An Exploration of C++ 14's N3597: Relaxing Requirements for ConstExpr Functions

Authors: Michelle Bray, Callan Fisher, Marc Simpson

We decided to explore the new C++14 update, implemented on August 18th of 2014. According to Bjarne Stroustrup, one of the major players in the creation of C++14, "[It] is simply the completion of the work that became C++11." The improvements in this update, although seemingly minor, are actually quite important. While deliberately small, they are the first big step towards making the language as a whole more appealing for novice users. It patches up some issues encountered with C++11, while making it a cleaner, simpler, and faster language.

In particular, we focused on approved proposal N3597, selected for inclusion in the release of C++14. This proposal aimed to relax a number of restrictions on constexpr functions. At the same time, however, it also imposes stricter rules when dealing with static and local variables to prevent side effects. These changes all received overwhelmingly strong or unopposed support under review of the Evolution Working Group. This update builds off of C++11, which introduced constexpr functions that were still rather restrictive. C++11 constexpr functions are executed at compile time to produce a value to be used where a constant expression is required. However, they could only contain a single expression that is returned. C++14 will relax these restrictions according to the following four major categories:

1. declaring a variable that is not static or `local_thread`
2. the ability to use if (else/ if else) and switch (but no goto)
3. the use of loops (for/ ranged-for, do/ do-while)
4. objects whose lifetime began within the constexpr evaluation can mutate.

Taking such changes into consideration, the new proposed definition of a constexpr function is as follows: First and foremost, it shall not be virtual. Furthermore, its return type shall be a literal type, as shall each of its parameter types. Its function body must be equal to delete, default, or a compound statement that cannot contain any of the following: an asm-definition, a goto statement, a try-block, or a definition of a variable of non-literal type or of static or thread storage duration or for which no initialization is performed. This differs greatly from the old definition which required that constexpr functions must contain only: null statements, `static_assert-declarations`, typedef declarations and alias-declarations that do not define classes or enumerations, using-declarations, using-directives, and exactly one return statement. Consider the following example code:

```
constexpr int prev(int x){
   return --x;                          // C++14 OK, C++11 error: use of increment
}

constexpr int g(int x, int n) {       // C++14 OK, C++11 error: body not just "return expr"
  int r = 1;
  while (--n > 0) r *= x;
  return r;
}
```

It is quite apparent that both these functions, while once illegal and prone to error, are now accepted under the refined definition of constexpr functions.

Apart from the obvious, one of the overarching goals of N3597's "relaxing constraints on constexpr functions," was to make C++ easier to learn and grasp. This was done by naturally combining constant expressions. C++11 had restrictions preventing natural combination between other parts of the language, such as 'for' loops. For example, you could only pass in a single expression so it wouldn't work. By removing these restrictions, however, not only does it become more straightforward to understand, but it also gives the programmer more freedom. Loosening these rules allows constant expressions to be passed multiple variables and still run at compile time. While C++11 allowed constexpr to be executed at compile time, it could only pass in a single value; With the C++14 update, on the other hand, it can be passed in multiple values or variables while still running at compile time. This allows for a more natural combination of expressions because constant expressions are no longer treated differently; they can handle multiple variables just like any other expression. This is a beneficial change because it unifies C++ while making it easier to use with less rules.

Now that constant expressions can handle multiple variables or values, it is possible to change objects within the constant expression; this is known as object mutation. Object mutation refers to objects that can be modified within the evaluation of a constant expression. This will occur until the evaluation of the constant expression ends or the lifetime of the object ends; however, mutable objects cannot be modified by later constant expression evaluation. Consider the following snippets of code:

```
constexpr int f(int a) {
  int n = a;
  ++n;                     // ++n is not a constant expression
  return n * a;
}

int k = f(42);               // int k = f(42) is a constant expression
                                 // function f can be modified because its lifetime
```

```
                                  // began during the evaluation of the expression

constexpr int k2 = ++k;       // error, not a constant expression, cannot modify
                              // k because its lifetime did not begin within
                                  // this expression

struct X {
  constexpr X() : n(5) {
    n *= 42;                      // Once again not a constant expression
  }
  int n;
};
constexpr int g() {
  X x;                            // initialization of x is a constant expression
  return x.n;
}
constexpr int k3 = g();       // int k3 = g() is a constant expression
                                  // x.n can be modified because the lifetime of
                               // x began during the evaluation of g()
```

This approach allows arbitrary variable mutation within an evaluation while keeping the essential properties of constant expression evaluation being independent from the mutable global state of the program. So a constant expression evaluates to the same value no matter when it is evaluated and excepting when the value is unspecified.

Although this doesn't seem like that big of a change, this new approach allows variable mutations within an evaluation while ensuring that constant expression evaluation is independent of the mutable global state of the program. With this change in place, the constant expression will evaluate the same value independently of when it is evaluated, except when the value is unspecified. A prime example is that of floating point calculation, which allows for differing orders of evaluation that return different results. The rules for use of objects whose lifetime did not begin within the evaluation are unchanged: they can be read (but not modified) if they are either declared with constexpr, or they are const and of integral or unscoped enumeration typec.

Richard Smith addresses both the C++11 and the C++14 definitions of a constexpr function in his online article, "Relaxing constraints on constexpr functions." C++11 states:

> A literal constant expression is a prvalue core constant expression of literal type, but not pointer type (after conversions as required by the context).

Each subobject of a literal constant expression, of either array or class type, must be initialized by a constant expression. A reference constant expression

is an lvalue (which essentially describes a function or an object) core constant expression that designates an object with static storage duration or a function. An address constant expression, on the other hand, is a prvalue core constant expression, either of type `std::nullptr_t` or of pointer type, that evaluates to one of the following: 1) address of an object with static storage duration, 2) the address of a function, or 3) to a null pointer value. A prvalue is a pure rvalue, which is a temporary object (or subobject thereof) or a value that is not associated with an object; however, this cannot be an xvalue. An xvalue refers to an object that is typically near the end of its lifetime; it is the result of certain kinds of expressions involving rvalue references. Collectively, literal, reference, and address constant expressions are unified under constant expressions. Conversely, N3597 proposed the following change:

> A constant expression is either a glvalue (a generalized lvalue) core constant expression whose value refers to an object with static storage duration or to a function, or a prvalue core constant expression whose value is an object where, for that object and each of its subobjects: 1) each non-static data member of reference type refers to an object with static storage duration or to a function, and 2) if the object or subobject is of pointer type, it contains the address of an object with static storage duration, the address past the end of such an object, the address of a function, or a null pointer value.

By relaxing the rules for constant expressions, we must be more strict with static local variables and how they are defined in order to prevent side effects. When declaring a constexpr function that contains a variable of static or thread storage duration, there needs to be additional restrictions to ensure that the evaluation of the code runs correctly. This means that you cannot set a static or unchanged variable to a variable as such:

```
constexpr int first_val(int n) {
  static int value = n;             // error: not a constant expression
  return value;
}
const int N = first_val(42);
int arr[first_val(422)];
```

If one is initializing a constexpr variable, rules need to be in place to prevent the dependence of the initial value of the variable on the order of the implementation of constexpr call. This means that the order in which you call variables is unimportant because it will find the error before the variables are actually called. In the above example, restricting constant expression declarations forces the program to terminate before it reads the variables. Restructuring the static and `local_thread` rules allows the constant expression.

In our personal opinions, we believe that the overall update to C++14 is a beneficial step towards improving the language as a whole. The primary goal of this particular update was to fix bugs; along similar lines, it also aims to incorporate small improvements to simplify mastery of the language. The particular proposal that we focused on, N3597, successfully aids in reaching this goal. This is primarily done by relaxing constant expressions while also improving the natural combination and flow of the language.

All in all, proposal N3597 was rightfully approved and included in C++14, the most recent improvement from C++11. Upon first inspection, the update may seem relatively minor, but it actually plays an important underlying role in helping the language transition to a more user-friendly state. Mr. Stroustrup explains his hope for the future of the language as a whole when he states, "I hope that the tide has turned so that C++ is becoming more novice friendly." This goal is theoretically obtainable in the next few planned updates, seeing as how changes in C++14 got the ball rolling, so to speak. This is especially evident with the relaxed restrictions for constexpr functions, the overarching goal of proposal N3597.

Sources:

http://meetingcpp.com/index.php/br/items/looking-at-c14.html

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3597.html

http://www.infoq.com/news/2014/08/cpp14-here-features

http://electronicdesign.com/dev-tools/bjarne-stroustrup-talks-about-c14

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