Spring 2015

Constrained Templates in C++

Braden L. Obrzut
The University Of Akron, blo15@zips.uakron.edu

Please take a moment to share how this work helps you through this survey. Your feedback will be important as we plan further development of our repository.

Follow this and additional works at: http://ideaexchange.uakron.edu/honors_research_projects

Part of the Programming Languages and Compilers Commons

Recommended Citation
http://ideaexchange.uakron.edu/honors_research_projects/39

This Honors Research Project is brought to you for free and open access by The Dr. Gary B. and Pamela S. Williams Honors College at IdeaExchange@UAkron, the institutional repository of The University of Akron in Akron, Ohio, USA. It has been accepted for inclusion in Honors Research Projects by an authorized administrator of IdeaExchange@UAkron. For more information, please contact mjon@uakron.edu, uapress@uakron.edu.
Constrained Templates in C++
An overview of C++ concepts lite and separate checking

Braden Obrzut
The University of Akron
May 4, 2015
## Contents

Abstract.................................................................................................................................................................2  
Introduction.............................................................................................................................................................2  
Background............................................................................................................................................................3  
  Constraining Templates in C++11.................................................................................................................................5  
Concepts.................................................................................................................................................................6  
  Terse Notations.........................................................................................................................................................8  
  Diagnostics..............................................................................................................................................................9  
The Implementation.....................................................................................................................................................9  
Discussion...............................................................................................................................................................10  
Conclusion...............................................................................................................................................................13  
References...............................................................................................................................................................13
Abstract

This paper seeks to answer what C++ concepts are, the benefit they provide to C++ programmers, and where we would like to go looking ahead. To accomplish this, it will examine the current state of generic programming in C++, look at what other languages do to avoid the issue, and how concepts can be used to solve the problem in a manner which provides more freedom to the programmer. In the general sense, concepts allow the programmer to specify preconditions which must be satisfied on the inputs to generic code. The desire being to fully describe what is required by the code is for earlier detection of errors in order to produce better, more terse diagnostics amongst other benefits.

This overview of C++ concepts additionally covers some of the work done in order to provide an implementation within the GNU Compiler Collection (GCC). Throughout the project, the concepts technical specification (TS) [3] has been a moving target. While we had hoped to look into leveraging the preconditions to check the template definition, a process known as “separate checking,” time was spent instead on refactoring the base TS implementation. This paper will touch on benefits of separate checking and many of the issues involved.

Introduction

In computer science there are many algorithms that can operate on a number of data types with similar interfaces. We desire to represent these algorithms in a generic or polymorphic fashion so that they can be written once with abstract data types and instantiated for whatever data structure is in use [2]. For example one may want to write a generic search algorithm.

```cpp
template<typename Iter, typename T>
Iter search(Iter start, Iter end, const T &needle) {
    do {
        if (*start == needle)
            return start;
        ++start;
    } while (start != end);
    return end; // Not found
}
```

The example could conceivably work for arrays, linked lists, vectors, or any other kind of container and find a particular element inside. There are, however, two issues of note in this example. The first is that there exists a set of assumptions that are being made within the code. Even though the template literally reads such that it takes any two type names, it is obvious that there exists some set of requirements that Iter must meet, specifically in terms of what interface and semantics the type is expected to implement, and even some other set of requirements that T
is expected to meet.

The second issue of note is that since we don't know anything about the underlying container implementing the iterator, we can't take advantage of any of the containers properties. Should the container be ordered, it may be worthwhile to perform some other search than the linear search presented. It would be ideal if the compiler could determine if the types meet the conditions for a faster searching mechanism and automatically resolve to the faster implementation.

Concepts are an abstraction of these requirements [5]. The theory behind concepts is not new, as, in addition to similar constructs in other languages [1], the C++ standard template library is already written in such a way that describes a template's requirements. What is new is the ability to express these concepts in C++ code and have the compiler enforce them allowing for clearer diagnostics and partial specialization based on the constraints.

The language technical specification for concepts is mostly implemented in GCC today. My contributions include: Providing initial support for C++14 variable templates, variable concepts as required by the concepts TS [3], template introductions as specified by the concepts TS, and changes to the core language semantics for resolving introduced template parameters.

For variable template, my goal was to implement the subset of the C++14 standard which was required for the concepts implementation. Since concepts never get instantiated, this means that only constant expression support was needed and only when used as an rvalue. This essentially was a task of determining an adequate way of representing variable templates internally. From there, variable concepts is an extension of the work to make the concepts semantic analysis code aware of the new construct.

Template introductions involve performing a tentative parse to find a concept id. The introduction list (parameters) then need to be matched to resolve a single concept. Originally the rules for doing this required a very strict one-to-one match with the resolved concept as far as variadic vs non-variadic templates go, but this ended up being relaxed as it required an unnecessary post processing step.

**Background**

Templates in C++ allow us to write code for which a type or constant will be provided at the point of use. With the current feature set we are able to write code which is defined “for all” types T, or “for any” integer N. This general idea of generic programming is applicable to many ideas in computer science including data containers and algorithms, which don't by nature operate on any particular data structure [2]. The problem is that it is very unlikely that the
generic code will be applicable for all possible inputs. The number of templates that are actually
valid for all possible inputs is close to zero. Instead there exists a finite set of assumptions that
are made (preconditions), which we can list as requirements [1]. Note that while generic
programs can be for types or a range of numerical constants, to avoid being needlessly verbose
most of this paper will assume it is understood that a constant can be used instead of a type.

It is important to understand what exactly a template is in C++. In languages like Java and to
some extent C#\(^1\) which have a strict hierarchy of types, generic programming is often
implemented via a mechanism known as type erasure. What this means is that generic code is
compiled as if the types were the lowest common denominator, usually the type Object. Generic
code is thus written assuming only what is available at the base type and the compiled code is
applicable to any type derived from it. The compiler can simply pretend the types were
otherwise and provide type safety. The take away here is that for any type derived from the base
class, the generic code will work since there can only be one instantiation. From an end user
perspective, this one instance serves all approach is very intuitive, but it comes at the cost of
requiring code to be written for a known safe mold.

For C++, as the name templates should imply, generic code is only parsed. One can imagine it as
if the compiler simply memorized the code (the declaration and/or definition) and took notes on
where inputs need to be substituted into the code, not unlike macros in C. At this point in time,
we can only determine that the sequence of tokens appears to be syntactically correct, but we
can't determine anything about the resulting program. When a template is used, the inputs are
placed into the code creating a specialization and the result is compiled in a process known as
instantiation. What is important to note here is that the actual compilation process doesn't occur
until the template is used, and only then can the compiler determine if the type produces a valid
program. If it doesn't, the compiler will produce errors very similarly to if you defined a specific
ad-hoc type. As a result the errors will point to the implementation details of the template rather
than a terse description of the assumption that was broken, and it will do this for each invalid
instantiation [1]. C++'s tendency to produce very large, cryptic error messages for even the
slightest mistake when using a template is very well known.

Much of the power of C++ templates comes from the idea of template specialization. Every
specialization of a template in C++ stamps out, as its instance, a free standing type. That is to
say vector<int> is exactly the same as if there was a type vector_int. There's no intrinsic value
establishing a relationship between specializations and the produced specialization has no special
behavior compared to non-template code. As a result, C++ gives the programmer the ability to
manually define specializations (known as explicit specialization), or even have an alternate

\(^1\) C# maintains some information about generics in their byte code, but they work similarly to Java except that the
VM has more optimization opportunities and it can generate instances for value types.
template cover a range of specializations (known as partial specialization). Some uses of explicit and partial specializations are to optimize for a particular type or to alter the range of types accepted by the template.

This brings us to concepts. By applying a set of requirements of template inputs during the specialization process, we can constrain the set of accepted inputs. If we can reject inputs before reaching instantiation, then the mess of errors that C++ produces when that process inevitably fails is adverted. Instead of pointing to a line in the template that failed to instantiate, the compiler can list the requirements that were violated. Additionally these can be used to overload template specializations and allow either more optimized implementations to be used where more assumptions can be made, or provide alternative implementations for disjoint types.

**Constraining Templates in C++11**

Before diving into concepts, it would be a good idea to first look at what solutions we have today to constrain the inputs of a template. As stated before, template specializations allow the programmer to adjust the range of valid inputs to a template. To give a pure demonstration of this, let’s assume we want to make a template that gives the maximum value of a type.

```cpp
template<typename T>  
T maximum();  // No body so specialization is required to avoid link error.

template<>  
int maximum<int>() { return INT_MAX; }

template<>  
short maximum<short>() { return SHRT_MAX; }
```

This function would be valid for the types int and short. Any other types would compile as a forward declared function until link time where an error is generated due to the lack of a definition. While it’s not exactly ideal to have the error appear that late, there are reasons one may decide to implement this way although outside the scope of this paper. With C++11 there is now static_assert which would allow the primary template (most generic case) to have a body which can’t be called. With static_assert one could also build a set of rejection specializations instead, but in either case the downside is that all types must be known ahead of time.

A more generic option would be to take advantage of what is known as SFINAE (substitution failure is not an error) [1]. In certain context, C++ will throw out template declarations if the substitution process fails rather than producing an error. This can be combined with template metaprogramming to selectively enable a template based on some condition, typically a type trait.
With this we have achieved our goal of making a template function which returns the maximum value of any integral type and won't compile for any non-integral type. At the same time though the code, specifically the function prototype, has become completely unreadable. Even if the documentation was written in such a way as to attempt to hide these details, the errors generated from invalid inputs (i.e. when there exists no possible matching template) will expose these details in a manner which doesn't tell the user explicitly what went wrong. The example given here has trivial requirements compared to more useful templates, so it is easy to imagine how convoluted this method can become. Additionally, since this is not the intended use for templates, the compiler has to do a fair amount of work in order to instantiate every template used here which will result in slower compile times compared to a proper solution.

### Concepts

Concepts are a new programming construct which represent a set of requirements on an input argument [5]. A more useful way of viewing what concepts do is it that a concept specifies a set of valid types for a template argument. That is the set of types defined by the rules specified as the requirements for the type to be in the set. These rules can be type traits similar to the metaprogamming example, or they can require that some syntax fragment be valid for the given type.

In C++ a concept is defined by a boolean function or variable template with the concept keyword. The body of these templates should be a “requires clause” which may contain any expression which operates on the template parameters, including referencing other concepts, but requirements are typically given within a requires expression. The requires expression allows for constraints to be written based on properties of expected to be valid syntax fragments, or based on expected to exist types. The expressions can be written in terms of placeholder variables. These variables have no storage or linkage and exist merely to form valid expressions. Note that
a requires expression refers only to the language construct which specifies individual (atomic) requirements where as a requires clause consists of the conjunction and disjunction of requires expressions or concepts.

```cpp
// Function concept
template<
typename T>
concept bool EqualityComparable() {
    return requires(T a, T b) {
        { a == b } -> bool;
    };
}

template<
typename T>
// Variable concept
class bool ReversibleContainer = Container<T> && requires(T a) {
    typename T::reverse_iterator;
    typename T::const_reverse_iterator;
    { a.rbegin() } -> typename T::reverse_iterator;
    { a.rbegin() } -> typename T::const_reverse_iterator;
    { a.rend() } -> typename T::reverse_iterator;
    { a.rend() } -> typename T::const_reverse_iterator;
    { a.crbegin() } -> typename T::reverse_iterator;
    { a.crend() } -> typename T::const_reverse_iterator;
};
```

Statements within a requires expression (requirements) come in four varieties: Simple, type, compound, and nested. The first is a simple syntax fragment terminated with a semicolon. These merely check that the given syntax is valid without checking any of the properties of the given expression. The type requirement specifies a typename which must exist. A compound requirement, seen in the above ReversibleContainer example, works like a simple requirement, but may also check that the result of an expression is convertible to a specific type. They may also check that the expression does not throw an exception (noexcept). The last type is a nested requirement which allows another requires clause to be included within a requires expression. These would be used if a concept check is needed based on an expression utilizing the placeholder variables.

The most basic place to utilize the defined concepts is by the new requires clause in a template header. These are introduced by the requires keyword following the template parameters.

```cpp
// This function only takes object which satisfy ReversibleContainer
template<
typename T> requires ReversibleContainer<T>
void f(const T &obj) { /* ... */ }
```

Template requires clauses solve most of the issues with the template metaprogramming approach. The requirements for a given template can be reduced to a single named concept, and of course when diagnosing a constraint failure, the compiler can provide the user with the concept and requirement that failed as opposed to implementation details. However, this particular method of constraint association still leaves room for improvement in documentation.
clarity, especially in the case where there are multiple template parameters with constraints.

**Terse Notations**

The concepts technical specification [3] includes a few ways of using concepts without having to write out the template requirements in long form. The first of which is to use the concept in the template header in place of the typename keyword, similar to non-type arguments. This will result in a requirement being synthesized which takes the given parameter. This method can be used even if the concept has more than one parameter by specifying the template id forgoing the first parameter.

As outlined before, concepts can stand for a set of valid types. This means that they can stand in place of an explicit type, much in the same way that auto is used in C++14. In the context of function parameters, auto could be seen as a concept with no requirements. For this case, a template argument is synthesized automatically with that argument being used as the input for the concept. For the sake of providing for terse function prototypes, if the function has multiple parameters which have the same concept as their type, then they will share the same template parameter.

```cpp
auto max(const LessThanComparable &a, const LessThanComparable &b) -> decltype(a); // a & b have the same type.
```

Finally, there are “introductions” which replace the traditional template header with a more terse form. Specifically they allow many templates with similar arguments and requirements to be expressed by conforming to the template header from a concept. Introductions do not need to strictly copy the concept's template header as parameter packs in a concept can instead be listed as individual named parameters.

```cpp
template<typename FwdIt, typename UnaryPred>
concept bool PredSearch = ForwardIterator<FwdIt> && Predicate<UnaryPred>;

PredSearch{ForwardIt, UnaryPredicate}(ForwardIt first, ForwardIt last, UnaryPredicate p);
PredSearch{ForwardIt, UnaryPredicate}(ForwardIt first, ForwardIt last, UnaryPredicate p);
PredSearch{ForwardIt, UnaryPredicate}(ForwardIt first, ForwardIt last, UnaryPredicate p);

// Also find_if, remove_if, etc

// Identical to:
// template<typename ForwardIterator, typename UnaryPredicate>
```

2 There are some subtle differences between auto and concepts in this case which are intentional.
All of these notations can be freely mixed, and it is important to note that none of these are designed to outright replace the long hand notation. What is recommended differs for each template. Deciding what notation to use should be based upon whatever form provides for the most readable code and documentation.

**Diagnostics**

One of the primary motivations for concepts was to allow the compiler to generate simpler diagnostics when invalid inputs were used with a template. The existing system waits until template instantiation in order to produce errors which often don't tell the end user concisely why their input type fails to compile, or if constrained via current means produces unreadable suggestions. In contrast, concepts allow issues to be named and for the specific requirements which were validated to be presented outside of the context of the implementation.

Unlike other languages which implement constraints on a type hierarchy, concepts specify more abstract requirements. This means that for any given concept, the complete requirement set can become quite large. Naive diagnostics which output all violated requirements presents an issue in that they may generate error messages which are equally large and unfriendly compared to the current solutions. It may be acceptable in many cases to list concepts by name along with a few selected requirements and require iterative corrections to be made. However, some large programs with longer compile times may find a more complete list is desirable.

The precise solution to this problem is not known at this time as there is little real world experience on which to base the solution. It is believed that some reasonable default verbosity level can be found and those with exceptional projects could use a compile time option to enable more complete error reporting.

**The Implementation**

As of this writing, there is a mostly complete implementation of the concept technical specification for GCC. It can be found as the c++-concepts branch in the GCC subversion repository.

The organization of GCC's C++ compiler is such that it performs lexical analysis on the input file. The sequence of tokens is then put through the parser, which somewhat conveniently has functions named such that they correspond to grammar terms in the C++ standard. During this phase semantic analysis is also performed. The resulting trees are then passed on for code generation. Implementing concepts mostly looks at the parsing and semantic analysis stage, with
the exception of changes to symbol generation to support resolution of constrained functions.

There were two major features that I was involved in implementing. The first being support for templated variables as rvalues. This was an unimplemented C++14 feature on which concepts depends on for using concepts without requiring the function call syntax. Since concepts are not instantiated, there was no need to implement symbol generation for templated variables although that work was finished soon after for GCC 5. Variable templates appear simply as template ids, so the work here came down to enabling bare template ids to be looked up as variables, which produces a template decl that results in a variable decl, and providing parallel interfaces with function templates.

The other feature was template introductions as described in the section on terse notations. This involved implementing the parsing rules and generating traditional template headers based on the resolution rules in the TS. Parsing a template header involved looking for the template keyword, but with introductions and tentative parse is needed. Introductions use the concept id, which once resolved to an overload set should be followed by an opening brace, otherwise it could be a variable or function declaration.

Introductions have a parameter list which consists solely of identifiers and if the parameter should be a variadic pack or not. Once parsed we can resolve the concept from the overload set using placeholder arguments based on the number of parameters in the introduction list using the normal resolution rules. Placeholders already existed for synthesizing parameters for the other terse notations, but a new flag was required for these placeholders to indicate if they should match the entirety of a parameter pack.

Throughout most of the project, the concepts TS was in a volatile state working towards becoming a PDTS. The concepts implementation had to be adjusted several times in order to conform to new changes that were being made. At the same time, while implementing the above features, changes were proposed and accepted based on how feasible the features were to implement in the compiler. For example, the original wording for the template introduction resolution required a strict match between the concept's template header and the introduction declarations. This was relaxed such that a variadic concept (C<class ... T>) can be resolved even if the template introduction does not indicate a parameter pack (C{A, B}). This was done as the resolution conforms with normal deduction rules for other terse notation and the restriction would require an additional post processing step to reject.

**Discussion**

Thus far, concepts have been used as a way of providing early diagnostics for template instantiation failures and as a way of selecting the most optimal template in an overload set.
From the end user perspective concepts also establish a contract with the template developer that if their input type meets the requirements then it should work with the template. However, there is currently no system preventing the template writer from including code which is not covered by the requirements, and there is nothing ensuring that other code called by the template is compatible with the callers requirements.

What is desired is for concept checking to be done on the template definition. This process is currently known as separate checking. The theory is that since concepts are a set of requirements, typically specifying valid syntax fragments, the compiler can ensure that every fragment of code within the template definition exists as a requirement. One way to accomplish this could be to essentially synthesize a type which conforms to the requirements and check that type. Although it may seem like performing checking on a definition should allow any conforming type to pass template instantiation, it may not be possible to make that guarantee within the context of C++. Separate checking however is more about providing a tool to help ensure that the template developer can establish the interface contract with the end user.

The abstract description of the feature is somewhat deceptively simple, but due to the complexities of the C++ language there are many issues to consider. Some time was spent analyzing the various issues with implementing separate checking based on the concepts lite TS. The thoughts resulting from the discussions are outlined below for future work.

Unlike constraint checking during specialization, separate checking may constrain the options the compiler has for compiling template code. That is, if we make the assumptions that built-in language constructs are always available, it is possible to subvert the requirements. Take the following as an example.

```cpp
template<
    typename T>
requires requires(T x) { {x} -> bool; }
void f(T x, T y)
{
    // Valid?
    if(x && y) { }
}
```

This would pass checking since operator&&(bool, bool) is assumed to be available and T is convertible to bool according to our constraint. However, C++ has operator overloading so if bool is the best conversion is not known at the point of definition. This is a known problem with implementing some algorithms generically and there are two possible solutions currently. With separate checking it would be desirable to constrain the operator's overload set to just the built ins.

```cpp
struct S1 {
```
int operator&&(const S1 &other) const { return 0; }
operator bool() const { return true; }

template<typename T>
requires requires(T x) { {x} -> bool; }
void f(T x, T y)
{
    // For T = S1: Logical behavior subverted.
    if(x && y) { }

    // Alternatives:
    if(bool(x) && bool(y)) { }
    if(x) {
        if(y) { }
    }
}

There are also issues in determining what other code a constrained template can call. The case for calling other constrained templates is obvious in that more constrained functions (i.e. a template with more requirements) can call less constrained functions. That is, if some function f1 requires the concepts C1 and C2, then it can call a function that requires only C1 or a function which requires only C2. This follows the normal subsumption rules defined in the concepts specification.

However a question is raised as far as how to handle unconstrained code. By necessity, as checking an unconstrained template wouldn't allow anything to be done with the arguments, unconstrained code can call any other template. This becomes a very important question when dealing with legacy code which is all unconstrained. More specifically it would rule out using any algorithm from the standard template library. There are two related issues to the point of unconstrained code. First is if there should be some escape hatch in order to execute unchecked code. The idea of an unchecked block was included in the C++0x concepts proposal which assumed all code was checked by default. Having an unchecked annotation would allow for the developer to be required to acknowledge their call to unsafe code.

The other issue is how normal non-templated functions (including C functions) should be handled. Can a template assume that normal functions are available to constrained code, or should all called functions be specified as a requirement. This actually becomes important as the overload set for a particular function call can be dependent on the context in which it is called. With proposals for unified call syntax (allowing member functions to be called as free standing functions or vice versa) this problem could become more interesting. A possible solution would be to have a checked template evaluate the overload set at the point of definition and only perform overload resolution on that particular subset. As far as calling unconstrained code goes,
the current belief is that it should be possible and that the template developer should be trusted to understand that the code is potentially unsafe.

One area that is not especially clear is how to handle type trait requirements in the context of checked code. Each trait will need to be examined to determine precisely in what way a type trait constrains a type and translate that to what syntax is available. The required work here is not complete. A more basic implementation of separate checking can be done while ignoring this issue while still assessing many of the other issues.

Lastly there is a question about implementation speed. Part of the reason that C++0x concepts was trimmed down to the concepts lite specification that is used today is that checked concepts were demonstrated to impose a significant compile time penalty. Implementation experience is needed in order to accurately assess if this is still valid for concepts lite.

**Conclusion**

With the technical specification now in PDTS, the expected semantics and available features should no longer be changing. Looking ahead there are still a few features in the specification left to implement into GCC including generalized auto although this is not considered critical for mainlining. The implementation is currently being considered for merging into the mainline development pending some fixes.

As far as further research on concepts, the discussion above on separate checking is still worth investigating. It does however appear to have become a more complex problem than imagined going in, but it is still believed that acceptable results can be obtained. Pending more real world usage data, the issue of diagnostic verbosity is also worth looking into.

Concepts are a powerful feature for library developers. They not only empower the compiler to select better template specializations to produce better performing code, but also help clearly communicate requirements to end users.

**References**


