Compile-time Unit Testing

ÁRON BARÁTH and ZOLTÁN PORKOLÁB, Eötvös Loránd University

Unit testing is an essential part in high quality software development. The prerequisite of the system-wide test is that all components are working correctly. Such components are checked by unit tests which are focused on small part of code. Unit tests are isolated from other code fragments and are frequently programmed by using third party tools. In this paper we show how to minimalize the number of the required third party tools, and how to automate unit tests. We aimed to express unit tests as part of the source code, and execute them during the compilation to ensure the quality of the code. Two different approaches are presented in this paper: the first is based on the C++11's new mechanisms, such as *constant expressions*, and *static assertions*; the second is based on our experimental programming language, *Welltype*, which offers more flexible unit testing than C++11 – however, both technique are similar. The key in both approaches to get the compiler to execute additional, user defined calculations which can result errors during the compilation. In C++11 the static assertions are used to evaluate parts of the unit tests. Since constant expressions are restricted, in this paper we also present a method how to overcome those restrictions, and how to utilize static assertion with constant expressions. Finally, we describe how our experimental language offers compiler support to evaluate *pure* expressions at compile-time.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**] Language Constructs and Features; D.2.4 [**Software Engineering**] Software/Program Verification

Additional Key Words and Phrases: Programming languages, C++, C++11, Compile-time, Unit testing

1. INTRODUCTION

Modern software development have to take testing into account to ensure the reliability of the software product. Test-driven development [Beck 2003] requires to specificate the new features first by writing new test cases, and after implement it to fulfill the test cases. This method provides clear specification for the new features. Moreover, any reported bugs become test cases, and the way of fixing it is the same. Finally, all written test cases are part of the regression test.

Two major kind of tests are known: black-box and white-box testing [Schach 2002]. The *black-box* tests are focused on the input and the output: for specific input, the specific output must be provided, and no matters how. We could say black-box tests are testing the preconditions and the postconditions [Plosch and Pichler 1999]. The *white-box* tests are dedicated to get as high code coverage as possible by providing different inputs to execute distinct parts of the code. Keeping the white-box tests cases up-to-date requires huge effort when the implementation changes, during a refactor for example.

The unit tests can be handled different ways. The tests can be written by hand as any regular program, and refer to the libraries which are tested. This technique is too unflexible, because numerous additional code is required to make detailed error messages. The more general way is to use an existing *test framework*, because of the detailed error messages, large number of tools which are helpful when

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

Authors' address: Á. Baráth and Z. Porkoláb, Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, H-1117 Budapest, Hungary; email: {baratharon, gsd}@caesar.elte.hu

In: Z. Budimac, M. Heričko (eds.): Proceedings of the 4th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications (SQAMIA 2015), Maribor, Slovenia, 8.-10.6.2015. Also published online by CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073)

1:2 • Á. Baráth, Z. Porkoláb

develop tests. Test frameworks are available for all wide-spread programming language, for example *gtest* [Google 2015] for C++ and *JUnit* [Tahchiev et al. 2010] for Java.

However, the test frameworks are usually third party extensions for a language, sometimes came with a platform-specific library. Furthermore, the created tests are regular programs, which must be executed manually – or they are part of the build system. When the execution of the tests are neglected, the quality of the software questionable. Therefore, we need compiler support or at least some compile-time mechanism to prevent the compilation of the wrong code at all. For example, the *Eiffel* programming language introduced the *Design by Contract* [Meyer 1992].

In this paper we show a new way to write tests cases in C++11, which are checked by the compiler. The method uses features are introduced in C++11: the constant expressions, which are expressions containing only constant expressions or literals, and the compiler is entitled to evaluate them during the compilation; and the static assertions, which are assertions evaluated during the compilation. However, these features are very restricted, and requires preparations in the source code to apply. We present the usage of these mechanisms by an example. Moreover, we present a similar technique in our experimental programming language: our ongoing development in our compiler is to evaluate *pure* expressions during the compilation. With this ability, our compiler will support a rich mechanism for compile-time unit testing.

This paper is organized as follows: In Section 2 we give a detailed guide how to write compile-time unit tests in C++11. The guide is presented with a concrete example. Also, we highlight the limitations of the method. In Section 3 we present our experimental programming language called *Welltype*, and its capability of compile-time unit testing. In Section 4 we describe our future plans to implement additional functionality for the compile-time unit testing. Our paper concludes in Section 5.

2. IMPLEMENTATION IN C++11

C++ is a multi-paradigm programming language designed for high performance programming [Stroustrup 1994; 2013]. In the continuous evolution of the language first object-oriented features have been introduced, later essential elements of generative programming, like templates and the STL library were invented. In recent versions in C++14, functional programming achieved more and more importance [Meyers 2014]. In the same time, C++ compile-time programming proved to be a Turingcomplete sublanguage [Veldhuizen 2003], and template metaprogram applications became popular [Alexandrescu 2001].

By design no language support exists in C++ to construct test cases. However, developers can use third-party tools and libraries to write tests. One of these tools is the Google Test, also known as *gtest* [Google 2015; Langr 2013; Sigerud et al. 2013], and the Boost Test Library [Rosental 2007]. All third-party tools carry the same problems:

-it is not ensured the availability on all architectures and operating systems;

-the developer's responsibility to execute the tests during the development.

The key idea is to relocate the functionality from third-party tools into the C++ compiler. This endeavor can be observed in other C++ specific areas, for example generating domain-specific languages at compile-time [Porkoláb and Sinkovics 2011]; or validating STL usages [Pataki et al. 2010].

The C++11 introduced the the static_assert mechanisms, and the programmers can write compiletime assertions. Furthermore, the constexpr is introduced for optimization purposes, since the compiler is entitled to evaluate all constexpr expressions. Note that, before C++11 the compiler could evaluate functions at compile time as well, but that is done by heavy optimization routines. The constexpr is more, because the compiler will accept a function as constexpr, when the all called functions are constexpr, and the functions itself has no side-effect. This definition excludes the usage of global variables. Consequently, the evaluation of a constexpr function depends only on the arguments. If all arguments are constants (literals, or return value of a constexpr function), the function will be evaluated at compile-time. Otherwise, it will be evaluated at run-time. The constexpr functions can be named as *pure* functions, since the return value depends on the arguments, and the result is the same every time. Furthermore, the C++14 relax the restrictions in the constexpr, so more complicated functions can be written [Smith 2013].

Putting the static_assert and the constexpr together, we can write tests which are evaluated at compile-time. The compile-time tests are the aid for all problems which came from the third-party tools, because all actions are performed by the compiler. So, there is no additional dependencies, and the C++ project is more portable. Using compile-time tests results more reliability, because the source code will not compile if one of the tests fail.

However, the compile-time tests requires specific preparation in the software project. The declaration of the sample class can be seen in Figure 1. Taking advantage of the behavior of the constexpr, all functions are annotated with the constexpr where it was possible. The constructors are emphasized, because it is mandatory to construct pair at compile-time. The targeted operation is the operator< – as it can be seen in the class declaration, it is a friend and a constexpr function.

```
class pair
{
    int x, y;
public:
    constexpr pair() : x(), y() { }
    constexpr pair(int x, int y) : x(x), y(y) { }
    constexpr int get_x() const { return x; }
    friend constexpr bool operator<(const pair & lhs, const pair & rhs);
    friend std::ostream & operator<((std::ostream & os, const pair & rhs);
};</pre>
```

Fig. 1. Preparations in the class declaration.

The implementation of the two declared operations in the class can be seen in Figure 2. The operator< has the ordinal behavior of a *less-than* operator. Note that, this function can be evaluated in compile-time and in run-time as well, thus, it can be used in static_assert statements.

```
constexpr bool operator<(const pair & lhs, const pair & rhs)
{
   return lhs.x<rhs.x || (lhs.x==rhs.x && lhs.y<rhs.y);
}
std::ostream & operator<<(std::ostream & os, const pair & rhs)
{
   return os << '(' << rhs.x << ", " << rhs.y << ')';
}</pre>
```

Fig. 2. Trivial implementation of the operators which belong to the pair class.

1:4 • Á. Baráth, Z. Porkoláb

In the sample code above, the implementation of the operator< is correct, however, the correctness of the operator is not trivial in most of the cases. Since the sample less-than operator can be evaluated at compile-time, and pair objects can be constructed at compile-time (see the constexpr), the code can be seen in Figure 3 is valid. Furthermore, the compiler evaluates them, and checks the correctness of the less-than operator based on the samples.

```
static_assert(!(pair() < pair()), "OK");
static_assert(pair() < pair(1, 0), "OK");
static_assert(pair(2, 5) < pair(5, 2), "OK");
static_assert(pair(2, 5) < pair(2, 6), "OK");</pre>
```

Fig. 3. Static assertions which pass at compile-time.

The counter-assertions, which fail on evaluation, can be seen in Figure 4. These assertions are intentionally wrong, and they are constructed to demonstrate the correctness of the static_assert.

```
//static_assert(pair() < pair(), "fail");
//static_assert(pair(3, 0) < pair(), "fail");</pre>
```

Fig. 4. Static assertions which *fail* at compile-time.

Note that, the static assertions in Figure 3 and in Figure 4 are placed right after the implementation of the less-than operator. Unlike the regular assert, the static_assert can be places everywhere, so the static assertions are placed outside all functions.

Using this technique, any test cases can be written inside the source code, and the compiler will evaluate them during the compilation. Furthermore, using the constexpr is a good practice, since the *pure* functions can be validated. Thus, the validation of the pure functions is an additional test, and also it is a motivation to write critical functions – such as less-than operator – side-effect free.

```
constexpr bool test_some(int from, int to)
{
   return (from>to) || (pair(from, 0)<pair(from+1, 0) && test_some(from+1, to));
}
static_assert(test_some(-100, 100), "test some");</pre>
```

Fig. 5. Generating test cases with recursive constexpr function.

Furthermore, test cases can be generated during compilation with constexpr functions. An example for the test case generator function can be seen in Figure 5. The sample generates 200 different test cases in total. However, different methods can be designed to increase the coverity.

2.1 Limitations

Unfortunately, the constexpr itself is a limitation, since only constexpr functions can be tested at compile-time. Many other functions, which use input and output (for example: file I/O, lock mechanisms, networking), and implement intentionally impure functionality (for example: random number generator, retrieving current time) can not be tested.

An other limitation is the standard library, because the STL containers can not be used in static assertions due to the lack of constexpr. But user-defined containers are permitted to use.

The assertions themselves can be wrong. However, this could be true for any other tests, so this is not a real limitation.

3. IMPLEMENTATION IN WELLTYPE

Our experimental programming language called Welltype [Baráth 2015], which is an imperative programming language with strict syntax and semantics. The language provides a strong static type system, in order to keep the source code clear and easily understandable. Due to the restrictions in the semantics, the language is resistant to numerous syntactical and semantical vulnerabilities [Baráth and Porkoláb 2014].

The Welltype programming language handles the difference between the *pure* and the *impure* functions. This information is explicitly written in the source code – a function without any from these two attributes defaulted to pure to prevent misuse. The pure functions in Welltype are nearly the same as the constexpr functions in C++14. The restrictions for the pure functions are clear: impure functions can not be used, usage of the global variables are forbidden. The compiler uses this information in several checks: for example, only pure functions can be used in *assertions*, and in *pre-* and *post conditions*. This is consequent, because it is unaccepatable when the expression in an assertion has side-effects. The meaning of the whole program can be totally different with and without assertions.

We are implementing a new feature in our programming language to support compile-time testing. The initial version uses the pure function mechanism, because the returned value of the pure functions are only depends on the arguments. Furthermore, the returned value is always the same, when the function get the same arguments. Due to this property, the testing of the pure functions are established.

The default output of our compiler is a binary which can be executed on our virtual machine. This virtual machine has some special instructions which needed to execute the compiled Welltype programs efficiently. Since the compiled code can be executed on a virtual machine, and this is part of the compiler, the compiler itself can execute (almost) any programs during the compilation – even if the compiler is in cross-compiler mode.

Putting the parts together, we can implement a compile-time validation mechanism to execute the tests. If one of the tests fail, the output of the compiler will be dropped, due to the failed test cases. The pure functions are important components in the mechanism, because the test results can be trusted for pure functions only.

A synthetic sample code can be seen in Figure 6.

```
pure function div(int n, int m) : int
pre { m != 0 }
post(R) { n, m ; old n >= R*old m &&& old n < (R+1)*old m }
assert { div(10, 2)==5 }
assert { div(0, 1)==0 }
assert { div(16, 6)==2 }
assert { div(16, 5)==3 }
assert { div(16, 4)==4 }
//assert { div(16, 4)==4 }
//assert { div(1, 0)==0 } // PreconditionViolationException
assert { div(2, 0)==0 ; PreconditionViolationException }
{
return n/m;
}
```

Fig. 6. An example to present the elements of the compile-time unit testing in Welltype.

1:6 • Á. Baráth, Z. Porkoláb

The compiler will gather the assert clauses during the compilation, and generates an other executable program in the memory. This new program will contain all of the assert clauses for every pure functions. When the new program is generated, the compiler will pass the binary to the virtual machine. If the virtual machine successfully executed the program, it means, all assertions are passed. It is important to notice, only the compiler program is used during the compilation and assertion evaluation. So, the tests will be checked at compile-time. Whenever an assertion fails while running the tests in the virtual machine, the output will be dropped. This is the expected behavior, because the implementation contains one or more errors.

The internal program, which checks the assertion clauses, will be built with the following algorithm:

- (1) gather all pure functions with at least one assert clause,
- (2) generate a function for each assert clause, which contains a regular assertion with the same condition,
- (3) if the assert clause contains an expected exception, add a single exception handler with the desired exception,
- (4) call all generated functions in the main block.

If the assertion in a generated function fails, an AssertionFailedException will be raised. This exception will passed through the main, and will be caught by the virtual machine. This error can be easily processed in the caller.

3.1 Limitations

The solution for the compile-time testing in Welltype uses the pure function mechanism – since the result can be guaranteed only for the pure functions – the same problem raised as in C++. In the current situation this limitation can be removed only at the expense of reliability. However, the reliability is the essence of this method.

Another limitation is the usage of external functions and types. Since any concrete implementation can be loaded into the virtual machine, the executed test will be meaningless despite of they necessarily be pure functions. Also, the dependencies will be required to compile a program, which is not acceptable. This limitation can be solved, if the *pre-* and *post-conditions* could be stored in the binary, and these conditions will be validated by the dynamic program loader, and compiler could generate mock objects for the conditions. Thus the testing mechanism could be more flexible.

4. FUTURE WORK

Our current method requires handmade test cases, even in case of the recursive test case in Figure 5. Using template metaprogramming, we can improve the method with automatic test case generation. For example, the input for the operator< can be generated by random numbers, and the developer have to write only a logical formula, such as |(a<b)|| pair(a, 0)<pair(b, 0) - or a similar formula in a general case. Furthermore, we are intented to apply axiom-based test generation [Bagge et al. 2011]. Thus the test cases could be more automated.

Our future plans to improve the Welltype compiler is to make able the compiler to generate white-box test cases which satisfy the pre- and the postconditions. Since the precondition defines an acceptable set of input, and the postcondition is a formal specification for the output, it is possible to generate test cases using the symbolic execution [King 1976]. This could be an outstanding and unique feature in our compiler to support the test-driven development.

5. CONCLUSION

In this paper we introduced a method with the related preparations to able to write compile-time unit tests in C++11. The method based on two new features are introduced in C++11: the constant expressions (constexpr) and the static assertions (static_assert). The reason of the selection was the fact they are evaluated by the compiler during the compilation. When the static assertion fails, the faulty source code can not be compiled. This feature is a great addition compared to any third party test frameworks. Also, we presented a basic test case generation by example – and it worked as expected. However, more complex generators could be written. The limitations of this method is summarized.

We introduced the abilities in our experimental programming language to describing compile-time tests. This language is called Welltype. The basic concept was to entitle the compiler to evaluate all pure expressions in the source code, since the result only depends on the arguments. Moreover, our language supports to attach assertions to function declarations. These asserts will perform as "static assertions" in C++11, but it uses a more flexible approach. However, the implementation is an ongoing work. The limitations of this method is summarized as well.

Our research is continued by constructing more advanced mechanisms in C++11 (or in C++14) to provide more flexible and richer kit for software developers. We intented to include template metaprograms to generate test cases. Also, in our Welltype compiler we intented to implement a white-box test case generator to reach a very high test coverage automatically.

REFERENCES

Andrei Alexandrescu. 2001. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, Boston, MA.

Anya Helene Bagge, Valentin David, and Magne Haveraaen. 2011. Testing with Axioms in C++ 2011. Journal of Object Technology 10, 10 (2011), 1–32.

Áron Baráth. 2014-2015. Welltype. (2014-2015). http://baratharon.web.elte.hu/welltype.

Áron Baráth and Zoltán Porkoláb. 2014. Towards Safer Programming Language Constructs. In 10th Joint Conference on Mathematics and Computer Science. 25.

Kent Beck. 2003. Test-driven development: by example. Addison-Wesley Professional.

Google. 2015. Google Test. (2015). https://code.google.com/p/googletest.

James C King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (1976), 385-394.

Jeff Langr. 2013. Modern C++ Programming with Test-driven Development: Code Better, Sleep Better. Pragmatic Bookshelf.

Bertrand Meyer. 1992. Applying 'Design by Contract'. Computer 25, 10 (1992), 40-51.

Scott Meyers. 2014. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media.

Norbert Pataki, Zalán Szűgyi, and Gergő Dévai. 2010. C++ Standard Template Library in a Safer Way. In Proc. of Workshop on Generative Technologies 2010 (WGT 2010). 46–55.

Reinhold Plosch and Josef Pichler. 1999. Contracts: From analysis to C++ implementation. In Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30 Proceedings. IEEE, 248–257.

Zoltán Porkoláb and Ábel Sinkovics. 2011. Domain-specific language integration with compile-time parser generator library. *ACM SIGPLAN Notices* 46, 2 (2011), 137–146.

Gennadiy Rosental. 2001-2007. The Boost Test Library. (2001-2007).

http://www.boost.org/doc/libs/1_57_0/libs/test/doc/html/utf.html.

Stephen R Schach. 2002. Object-oriented and classical software engineering. Vol. 6. McGraw-Hill New York.

Katarina Sigerud, Wojciech Sliwinski, J Nguyen Xuan, S Deghaye, X Piroux, V Baggiolini, JC Bau, Gennady Sivatskiy, and Ilia Yastrebov. 2013. Tools and rules to encourage quality for C/C++ software. Technical Report.

Richard Smith. 2013. Relaxing the constraint on constexpr functions. (2013).

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3652.html.

Bjarne Stroustrup. 1994. Design and Evolution of C++. Addison-Wesley.

Bjarne Stroustrup. 2013. The C++ Programming Language, 4th Edition. Addison-Wesley Professional.

Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. 2010. JUnit in action. Manning Publications Co.

Todd L. Veldhuizen. 2003. C++ Templates are Turing Complete. Technical Report.