Annales Mathematicae et Informaticae
Accepted manuscript
DOI: https://doi.org/10.33039/ami.2023.08.006
URL: https://ami.uni-eszterhazy.hu

A static analysis approach for modern iterator development

Dániel Kolozsvári, Norbert Pataki

Department of Programming Languages and Compilers,
Eötvös Loránd University,
Budapest, Hungary
kolozsvari.dl@gmail.com
patakino@elte.hu

Abstract. Programming languages evolve in the long term, new standards are specified in which new constructs appear, old elements may become deprecated. Standard library of programming languages also changes time by time.

The standard of the C++ programming language defines the elements of the C++ Standard Template Library (STL) that provides containers, algorithms, and iterators. According to the STL's generic programming approach, these sets can be extended in a convenient way. The `std::iterator` class template had been in the C++ since beginning and has been deprecated in the C++17 standard. This class template's purpose was to specify the traits of an iterator. Typically, it was a base class of many standard and non-standard iterator class to provide the necessary traits. However, the usage of iterator is straightforward and fits into the object-oriented programming paradigm. Many non-standard containers offer custom iterators because of the STL compatibility. Using this base class does not cause any weird effect, therefore usage of iterator can be found in code legacy.

In this paper, we present a static analysis approach to assist the development of iterator classes in a modern way in which the iterator class template is not taken advantage of. We utilize the Clang compiler infrastructure to look for how the deprecated iterator classes can be found in legacy code and present an approach how to modernize them.

Keywords: C++, static analysis, iterator, Clang

AMS Subject Classification: 68N19 Other programming techniques

Submitted: October 29, 2022
Accepted: August 7, 2023
Published online: August 8, 2023

1. Introduction

Every programming language evolves regularly. New standards of programming languages published, new compiler techniques and constructs become available. For instance, many different Fortran standards have been developed in the last sixty years and many constructs have evolved during the years [7]. In 2022, Oracle announced the nineteenth release of the Java standard [15].

This language evolution takes part in the history of the C++ programming language, as well. C++98, C++03, C++11, C++14, C++17, and C++20 are the official standards. C++23 is already done, but it is not offial yet. Therefore, no compiler supports the entire C++23 standard recently. New standards may affect the core language and its standard library.

In general, language standard updates introduce new language constructs and may deprecate older constructs [3]. For instance, C++11 made the template class `std::auto_ptr` deprecated and provides new standard smart pointers instead [2]. Later, `std::auto_ptr` has been removed from the C++ standard library.

New language constructs and standard libraries can require migration in code legacies with a method called *source code rejuvenation* that is not considered code refactoring [13].

The `std::iterator` class template had been in the C++ since beginning and has been deprecated in the C++17 standard [10]. This class template's purpose was to specify the traits of an iterator [11]. Typically, it was a base class of many standard and non-standard iterator class to provide the necessary traits [12]. However, the usage of `iterator` is straightforward and fits into the object-oriented programming paradigm. Many non-standard containers offer custom iterators because of the Standard Template Library compatibility [1]. Using this base class does not cause any weird effect, therefore usage of `iterator` can be found in code legacy.

In this paper, we present a static analysis approach to assist the development of iterator classes in a modern way in which the iterator class template is not taken advantage of. We utilize the Clang compiler infrastructure to look for how the deprecated iterator classes can be found in legacy code and present an approach how to rejuvenate them. Clang's checker approach is proper to detect and emit warning based on static analysis [8].

This paper is organized as follows. In Section 2, we give an overview about the C++ Standard Template Library (STL) and iterators. We detail our approach in Section 3 and we present its evaluation in Section 4. Section 5 provides possible ways of the future work. Finally, this paper concludes in Section 6.

2. Iterators

C++ Standard Template Library is an examplar library based on the generic programming paradigm [1]. STL provides containers (e.g. `std::vector`, `std::map`) and container-independent algorithms (e.g. `std::max_element`, `std::sort`) [14].

These components are separated and they can be extended simultaneously in a non-intrusive way. Iterators bridge the gap between containers and algorithms that are abstraction of pointers [9].

In the C++ language, it is possible to access or manage the memory directly from the source code. One of the tools provided by the language which can be used for such purposes are the pointers – these are a special kind of variables. They store an integer value which represents the memory address to which the variable is pointing to, therefore the information stored in that memory block can be retrieved or modified by using the pointer for it. In contrast to reference variables, the pointers can store a different value than the one which they were initialized with: the memory addresses they point to can be shifted in both directions (forward or backward) based on the allocated size of the type of the value which they hold the pointer for – this is done by using pointer arithmetics.

However, pointers are compound types, so they do not store much additional information or metadata about themselves, nor have we the ability to customize them – how dereferencing the variable, or shifting it should happen exactly. To solve these issues we could use the concept of iterators. An iterator is an object which can be used to maintain an element of a given range, using a set of operators. A special form of the iterators are the pointers, however, sometimes we do not need to have all the capabilities of a pointer implemented in our custom iterator: depending on the use-case, it might be enough to have an iterator which is only capable of stepping forward, or can only be written to but it does not have the ability to be read. To achieve this, iterators can be sorted into one of the five main iterator categories: *input*, *output*, *forward*, *bidirectional* or *random access iterators*.

On top of the customized methods iterators can – and in a lot of cases have to – define additional information about themselves. This information is available in the form of iterator traits: there should be five iterator traits defined in total. The `difference_type` should express the result of subtracting one iterator from another, `value_type` stores information about the type of the value which the iterator points to, `pointer` is the type of a pointer which can point to the value maintained by the iterator, so is `reference` but instead of pointers the reference type is described, `iterator_category` shows us into which one of the iterator categories does the iterator belong to. The metadata defined by the iterator traits will be used by several STL algorithms to provide the most optimal behavior, or to be able to check whether the instance of the iterator type provided to them is implementing all the operators or methods they require, so the iterator object has all the capabilities they need [14].

2.1. Defining custom iterators – legacy way

To check the capabilities of an iterator object, the `std::iterator_traits` class of the STL can be used – this wrapper class is needed when both pointers and iterator objects can be accepted. Based on the template parameter it receives, it can generate a proper definition through which all the needed information can be accessed e.g. by an algorithm, and accepts both pointers and iterator objects

as a template parameter. Before C++17, to declare all the information needed to this specific wrapper class needed by the majority of the standard algorithms, the STL provided us a helper class named `std::iterator` [9]. When creating our own custom iterator class, by deriving from the `std::iterator` it was possible to define all the needed iterator traits by passing them to the parent iterator class as template arguments:

```
struct DummyIteratorDepr :
    std::iterator<std::forward_iterator_tag,  // iterator_category
                  int,                        // value_type
                  int,                        // difference_type
                  int*,                       // pointer
                  int &                       // reference
                  >
{
public:
    DummyIteratorDepr(pointer ptr) {n = ptr;}
    DummyIteratorDepr& operator++() {return *this;}
    DummyIteratorDepr operator++(int) {return *this;}
    reference operator*() {return *n;}
    pointer operator->() {return n;}
    bool operator==(const DummyIteratorDepr &rhs) {return true;}
    bool operator!=(const DummyIteratorDepr &rhs) {return true;}

private:
    int *n;
};
```

Another advantage of inheriting the `std::iterator` to have compatibility with the STL containers and algorithms is that – by making use of the default template arguments the parent class has – we do not even have to define all the attributes if they do not have to be specific ones, or we are sure they will not be needed at all: `difference_type`, `pointer` and `reference` all have default values, which can be deduced from the values we provided to the mandatory `iterator_category` and `value_type` fields.

Since this tool provided by the standard library seems to be extremely useful, it would be understandable to ask why did it become deprecated in C++17? It is worth to mention, that the concept of iterator traits for providing compatibility did not become deprecated, only the `std::iterator` class, and the main reason for that is its ambiguity. Consider the following example taken from the standard [5]:

```
template <class T,
          class charT = char,
          class traits = char_traits<charT> >
class ostream_iterator:
  public iterator<output_iterator_tag, void, void, void, void>;
```

In this example, it is hard to understand which `void` stands for which attribute. Declaring an iterator like this could be very confusing and hard to read. Another reason for deprecating the class is that if the custom iterator itself depends on a template argument which is then passed to the `std::iterator` class, finding the traits during name lookup could fail:

```
template <typename T>
struct MyIterator : std::iterator<std::random_access_iterator_tag,
                                  T>
{
  value_type data;  // Error: value_type is not found by name lookup
};
```

The result would be the same if we put `void` instead of `T` to the parent iterator class as second template argument.

2.2. Defining custom iterators – modern way

As the usage of `std::iterator` would be deprecated now, all the attributes which are needed to describe our custom iterator class have to be declared explicitly by using type aliases:

```
class DummyIterator
{
public:
    using iterator_category = std::forward_iterator_tag;
    using value_type = int;
    using difference_type = int;
    using pointer = int*;
    using reference = int&;

    DummyIterator (int* ptr) {n = ptr;}
    DummyIterator& operator++() {return *this;}
    DummyIterator operator++(int n) {return *this;}
    reference operator*() {return *n;}
    int* operator->() {return n;}
    bool operator==(const DummyIterator &rhs) {return true;}
    bool operator!=(const DummyIterator &rhs) {return true;}

private:
    int *n;
};
```

Note that for declaring type aliases both `typedef` and `using` keywords can be used, in this specific case they would be semantically equivalent, since we do not make the aliases depend on template parameters. Despite their semantic equivalence the syntax would differ a bit, consider the following example:

```
using value_type_u = int;
typedef int value_type_t;
```

The latter one is the method of creating type aliases in the legacy way, but since C++11 the first one is preferred, as being a more powerful tool compared to the second one. To ensure this, clang-tidy already implements several checkers which would warn in case of using the old approach instead of the modern one [6].

We can see that by using the modern iterator definition method, it is much more readable and much easier to understand what properties a given custom iterator has. This way the original concerns regarding the usage of the `std::iterator` class have been overcome, however, we have to face a new problem, which in some situations could be uncomfortable: we lost the ability the make use of the default template arguments – since now we do not have any. We have to declare every trait properly for our custom iterator to be compatible with the standard library, even if some of them (the ones which had default values earlier) would be trivial. In the following, we will try to create a tool, or to be more specific a set of tools to help the transition between the old and new way of defining a custom iterator, and to help to avoid the potential incompatibilities between the STL and our custom iterators created by using the modern approach.

3. Verifying custom iterators with static code analysis

We will define two major problem categories that can be divided into smaller problems, then we will address these smaller problems with static analysis tools provided by the Clang compiler infrastructure. The two major problems would be the handling of legacy custom iterators, and detecting potential custom iterators defined without using class `std::iterator`. We will then decompose the latter by ranking potential iterator findings based on how likely it is, that the class we found is meant to be used as an iterator, which has to be compatible with the standard library. To achieve this, we implemented a new clang-tidy checker named *modernize-replace-std-iterator*, as part of the *modernize* checker category. The exact behavior and logic behind the checker is described below.

3.1. Transforming legacy custom iterators

Since the usage of `std::iterator` has become deprecated, it is better to avoid using it when developing custom iterators. To help this, we developed a static analysis tool based on Clang. Clang supports developing new tools, thus the built abstract syntax tree (AST) can be utilized, queried and visited. We have implemented an AST matcher to find and warn for every class definition which derives from the class `std::iterator`. An example for these kind of classes could be `DummyIteratorDepr`. However, we have to consider the cases when the custom iterator class is derived not directly, but indirectly from the deprecated iterator.

This means one of the parents of our iterator in the inheritance chain would have `std::iterator` as a parent class (or at least as one of the parent classes, since in C++ it is allowed to have multiple inheritance, which means that it is possible to inherit from multiple base classes to create a common derived type) [14]. To handle these issues, it is not enough to simply check whether the class in question has the `std::iterator` as a base class, but we should also check if one of its parents at any level has it as a parent.

```
struct DummyIteratorDeprDesc : DummyIteratorDepr
{
    DummyIteratorDeprDesc(pointer ptr) : DummyIteratorDepr(ptr){}
};
```

To avoid unnecessary and redundant findings, the warning will only be triggered if our class has inherited the legacy iterator in a direct way. Since all class definitions will be checked, we will cover all the possible results, all the nodes of all inheritance chains. To modernize our iterators it is needed to update that one exact base class, which we get the warning for, since all the newly declared type aliases will be inherited by all the (directly or indirectly) deriving child classes (if we make sure that the access specifier of the traits is at least *protected* but considering that the purpose of them is to provide information about the iterator to the outside world, we should declare them as public aliases).

We have now clarified that our approach would be the detection of direct inheritances, which has an additional advantage on top of avoiding duplicate matches and redundant steps. The scope of the analysis will be the translation unit which we are currently analysing - narrowing this down to our problem we get, that the scope of the analysis would be the class definitions described in the given translation unit. However, we will have cases when the removal of the `std::iterator` class would be a valid step without modifying any of the iterator definitions we have in our translation unit, despite the fact that they had the standard iterator class as an indirect base class. This is the situation when we have a "custom" iterator class in the middle of our inheritance chain, but outside the unit which we are analysing right now. In this case, two explanations are possible: one of them is that we will take care of the custom class when analysing the translation unit introducing it – the other one is that the custom iterator is defined outside of our project. If we face the latter, we have to trust the project defining the iterator will solve the issues caused by the deprecation of the standard iterator.

To provide more information to the developer, our tool not only warns about the class definitions mentioned above, but also gives hints about how to update them. After analysing a proper iterator class, we will get the following warning:

```
test_iterator.cc:12:8: warning: Derived from std::iterator,
which is deprecated since C++17. From C++17 type aliases
should be declared:
 using iterator_category = std::forward_iterator_tag;
```

```
 using value_type = int;
 using difference_type = int;
 using pointer = int*;
 using reference = int&; [modernize-replace-std-iterator]
struct DummyIteratorDepr : std::iterator<std::forward_iterator_tag,
                                         ...>
```

This is done by querying the concrete type parameters of the template instantiation we defined when inheriting the base class. It is worth to mention, that we would get the same list of arguments if we relied only on the mandatory template parameters:

```
struct DummyIteratorDepr :
    std::iterator<std::forward_iterator_tag,  // iterator_category
                  int>                        // value_type
{
  // ...
}
```

We now have all the information which is needed to automatize the transformation, which would result in the class definition having the iterator traits declared. Currently according to the scope and the goal of our tool, only a warning would be triggered, but as a future improvement it would be easy to implement the transformation itself by using the fix-it hints of the clang-tidy tool.

3.2. Detecting custom iterators

3.2.1. Analysing potential iterators

As we have described earlier, detecting usages of the deprecated custom iterator defining method is only one part of our goal. Another part would be to detect all the existing iterators, or classes which seem to be iterators which can face compatibility issues when used with the Standard Template Library. Also, we try to keep in mind the motivation behind the deprecation of `std::iterator` – readability is an aspect which should be considered when analysing the code.

First, we try to focus on the classes which – apart from some extreme cases – can convince the analyser that they are iterators, and they are used as if they were one. To achieve this, we will define two key criteria: the custom iterator should define at least one of the mandatory iterator traits (or should derive from a class which defines one of them), and an instance of this custom class can be used as an argument for algorithms defined by the Standard Template Library. The library defines a wide range of methods operating on a given range of elements, for multiple purposes. These algorithms can be found in the *algorithm* header, and since they are analysing/modifying ranges, or a range of elements, the range itself should be determined when trying to execute them.

This is done by passing iterators as arguments to them, which iterators can define the range by marking the start and the end of the range we want to use. When we create custom iterator classes, a typical usage would be to combine them with the powerful tools provided by the STL algorithms. Assuming that we have a custom iterator class named `I` declaring all the needed iterator traits properly, and the type of the value to which its instances are pointing to is `int`. In this case, we could find e.g. the value 2 in the following way (`i_begin` and `i_end` are instances of our iterator class, defining the range which we would like to analyse):

```
std::find(i_begin, i_end, 2)
```

In this example, if 2 is part of the range, the iterator pointing to the first occurrence will be returned, otherwise the result will be *i_end*, which points after the last element. Sticking to this example, this specific function will require all the iterator traits to be declared, otherwise compiling the code would result in an error. Consider the following example:

```
class CustomIterator {
public:
    using iterator_category = std::forward_iterator_tag;
    using value_type = int;
    using difference_type = int;
    ...
};
```

We have only three attributes defined, *pointer* and *reference* are missing. Because of this, a compilation error should happen, and we would get an error message similar to this:

```
error: no matching function for call to '__iterator_category'
...
substitution failure [with _Iter = CustomIterator]: no type
named 'iterator_category' in 'std::iterator_traits<CustomIterator>'
```

What interesting here is, that even if we had the trait `iterator_category` defined because of the template substitution failure of class `std::iterator_traits`, compiling a code like this will result in an error which can be misleading. Of course, if we would have used the legacy way for defining `CustomIterator`, the problem would not be present since the missing parameters could be determined by using the default template argument values of `std::iterator`:

```
struct CustomIteratorDepr :
    std::iterator<std::forward_iterator_tag,
                                        int,
                                        int
                                        >
```

```
{
  // ...
};
```

Now we have seen that despite its advantages, the modern approach prevents us to make use of the default template arguments of the legacy class. Using the legacy method, we do not have the possibility to skip any of the non-mandatory parameters, the order of the template arguments matters and should be considered to avoid failures during compilation, but it was a bit easier to define iterators which did not require all the iterator attributes being defined by their own.

However, not all algorithms make use of the iterator traits or require them to be present in the class defining the iterator they got as an argument. Let us take `std::swap_ranges` as an example. The function exchanges the elements of two ranges, and requires three parameters to achieve this: the first two parameters define the first range, the third one points to the beginning of the second range. Let us define our custom iterator in the following way:

```
class DummyIterator
{
public:
    DummyIterator (int* ptr) {n = ptr;}
    DummyIterator& operator++() {return *this;}
    DummyIterator operator++(int n) {return *this;}
    int& operator*() {return *n;}
    int* operator->() {return n;}
    bool operator==(const DummyIterator &rhs) {return true;}
    bool operator!=(const DummyIterator &rhs) {return true;}

private:
    int *n;
};
```

As we can see, we defined only the operators required by the `std::swap_ranges`, but none of the iterator traits. Based on the previous examples we have seen, using this iterator with for example `std::find` would lead to compilation error. This is not the case with this function:

```
std::swap_ranges(d1_begin, d1_end, d2_begin);
```

This example compiles just fine, if `d1_begin`, `d1_end` and `d2_begin` are all instances of `DummyIterator` we defined above. In case the method does not require the substitution of the template arguments defined by `std::iterator_traits`, it is possible to be compatible with the function by having only a number of traits defined (or defining none of them). At this point, we can divide the problem of being compatible with STL algorithms into two subcategories: in one case, the function call will compile just fine, in the other case a compilation error will happen. We

have implemented our clang-tidy checker to address both problems at the same time.

Matching the AST Our concept here would be to help to avoid compatibility issues with the standard library while keeping the code as much readable as possible. The base concept of our AST matcher is to find all nodes, which belong to the class declaration defining the objects which are used as parameters when calling functions from the `std` namespace. Note that we mentioned earlier that we are interested in the calls of functions defined in the *algorithm* header. This approach would be much more strict than analysing all the function calls using methods from the `std` namespace, however, members of the *algorithm* library are also part of it. The reasoning behind it is that we would like to help the developer to avoid incompatibilities in the future, for use cases which might not be relevant right now. If a custom iterator is used with the Standard Template Library, it has the potential to be used later together with a method, with which it would have compatibility issues resulting in unexpected errors, mainly during compilation time.

In case of larger code bases and rather complex projects, it is not unusual to have a lot of legacy code in it. Due to this, we have to handle the cases which would be covered by using the matcher of our checker tool described previously (detecting legacy `std::iterator` usage). These are the cases when the custom iterator inherits its attributes from the `std::iterator` class – therefore we exclude these matches, since they are not part of the scope of the current analysis.

To determine if the arguments are meant to be used as iterators, we are looking for the explicitly declared type aliases representing the iterator traits, or to be more specific, we are looking for one of the mandatory ones: `iterator_category`. Since the legacy cases had been excluded, all our custom iterators should define the two mandatory attributes, which are not derived from a custom iterator class coming from a third party library. However, based on this logic the false-positive findings should be considered too: what happens, if a class declares the type alias `value_type`, and an instance of it is used as an argument of a standard function, but it is not an iterator?

```
struct A
{
    using value_type = int;
};
...
std::vector<A> v;
A a;
v.insert(v.begin(), a);
```

This example meets all of our conditions, so `A` could be considered as an iterator. To avoid this, we only look for the attribute `iterator_category`, which is less likely to be defined in a class representing a different concept then the iterators. Another thing which we have to deal with is the case, when the class definition does not

contain the traits, but a base class of it does. In this case, the custom iterator has a more abstract iterator class from which it inherits the common attributes, which applies for this specific subtype too.

```
struct Iter_A
{
    using iterator_category = std::forward_iterator_tag;
    using value_type = int;
};

struct Iter_B : Iter_A
{
    using difference_type = int;
    using pointer = int*;
    using reference = int&;
};
```

In this example, the parent (`Iter_A`) only defines the mandatory attributes, the rest of them are present only in the derived class. To handle the problem of abstract iterators, the matcher would follow the following logic: if the definition of the parameter type contains the mandatory attribute, then this AST node should be matched, if not, the matcher will look for the node which implements it. If we have multiple matches, because the traits have been redefined multiple times in the inheritance chain, we will look for the first one, which declares the attribute (the top one). The motivation behind this is to find the first class definition which can potentially act as a standalone iterator itself. After we have found all the nodes which should be considered regarding a function call, the checker will analyse the class definitions and determine which iterator traits are missing. When we say "missing", it means that we are interested in what are the traits which are not declared in this exact class definition. The traits can be present without having to declare them: this is the case if the class inherits these attributes from a base class. After we have collected the missing type aliases, a warning will be triggered for the user to see what should be declared on top of the existing aliases. Using the class we declared earlier (`CustomIterator`) with `std::swap_ranges`, we will get the following warning:

```
test_iterator.cc:36:7: warning: Type seems to be an iterator used
by std::swap_ranges. The following type aliases should be
declared additionally within the class:

pointer
reference
```

In this case, the `pointer` and `reference` attributes were missing. By analysing the class definition further, it could be determined, or at least suggested how the iterator traits should be declared, but for now triggering a warning like the one

above is a limitation of our tool. Now let us see, what it means regarding the separate problem categories we defined:

Matching the base or the derived class If the base class is missing some of these type aliases, it could be useful to add them, since once we have done it, all the newly defined deriving classes would inherit all the iterator traits, which means they could be handled as an iterator by themselves. If we have matched the deriving class, declaring the traits can be redundant: if the base class (or in case of multiple inheritance, or a longer inheritance chain one of the base classes) also defines these attributes, the type alias in the deriving class will hide all the previous declarations, avoiding name collisions. However, by doing so the class definition could be much more readable since we would have all the aliases declared in one place, and it could be understood easily without investigating the parent-child relationships further. Of course, if none of the attributes are missing either in the base, or in the derived class, no warning will be triggered.

Matching function calls which would not compile We have mentioned earlier that in a number of cases it is mandatory to have all the traits defined properly, since they are needed by `std::iterator_traits`. It is possible for a custom iterator class to possesses all the values required via inheritance: in this case, triggering the warning could be relevant to have all the aliases declared in one place (see the case of base-derived classes). Our warnings will have one more advantage in case of calls which could not compile at all: it can give a hint, which traits are missing. As we could see earlier, it is possible that the error message triggered by the compiler only tells us that the substitution of the template arguments failed. In this case, our warning would highlight which attributes seem to be missing. However, it will not consider the attributes inherited, but a warning like this could be a motivation to define the class in a more comprehensible, readable way – otherwise these warnings would count as false positives, but since readability is one of the main aspect we follow, these warnings could be relevant also.

3.2.2. Analysing possible iterators

The last problem category which we wanted to cover consists of custom iterator candidates, which have the possibility – based on our conditions – to be treated as iterators. We can not be as confident as we were in case of the previous cases, regarding the false-positive results, since our conditions are much less strict for this category. The goal here also would be to provide readability and compatibility with the standard library, but the scope of our matchers will be much wider. We do not limit our findings to classes defining type aliases which could be interpreted as iterator traits, or to classes whose instances are being used as parameters for functions of the `std` namespace.

Our matching logic here will be similar to what is known as *duck typing* [4]. We will find and mark classes which have similar structure to an STL compatible iterator. The similarity in this case will not be defined by the members, types, etc.

defined by the class, but the member methods and operators overloaded by it. As we have seen in our previous examples, an iterator has to overload a given set of operators to be treated as a valid one. The set is determined by the type of the iterator - this is the same type which is stored as the value of `iterator_category`. As we have described earlier, all iterators must belong to one of the five iterator categories, which defines all the capabilities expected from the iterator.

Table 1. Iterator categories and their operations.

Output	`*p=`, `++`
Input	`*p`, `++`, `->`, `==`, `!=`
Forward	`*p=`, *(+Input iterator)*
Bidirectional	`-`, *(+Forward iterator)*
Random Access	`[]`, `+`, `-`, `+=`, `-=`, `<`, `>`, `<=`, `>=` *(+Bidirectional iterator)*

In Table 1, we can see all the required methods for each iterator categories. Based on that, if we see a class which implements all the operations needed for a forward iterator for example, we can mark it as a potential forward iterator. After we have done that, we can generate a warning that this class could be a potential iterator, and we can give a hint what values could be used for the iterator traits (in this example, `iterator_category` would get the value `forward_iterator_tag`). Similarly to the previous case, if all the traits have been defined by the class or one of its parents, the warning will not be issued. The reason why we match for the operations defined instead of the type aliases declared is, that – as we have shown earlier – in several cases the iterator traits are not needed at all by the function which takes the pointer as an argument, but this is not the case with the operator overloads. Missing a mandatory operator would result in a compilation error, hence relying on them would be useful. Also, the operators to overload (for example the unary `*`) are specific enough to match for them:

```
struct IteratorCandidateA
{
    IteratorCandidateA& operator++() { ... }
    IteratorCandidateA operator++(int n) { ... }
    int& operator*() { ..}
    int* operator->() { ... }
    bool operator==(const IteratorCandidateA &rhs) { ... }
    bool operator!=(const IteratorCandidateA &rhs) { ... }
};
```

The example shows us a candidate for the category *forward iterator*. If we find a candidate which defines all the operations needed by a category, which is a superset of another one (regarding the operators overloaded), then we will warn for it using the iterator category which would provide the most features.

4. Evaluation

To verify our three different approaches we have defined so far, we have executed the checker with all the different AST matchers in it on the LLVM project. LLVM is an open source compiler infrastructure, including Clang and also clang-tidy, the tool which we implemented our checkers in. Because of its nature, LLVM implements a number of custom iterator classes, making it a good candidate for our analysis. After analysing the results, filtering out all the duplicates caused by the same findings in header files included in multiple translation units, we had proper matches for all three categories, without any false-positive results.

In total, we have found examples for deprecated `std::iterator` usage in case of 52 class definitions, 1 custom iterator class whose instance had been used as an argument of a function defined in the Standard Template Library (and whose definition does not contain all the possibly required iterator traits). For the last category, we have identified a total of 12 iterator-like classes – class definitions which seem to be iterators based on the operator overloads they implemented.

After analysing the results further, we came to the conclusion that all the findings are valid, there are no false-positive matches among them. Based on these facts, it is proven that our tool can be used as a tool for modernizing the source code, and for updating the code base in a way, that future incompatibilities with the standard library can be avoided.

5. Future work

We have shown that our checker can be a useful tool when modernizing the source code, however, the matcher logics could be further refined, to find more accurate results. A refinement like this would be in case of the third problem category to not only match for the operator overloads (by name), but to consider the parameters and return types of these overloads also. However, in case of asterisk (`*`) operators the `void` return values are checked even now, to avoid false-positive findings for output iterators.

We have mentioned earlier, that in a number of cases rejuvenation of the code could be done automatically – we have all the information available which is needed to insert all the type aliases which are required to define the iterator traits: we can extract the proper types from the template arguments of `std::iterator` when deriving from it (in case of the first problem category), or we could define the missing attributes by analysing the return values and parameter types of the overloaded operators in case of the second and third problem categories.

In our example run, we have not faced any false-positive matches. However, earlier we have shown that finding faulty results might be possible. One improvement to avoid these findings would be to filter only for the members of the *algorithm* part of the STL, instead of matching for calls of functions defined in the `std` namespace. Also, cases when class definitions are hidden to the static analyser by macro definitions should be considered too.

6. Conclusion

Languages and their standard libraries evolve over time. For instance, C++17 provides much more constructs than C++98. On the other hand, some of the language elements become deprecated sometimes, C++98's `auto_ptr` is a typical obsolete component of the C++ Standard Template Library.

C++17 standard made `iterator` base class deprecated. However, its usage was common and safe, there was some reasons to make this class template obsolete. This class template is widely used when one develops a new iterator to specify the traits.

We implemented a static analysis method to emit warning if the usage of `iterator` base class can be found. Moreover, we presented an approach how custom iterators can be found. Our approaches provide hint how to improve the source code. Our method gives feedback if any trait is missing from iterator-like class. We have implemented a tool for the approaches based on the Clang compiler infrastructure. We evaluated our solution with real-world software artifacts, the result is promising.

References

[1] B. Babati, G. Horváth, N. Pataki, A. Páter-Részeg: *On the Validated Usage of the C++ Standard Template Library*, in: Proceedings of the 9th Balkan Conference on Informatics, BCI 2019, Sofia, Bulgaria, September 26-28, 2019, ed. by G. Eleftherakis, M. Lazarova, A. Aleksieva-Petrova, A. Tasheva, ACM, 2019, 23:1–23:8, DOI: https://doi.org/10.1145/3351556.3351570.

[2] B. Babati, N. Pataki: *Comprehensive performance analysis of C++ smart pointers*, Pollack Periodica 12.3 (2017), pp. 157–166, DOI: https://doi.org/10.1556/606.2017.12.3.14, URL: https://akjournals.com/view/journals/606/12/3/article-p157.xml.

[3] T. Brunner, Z. Porkoláb: *Programming Language History: Experiences based on the Evolution of C++*, in: Proceedings of the 10th International Conference on Applied Informatics, 2017, pp. 63–71, DOI: https://doi.org/10.14794/ICAI.10.2017.63.

[4] R. Chugh, P. M. Rondon, R. Jhala: *Nested Refinements: A Logic for Duck Typing*, in: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, Philadelphia, PA, USA: Association for Computing Machinery, 2012, pp. 231–244, ISBN: 9781450310833, DOI: https://doi.org/10.1145/2103656.2103686.

[5] *Deprecating Vestigial Library Parts in C++17*, 2016, URL: http://www.open-std.org/jtc1/sc22/%20wg21/docs/papers/2016/p0174r2.html.

[6] *Extra Clang Tools 12 documentation*, 2022, URL: https://clang.llvm.org/extra/index.html.

[7] A. M. Gorelik: *Statements, Data Types and Intrinsic Procedures in the Fortran Standards (1966-2008)*, SIGPLAN Fortran Forum 33.3 (Dec. 2014), pp. 5–17, ISSN: 1061-7264, DOI: https://doi.org/10.1145/2701654.2701655.

[8] G. Horváth, N. Pataki: *Clang matchers for verified usage of the C++ Standard Template Library*, Annales Mathematicae et Informaticae 44 (2015), pp. 99–109.

[9] S. Meyers: *Effective STL*, Addison-Wesley, 2001, ISBN: 0-201-74962-9.

[10] A. O'Dwyer: *Mastering the C++17 STL: Make Full Use of the Standard Library Components in C++17*, Packt Publishing, 2017, ISBN: 178712682X.

[11] N. Pataki: *Safe iterator framework for the C++ Standard Template Library*, Acta Electrotechnica et Informatica 12.1 (2012), p. 17.

[12] N. Pataki, Z. Porkoláb: *Extension of iterator traits in the C++ Standard Template Library*, in: 2011 Federated Conference on Computer Science and Information Systems (FedCSIS), IEEE, 2011, pp. 911–914.

[13] P. Pirkelbauer, D. Dechev, B. Stroustrup: *Source Code Rejuvenation Is Not Refactoring*, in: SOFSEM 2010: Theory and Practice of Computer Science, ed. by J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, B. Rumpe, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 639–650, isbn: 978-3-642-11266-9, doi: https://doi.org/10.1007/978-3-642-11266-9_53.

[14] B. Stroustrup: *The C++ Programming Language (special edition)*, Addison-Wesley, 2000, isbn: 0-201-70073-5.

[15] *The Arrival of Java 19*, 2022, url: https://blogs.oracle.com/java/post/the-arrival-of-java-19.