due to recent advancements in both the programming language specification and compiler.

These days, the ISO standardization group is releasing a new version roughly every three years and the C++ language standard is evolving and improving faster than ever. Yet a significant number of new and helpful language features still aren't used in the majority of software projects. It's possible that programmers overlook C++'s improvements because they aren't aware of the new features. Or they don't have management support to fix something that is not yet technically broken.

The truth is that modernization improvements do have immediate pay-offs – both in terms of development time and (ultimately) money.

Regardless of the positive impact on your software project and the company's bottom line, there's a reasonable fear that changes distributed across source code may introduce bugs into what was once a stable product. To mitigate the cost and risk associated with code modernizations, tools for diagnosing and refactoring the usual programming patterns have become more and more popular. This whitepaper looks at a number of techniques used by automation tools to transform commonly used coding patterns

Modern C++ language standards and newer compilers are continually advancing the state of the art to find more and more hidden problems in your code.

to a more modern version. These techniques provide a small sample – there are many code improvements that can be automated that your modernization effort may want to consider. You may also want to read KDAB's Modernizing Legacy Systems whitepaper for a detailed guide on how to evaluate, plan, and execute a full-scale modernization effort.

1) Avoiding programming mistakes

Besides generating an executable from your source code, the compiler is an indispensable guide that helps you write a program by ensuring your code makes sense, reporting errors, and warning about situations that are confusing or ambiguous. Compilers help find so many issues with code before they become run-time bugs that we developers often wish they could flag even more problems. Thankfully, modern C++ language standards and newer compilers are continually advancing the state of the art; by keeping up with the newest C++ compiler, you'll find more and more hidden problems in your code.

C++11's override

One of the least complicated features introduced in C++11 is the new override keyword. Have you ever tried overloading a virtual function in a derived class, only to discover your new function is never called when running the program? Using override makes the intended use of a method declaration explicit, providing more clarity of the author's intent for derived classes and overloading virtual functions – both to the human reader as well as the compiler.

Example:
```
struct Base {
    virtual void reimplementMe(int a)
       const {}
};
struct Derived : public Base  {
    // override base class method
    virtual void reimplementMe(int a)
       {}
};
```
In the *Derived* class example above we attempted to override the method *Base::reimplementMe()* but we accidentally introduced a mistake in it's method signature – we forgot the *const*. Thus, the signature of the two declarations differ, so the *Derived* method does not actually override the *Base* method and *Derived::reimplementMe()* will not be called at runtime. Compiling this code snippet under C++03 won't issue any warnings or errors about a potential mistake.

Nearly every seasoned C++ programmer has wasted hours of their life figuring out why an overridden method was never called at runtime only to discover that it's due to an inadvertently mismatched method signature. Another very common way to encounter this problem is if you change the signature of the base class virtual function and the derived classes re-implement that method – but you forget to adapt the overridden methods accordingly. Again, this will go unnoticed by the compiler if override isn't used.

Thankfully, with the introduction of the new C++ override keyword in C++11, you can indicate the belief that you're overriding a function and the compiler can double-check if your assumption is correct.

The introduction of the new C++ *override* keyword in C++11 can save you countless hours of debugging when applied consistently.

Improved solution:

```
struct Base {
    virtual void reimplementMe(int a)
        const {}
};
struct Derived : public Base  {
    // override base class method
    virtual void reimplementMe(int a)
        override {}
};
```

Now if we compile this snippet, we get:

```
% clang++ -std=c++11 test.cpp
test.cpp:6:18: error: 'reimplementMe'
marked 'override' but does not override any
member functions
    virtual void reimplementMe(int a)
override {}
                 ^
1 error generated.
```

With the *override* keyword the compiler ensures that the signature of the method marked with override matches one of the method signatures in the base class (or classes). If it doesn't, the compiler reports the oversight and aborts. Applying this consistently throughout your codebase can save countless hours of debugging.

C++11's range-based for

Another easy-to-use feature of C++11 is the new range-based *for*, which can replace the traditional *for* loop when iterating over a range of values. This feature provides a safer way to loop over all elements of a container since you don't have to deal with iterators or index variables and instead can work with ranges directly.

Example:

```
const int N = 5;
int arr[] = {1,2,3,4,5};

vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);

// safe conversion
for (int i = 0; i < N; ++i)
  cout << arr[i];

// reasonable conversion
for (vector<int>::iterator it = v.begin();
it != v.end(); ++it)
  cout << *it;

// reasonable conversion
for (int i = 0; i < v.size(); ++i)
  cout << v[i];
```

In this case, we can convert these loops to use a range-based *for*. Because it can be mentally taxing to determine exactly how to convert a loop and whether it will have any unintended side effects, the C++ linter tool Clang-Tidy provides automatic conversion of for into range-based for as one of its rules. The tool assigns different levels of confidence (risky > reasonable > safe) to each type of transformation. In this example, a for loop that calls *.end()* or *.size()* after each iteration will be transformed with reasonable confidence since the new version with a range-based for will only call these methods once during the initialization of the loop. In the admittedly unusual circumstance that *.end()* has side-effects, the semantics of the transformed loop will differ. Most likely though, this transformation will only improve performance since repeatedly calling these methods isn't necessary for processing the loop.

Tools that optimize code without firing up the profiler make sense to use on code without identified performance problems – because even the tiniest penalties add up.

Improved solution:

```
// safe conversion
for (auto & elem : arr)
  cout << elem;

// reasonable conversion
for (auto & elem : v)
  cout << elem;

// reasonable conversion
for (auto & elem : v)
  cout << elem;
```

Default member initializers

C++11 allows class members to be initialized at the point of declaration. This makes it a lot less likely to forget initializing class members in constructor definitions. This feature is explained in the following example.

Example:

```
struct A {
  A() : i(5), j(10.0) {}
  A(int i) : i(i), j(10.0) {}
  int i;
  double j;
};
```

Improved solution:

```
struct A {
  A() {}
  A(int i) : i(i) {}
  int i = 5;
  double j = 10.0;
};
```

The algorithm in this example converts a default constructor's member initializers into the new default member initializers in C++11, and member initializers that match the default are removed, reducing repeated code. Because it eliminates the need to explicitly initialize member variable values, this may even support *'= default'* for some simple constructors where the compiler supplies a default constructor.

2) Improving performance

Many programmers wait until they have serious performance problems before they try to make their code run faster. Let's face it: it's not easy optimizing for performance – locating and fixing poorly performing code can be a time-consuming and expensive operation. However, some tools can help optimize code without even firing up the profiler. They make sense to run even on code that doesn't have identified performance problems because even the tiniest penalties add up – many performance issues are death by a thousand cuts. Besides, no user has ever complained about their program running too fast or using too little battery.

Avoiding unnecessary copy initialization

This optimization finds local variable declarations that are initialized using the copy constructor of a non-trivially copy-able type, where a non-modifiable *const* reference would suffice.

Example:

```
const string& constReference();
void Function() {
    // The warning will suggest
    // making this a const reference.
    const string UnnecessaryCopy =
        constReference();
}
```

The Clang-Tidy script will suggest replacing the copy by a *const* reference if the variable is already *const* qualified or if it is only used as a constant in subsequent code.

Doing so avoids a full copy of the referenced instance, likely with a slight performance boost – depending on how often this construct is found in your code.

Minimize excessive padding in record structures – and its needless memory consumption – by following the order recommended by the tool.

Avoiding unnecessary copy initialization in range-based for

The following optimization example, while similar to the previous one, focuses on unnecessary copy initializations inside the initialization of the range-based *for* loops. If the loop variable is used by value and hence copied in each iteration but a *const* reference would work as well, the code is flagged.

Example:

```
for (const auto included_category :
included_categories) {
    if (category == included_category)
        return true;
}
```

Improved solution:

```
for (const auto& included_category :
included_categories) {
    if (category == included_category)
        return true;
}
```

Only loop variables that are expensive to copy (having a non-trivial copy constructor or destructor) will be replaced with a *const* reference as above.

Excessive padding in record structures

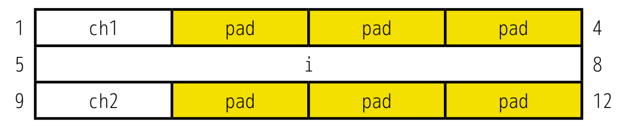
Every data type has a memory alignment that's mandated by the processor architecture. Aligning variables in memory allows the processor to fetch data in an efficient manner, improving performance. As an example, let's examine just two data types: *char* (with an alignment of 1 byte on 32- & 64 bit systems) and *int* (an alignment of 4 bytes on 32- & 64-bit systems). In other words, a *char* will use 1 byte, whereas an *int* will use 4 bytes.

So far so good ... but it becomes more complicated because the compiler tries to maintain proper alignment of data elements by inserting unused memory between elements within a *struct*, *class*, or *union*. This technique is known as padding. Of course, the compiler will be wasting memory unless your classes use the smallest amount of padding possible.

Example:

```
struct Record {
  char ch1;
  int i;
  char ch2;
};
```

The memory layout for *Record* looks like this:



Our structure has a total of 12 bytes of memory, with six wasted bytes of padding, which the Clang Static Analyzer will conveniently tell us:

```
% clang-5.0 -cc1 -analyze -analyzer-
checker=optin.performance -analyzer-config
optin.performance.Padding:AllowedPad=2
test.cpp
test.cpp:1:8: warning: Excessive padding
in 'struct Record' (6 padding bytes, where
2 is optimal). Optimal fields order: i,
ch1, ch2, consider reordering the fields or
adding explicit padding members
```

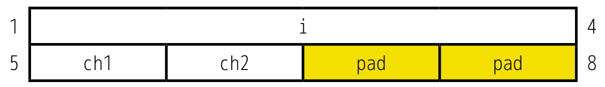
If you had a container filled with several million *Record* structs, you would waste six megabytes of memory with padding! You can minimize this needless memory consumption by following the order recommended by the tool (generally ordering the widest data types first).

Improved solution:

```
struct Record {
  int i;
  char ch1;
  char ch2;
};
```

Reading clean code without excess clutter is far easier than unstructured source with lots of extra information that's not needed.

Now, the memory layout for Record looks like this, saving four bytes at the end:



Analyzing this code snippet with Clang Static Analyzer again will no longer show any issues. We removed the excessive padding, thus shrinking the *Record* class by one third. Our memory caches will be happy!

3) Improving readability

Reading clean code without excess clutter is far easier than unstructured source with lots of extra information that's not needed. Less is more, at least for source code.

Simplifying boolean expressions

If your boolean expressions involve boolean constants, it's better to simplify them to use the appropriate boolean expression directly. These can result in some pretty straightforward replacements; a small sample of them represented below.

Expression	Simplified
`if (b == true)`	`if (b)`
`if (b == false)`	`if (!b)`
`if (b && true)`	`if (b)`
`if (true) t(); else f();`	`t();`

`if (e) return true;`
`else return false;`
	`return e;`

The less code you have to read while maintaining overall readability, the better.

Use empty() instead of size() > 0

Standard library containers have both a *size()* method (returning the number of elements in the container) as well as an *empty()* function for checking whether a container is empty or not. It's a relatively common pattern that developers use to check whether the container's size is greater than zero instead of asking if the container is empty. The latter is preferred since calling the *empty()* function may be more efficient and increases the readability of the code with a clear intent to check for emptiness.

Example:
```
if (myVector.size() > 0) {
    // do something
}
```

Improved solution:
```
if (!myVector.empty()) {
    // do something
}
```

Use auto keyword

Iterator type specifiers tend to be long and used frequently, especially in loop constructs. The *auto* keyword introduced in C++11 allows the compiler to deduce a variable's type, which works perfectly in the case of iterators, since only one possible type can apply. Replacing a long complicated iterator type with *auto* improves readability and maintainability, and makes code less obscure.

Example:
```
for (std::vector<int>::iterator
          I = my_container.begin(),
          E = my_container.end();
          I != E; ++I) {
}
```

Improved solution:
```
for (auto I = my_container.begin(),
          E = my_container.end();
          I != E; ++I) {
}
```

Frequently, when a pointer is declared and initialized with new, the type of the pointer is written twice – once in the declaration and once in the new expression. In these cases, the declaration type can also be replaced with *auto*, leaving a single instance of the type declaration, and again improving readability and maintainability.

Compiler-guided refactoring is a great way to automatically rewrite large portions of your source code to improve its performance and increase its readability.

Example:
```
TypeName *my_pointer =
          new TypeName(my_param);
```

Improved solution:
```
auto *my_pointer =
      new TypeName(my_param);
```

4) Project-specific transformations

Some projects require custom algorithms in order to transform large quantities of code in a specific way.

As an example, Clazy is a custom compiler plugin (developed by KDAB) that understands Qt semantics and gives the compiler the ability to check for more than 50 Qt related issues. In some specific cases, it can also automatically refactor code.

Fixing old style connects in Qt code

In Qt, the older syntax that connects events with *SIGNAL(...)* and *SLOT(...)* is much slower than its newer replacement, which uses a pointer to a member function (PMF).

Example:
```
connect(model,
  SIGNAL(registeredToView(
         KTextEditor::View*)),
  this,
  SLOT(disableKeywordCompletion(
         KTextEditor::View*))
);
connect(model,
  SIGNAL(unregisteredFromView(
         KTextEditor::View*)),
  this,
  SLOT(enableKeywordCompletion(
         KTextEditor::View*))
);
```

Improved solution:
```
connect(model,
  CodeCompletion::registeredToView,
  this,
  &ClangSupport::disableKeywordCompletion
);
connect(model,
  &CodeCompletion::unregisteredFromView,
  this,
  &ClangSupport::enableKeywordCompletion
);
```

Summary

Compiler-guided refactoring is a great way to automatically rewrite large portions of your source code to improve its performance, increase its readability, or take advantage of modern features.

You 'only' need to write the algorithms that analyze your existing codebase, apply the transformation, and output the updated code – something that requires a solid understanding of the C++ programming language, compiler parsing techniques, and compiler internals. While developing automatic code transformation is not for the squeamish, it is a technique that can provide tremendous timesaving when amortized over a large code base and with much less risk than thousands of manual source edits.

At KDAB, we've helped many companies with their modernization efforts. Should you want to bring your source code up to the latest and greatest C++ standards, we'd be happy to help you tackle the job.

Developing automatic code transformation is not for the squeamish but can provide tremendous timesaving with much less risk than manual source edits.

Sources for examples

- https://clang.llvm.org/extra/clang-tidy/checks/modernize-use-override.html
- https://clang.llvm.org/extra/clang-tidy/checks/modernize-loop-convert.html
- https://clang.llvm.org/extra/clang-tidy/checks/modernize-use-default-member-init.html
- http://releases.llvm.org/5.0.1/tools/clang/tools/extra/docs/clang-tidy/checks/performance-unnecessary-copy-initialization.html
- http://releases.llvm.org/5.0.1/tools/clang/tools/extra/docs/clang-tidy/checks/performance-for-range-copy.html
- https://reviews.llvm.org/D14779
- https://clang.llvm.org/extra/clang-tidy/checks/readability-simplify-boolean-expr.html
- https://clang.llvm.org/extra/clang-tidy/checks/readability-container-size-empty.html
- https://clang.llvm.org/extra/clang-tidy/checks/modernize-use-auto.html

Other sources

- https://www.cppdepend.com/modernizer
- https://www.kdab.com/clang-tidy-part-1-modernize-source-code-using-c11c14/
- https://dzone.com/articles/a-software-developers-guide-to-maintaining-code/