



Software Analyzers

ACSL++: ANSI/ISO C++ Specification Language Version DRAFT C++ excerpts only





list



ACSL++: ANSI/ISO C++ Specification Language

Version DRAFT

David R. Cok¹, ...

¹ CEA LIST, Software Reliability Laboratory, Saclay, F-91191

² France Télécom, Lannion, F-22307

³ INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893

⁴ LRI, Univ Paris-Sud, CNRS, Orsay, F-91405



This work is licensed under a “CC BY 4.0” license.
Sources are available at <https://github.com/acsl/acsl>.

©2018 CEA LIST and INRIA

This work has been supported by TODO

«Identify support»

TODO



Foreword [C++]

This document describes version DRAFT of the ANSI/ISO C++ Specification Language (ACSL++). The language features will evolve in the future. In particular, some features in this document are considered *experimental*, meaning that their syntax and semantics is not yet fixed. These features are marked with EXPERIMENTAL. They must also be considered advanced features, which are not needed for basic use of this specification language.

Acknowledgements [C++]

This language design and document incorporate nearly all the ACSL language and documentation; many portions of the text are explicitly shared. Some sections are nearly verbatim duplicates of the corresponding ACSL material, with light edits to reflect the use in ACSL++ as well. Other sections, marked [C++], are new to ACSL++.

Thus we acknowledge the extensive set of collaborators and commenters on ACSL that are listed in the companion ACSL documentation and replicated below.

The design of ACSL++ also draws from the specification languages of other object-oriented programming languages, such as Java [?].

The following have contributed explicitly to the formation of this document and the design of ACSL++ itself. We gratefully thank all the people who contributed to this document:

David Cok, Virgile Prevosto, «*Other contributors*»

TODO

Acknowledgements — ACSL

There are many contributors to ACSL, including principally the following (as listed in the ACSL Reference Manual): Sylvie Boldo, David Cok, Jean-Louis Colaço, Pierre Crégut, David Delmas, Catherine Dubois, Stéphane Duprat, Arnaud Gotlieb, Philippe Herrmann, Thierry Hubert, André Maroneze, Dillon Pariente, Pierre Rousseau, Julien Signoles, Jean Souyris, Asma Tafat.

Funding

The work on ACSL++ is done in the context of project VESSEDIA, which received funding from the European Union's 2020 Research and Innovation Program under grant agreement No. 731453.

$term ::= \backslash default_value \ (\ id \)$

Figure 1: Grammar of contracts about default values of formal parameters

0.0.1 Default arguments [C++]

In C++, functions can be declared to have parameters that can be omitted in a function call, with their values supplied by default values.

The arguments that have default values still have names, so the arguments can be referred to in function and statement contracts as usual. However it may also be useful to refer to the default value of the argument. This is done using the syntax `\default_value (<name>)`. The type of this term is the same as the declared type of the formal parameter, which may be different than the natural type of the expression giving the default value, since that expression may be implicitly cast to the argument's type. The additions to the grammar are given in Fig. 1.

As the `\default_value` syntax applies only to formal parameters, any use of such a term within annotations in the body of a method refers to the nearest enclosing formal parameter that has a default value declaration, no matter what other shadowing declarations may exist.

An example is shown in the code below.

«Insert example»

0.1 Class contracts [C++]

C++ allows the declaration of functions and data fields within the scope of the declaration of an aggregate type (class, struct or union). Similarly ACSL++ specification annotations may be present at class (or struct or union) scope.

0.1.1 Global declarations

As described in Section ??, logical specifications (predicates, functions, lemmas, and constants) may be defined at global scope. They may also be defined in class (that is, aggregate) scope. The following considerations apply:

- Just as for C++ identifiers, any names declared are part of the aggregate scope. The name may be referred to outside of the aggregate using C++'s qualification syntax; for example, `A::m` names a logic predicate declared as `m` in class or namespace `A`.
- As for any other specification constructs, those defined in aggregate scope are only visible in specification annotations.
- Within aggregate scope, a predicate or function definition may be declared `static`, with the same meaning as `static` has for C++ function declarations: non-static logic predicates and functions may refer to `this` and are invoked with the C++ arrow or dot syntax.
- Logic lemmas, logic type definitions, axiomatic definitions, and logic constant definitions are always implicitly static. The effect of declaring them within an aggregate is just one of name scope.

«Should give some significant examples»

TODO

0.1.2 Function contracts within class scope

Function declarations or definitions that belong to an aggregate may have function contracts just like functions declared or defined at global scope. If the function is not declared `static`, then the keyword `this` may be used within the function contract and within any statement contract within the function. The keyword `this` refers to the current object, just as `this` does within C++ code. The type of `this` in contracts is the same as the type of `this` in C++ code: `T*` or `const T*`, where `T` is the type of the enclosing aggregate.

0.1.3 Ghost functions

Aggregates may also contain declarations of ghost functions, as described in section TBD
«Complete this subsection»

TODO

0.1.4 Inheritance of function contracts

A key aspect of object-oriented programming is that one can derive classes from parent classes and in the process inherit behaviors of the parent class, while specializing the implementation to something appropriate to the derived class. To use a standard example, one might define an

abstract base class `Shape` to describe arbitrary 2D geometric shapes, with an abstract method `area` that returns the area of the shape. Derived classes `Circle` and `Square` would contain concrete data fields that defined the parameters of circle and square shapes respectively; the derived classes would each have its own implementation of the `area()` method appropriate to its kind of shape. However, the key idea is that methods of a `Shape` object can be used without knowing which derived class the object is actually an instance of.

The usual design intent is to obey Liskov and Wing's principle of *behavioral subtyping*^[?]: anything provable about a base type should be provable about a subtype. ACSL++ imposes that requirement on derived classes by requiring that methods of derived classes obey the contract of any methods of parent classes that the derived class methods override. If behavior subtyping does not hold between a base and derived class, then the behavior of methods invoked on a pointer or reference whose static type is the parent class (but whose dynamic class is any derived class) may depend on the actual dynamic class, a complex situation for program understanding, debugging, or verification.

Consider a derived class method `D::m` that overrides a parent class method `P::m`. Behavioral subtyping is obeyed if (a) the (composite) precondition of `D::m` implies the (composite) precondition of `P::m` and (b) the (composite) postcondition of `P::m` implies the (composite) postcondition of `D::m`. That is, preconditions may be more lenient in derived classes, and postconditions may be more strict.

This fairly straightforward requirement becomes complex in its interaction with function contracts that have multiple behaviors, as described in section TBD «*Sec ref needed*». ¹

0.1.5 Specifications of special member functions

C++ implicitly declares and provides a default implementation for some member functions under some circumstances. Thus they also have an implicit specification. However, the user may also wish to provide an explicit specification for such implicit functions, but there is no explicit declaration in the C++ code to which to attach such a specification.

In such a case, a corresponding declaration may be written as a ACSL++ declaration, with an attached specification. In the following example, class `C` has an implicitly defined copy constructor. The ACSL++ comment gives an explicit specification. No body is given for the declaration of the C++ copy constructor. Such declarations are only valid if the rules of C++ state that such a member function is implicitly defined.

```

1  class C {
2      public:
3          //@ logic integer value;
4
5          /*@
6              @   requires  \true;
7              @   assigns  \nothing;
8              @   ensures  this.value == that.value;
9              @ C(const C& that);
10             @*/
11 }

```

Grammar additions are needed

¹JML was designed explicitly around behavioral subtyping and the implications of behavioral subtyping are much simpler. See the comparative discussion in Appendix ??.

The member functions for which implicit default functions are provided by the compiler are the following. In each case the default action (and corresponding implicit specification) is to apply the corresponding operation on each base class and data member of the object at hand.

- default constructor `C()`:
- default copy constructor `C(C&)` or `C(const C&)`
- default move constructor `C(C&&)`
- default copy assignment operator `C::operator=(C&)` or `C::operator=(const C&)`
- default move assignment operator `C::operator=(C&&)`
- default destructor `C::~~C()`

The `operator new(...)` and `operator delete(...)` functions have the appearance of being implicitly defined. However, they are defined in the standard library (include file `new`). Their specifications are given there as part of the specifications of the standard library.

0.1.6 Deleted and defaulted definitions

C++ functions can be defined to be either deleted or defaulted, as in this code:

```
1 | C::C(const& C) = delete;
2 | C::C(const& C) = default;
```

A definition that is `=delete` results in a program for which any use of the given method results in a compile-time error, as if the function is not at all defined. It can be used to preclude the generation of otherwise implicitly generated functions.

A definition that is `=default` results in the corresponding function defined to have the body it would have as an implicit default member, even if the C++ rules would otherwise preclude the function from being implicitly generated. That is, it forces generation of the member function where it would not otherwise be implicitly generated.

Both of these features are handled by C++ and do not require extra features in ACSL++. Only, if a method is implicitly generated, whether by using `=default` or not, an implicit specification must be correspondingly generated.

0.2 Namespaces [C++]

C++ introduced namespaces as a means to structure the scopes of declared names. Namespaces apply equivalently to ACSL++ names. In particular,

- Any ACSL++ construct that may be declared or defined in global scope may be declared or defined within a namespace.
- Identifiers for such constructs may be constructed using namespaces names and `::` tokens, just as for C++ names.
- Name resolution of ACSL++ names is performed just as for C++ names, with ACSL++ names being in scope only within ACSL++ annotations.
- It is preferable to avoid using names declared in C++ as the names of ACSL++ constructs as well. However, since namespaces can be extended, the author of ACSL++ annotations may not know about the C++ names that may be added in a namespace extension. Thus ambiguities can in principle arise: a C++ declaration and an ACSL++ declaration may be equally applicable to a use of that name in some ACSL++ construct. In that case the ACSL++ declaration applies. Tools may wish to emit warnings for such cases.

«This rule for ambiguity resolution is open for discussion.»

<i>abrupt-clause</i>	::=	<i>throws-clause</i>
<i>abrupt-clause-stmt</i>	::=	<i>throws-clause</i>
<i>throws-clause</i>	::=	<code>throws <i>pred</i> ;</code>
		<code>throws { ... } <i>pred</i> ;</code>
		<code>throws { <i>C-type-name</i> (, <i>C-type-name</i>)[*] } <i>pred</i> ;</code>
<i>term</i>	::=	<code>\exception</code>

Figure 2: Grammar of contracts about exceptions

0.3 Exceptions [C++]

Exceptions are a C++ feature to enable structured alternate termination of a function, particularly when an error has occurred that is not locally recoverable. Although it can be challenging to implement exceptions efficiently, from the perspective of specification, exceptions are straightforward: they define alternate control paths, just like the various abrupt termination mechanisms of section ??.

The syntax additions related to exceptions are given in Figure 2. Note the following points:

- A throws clause may be part of either a function contract or a statement contract.
- A throws clause may include a list of type names; these are the names of exception types for which the clause applies.
- The list of types may optionally be simply ..., meaning, as in C++ that the clause applies for any type.
- A throws clause with ... for the list of type names is equivalent to the form with no type names at all.
- The new term `\exception` may be used within the clause's predicate to refer to the exception object being thrown.
- `\exception` may be used only in the form of *throws-clause* that gives one or more type names. The type of `\exception` is `T&`, where T is the stated type.
- If more than one type name is listed, the clauses's predicate must be syntactically and semantically valid for each type name, with `\exception` taking on each type in turn.
- There is no implicit order to *throws* clauses, as there is for *catch* clauses.

The semantics of a *throws-clause* is that if the function or statement exits with an exception having a listed type (or any exception if no types are listed), then the predicate must be true in the post-state of the function or statement. This is similar to the *ensures* and the other abrupt termination clauses: if the program construct terminates in the stated way, then the associated predicate is true in the post-state (the pre-state for the *exits* clause).

Exception specifications are being deprecated in C++. Consequently, ACSL++ does not specify or encourage tools to reason about C++ exception specifications, except for uses of **noexcept** (cf. Section ??).

«Add example»

TODO

0.4 Attributes [C++]

C++ allows decorating declarations with *attributes*. These can be thought of as supplying small amounts of specification information.

0.4.1 `[[noreturn]]`

The `[[noreturn]]` attribute on a C++ function declaration means that the function never returns: if it terminates at all, it always throws exceptions. Consequently, the only appropriate `ensures` clause is `ensures \false ;`.

Thus

- It is a syntactic error if a function has both a `[[noreturn]]` attribute and an `ensures` clause whose predicate is not the boolean literal `\false`.
- If a function with an `[[noreturn]]` attribute has a behavior that contains no `ensures` clause, the default `ensures` clause is `ensures \false ;` (rather than the usual `ensures \true ;`).

0.4.2 `[[noexcept]]`

The `[[noexcept]]` attribute on a C++ function declaration means that the function never throws an exception: if it terminates at all, it always terminates normally. Consequently, the only appropriate `throws` clause is `throws \false ;`.

Thus

- It is a syntactic error if a function has both a `[[noexcept]]` attribute and a `throws` clause whose predicate is not the boolean literal `\false`.
- If a function with a `[[noexcept]]` attribute has a behavior that contains no `throws` clause, the default `throws` clause is `throws \false ;` (rather than the usual `throws \true ;`).

0.5 Pure functions as logic functions

As part of information hiding, programs often have utility functions that simply retrieve a value from a class field; using the utility (‘getter’) function hides the actual representation of the concept the value returned by the function represents. The value returned may just be the value of an internal field or it may be the result of a more complex calculation.

It is often necessary to replicate such functions as logic functions. This is verbose and a maintenance burden. Accordingly, it is a convenience to, in some circumstances, allow a C function to be used as a logic function.

The key criterion for a C function to be allowed as a logic function is that the function be *pure*, that is, that it have no side effects. Purity is judged by the conservative criterion that no assignments are made to any non-local variable; any functions that are called must also be pure.

Purity is indicated by including the keyword `pure` in the function specification. The intention to allow the function to be used in logic expressions is indicated by including the keyword `logic` in the function specification.

If a function is *pure*, it is an error for its specification to include any `assigns` clause other than `assigns \empty;` or `assigns \nothing;`. If a behavior does not include an explicit `assigns` clause, the implicit default is `assigns \nothing`.

Purity is inherited. If a *pure* function is virtual and overridden by a function in a derived class, the overriding function must also be *pure*.

Note that such functions may have an implicit argument: the `this` pointer to the class object.

An example is shown in the following listing: (TODO fix visibility)

```
class C {
  private:
    int value;

  public:

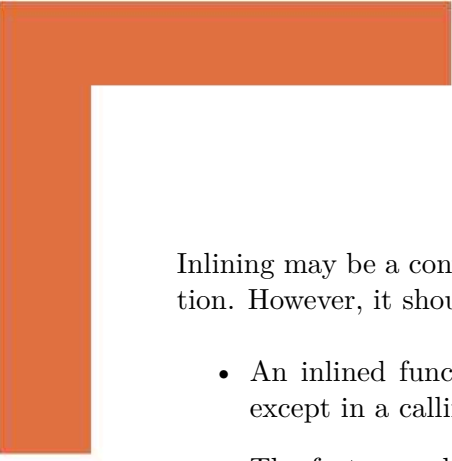
    /*@ ensures \result == this->value;
       @ pure logic
       @*/
    int value() {
      return value;
    }
}
```

0.6 Inline functions

EXPERIMENTAL

A second way to reduce verboseness and duplication between code and specifications is to use inlining. In this case the body of a function is used as its specification. When the function is called from elsewhere, an analysis tool will effectively inline the body of the function at the call site (with appropriate replacement of formal arguments with actual arguments).

When inlining is desired, a function has no specification other than the keyword `inline`, except perhaps also `pure` and `logic`.



Inlining may be a convenience; tools may even use it by default if there is no other specification. However, it should be used with caution:

- An inlined function has no specification to be validated against its implementation, except in a calling context.
- The feature reduces modularity, because now a calling function depends on the body of a called function.
- It increases the effective size of the body of a calling function for analysis purposes.

0.7 Enums [C++]

«Needs grammar additions»

TODO

C++ introduced enum types that are more strongly checked than in C. C++ allows the declaration of *unscoped* and *scoped* enum types. In either case, the underlying type of the enum is some integral type. Accordingly, C++ still allows some conversions between enum values and integral values. Unscoped enumerations have an implicit conversion to an int, as in,

```
enum color { red, orange, yellow}; // Unscoped type,
                                   // because there is no 'struct' or 'class' keyword
color c1 = red;
color c2 = 1; // Illegal in C++
color c3 = (color)10; // Cast is permitted
int j = color; // Allowed for unscoped enum types
```

However, scoped enumeration values are not implicitly cast to integral types:

```
enum struct color { red, orange, yellow}; // Scoped type,
                                           // because of the 'struct' or 'class' keyword
color c1 = red;
color c2 = 1; // Illegal in C++
color c3 = (color)10; // Cast is permitted
int j = color; // Forbidden for scoped enum types
int k = (int)color; // Cast is permitted
```

Note that in C++ an integral value may be cast to an enum value even if the value does not correspond to one of the named enumerator values, as long as it falls within the implementation range of the enumeration.

In some situations, however, the author would like an enum type to be strictly distinguished from integral types in one or both of two respects:

- To forbid any conversions to or from integral values, even with casts.
- To forbid creating any enumerator values that are not named in the enumeration declaration.

ACSL++ allows two modifiers on an enum declaration: **strong** and **complete**. If these are present on any declaration or definition of an enum, they must be present on all declarations and definitions of the enumeration type.

- **complete** means that no integral values are permitted that do not correspond to a named enumerator. A declaration of **complete** means that any conversion from an integral value to an enum value creates a proof obligation that the integral value belongs to one of the declared enumerators.
- **strong** means that the enum is strongly typed, that is, that no conversions to or from integral values are permitted. This check can be performed during type checking and does not create any *proof obligations*. A **strong enum forbids** any conversion from an integral value so it is also necessarily complete.

0.8 C++ types and casting [C++]

0.8.1 Logic types

ACSL already allows the definition of logic types (cf. §??). One can declare a possibly polymorphic type name and then declare its properties by axiomatization. Alternatively, one can declare a type name and give a concrete definition as either a record or sum type.

These capabilities are still present in ACSL++, but note that such types are strictly logic types; there is no inheritance relationship or dynamic dispatching of method calls.

EXPERIMENTAL

An additional means of defining types is to declare C++ class (or struct) types as logic types. The syntax for such a definition is simply to preface the declaration with the keyword **ghost** and write it within an ACSL annotation. Such a ghost class is considered a kind of logic type.

Consistent with other rules that maintain a separation between logic entities and C entities,

- ghost classes may not appear in non-ghost code
- a non-ghost class may not inherit from a ghost class
- the fields of ghost classes may only use logic types
- any implementations given for methods of ghost classes may only use logic types and may only write to ghost memory locations
- ghost classes may contain ACSL++ annotations in the same way that C classes do, with the syntax adjusted for being written within ghost code

As with C++ classes, ghost classes may inherit fields and methods from their base classes and method calls are subject to dynamic dispatch.

This concept is still experimental because use cases are still be explored that would require this feature.

0.8.2 Static type information

Need grammar additions and examples

C++ includes the `type_traits` template to determine information about types. For example, for a type name `T`, one can write `std::is_fundamental<T>::value` or `std::is_fundamental<T>()`, obtaining a `bool` value indicating whether `T` is a fundamental type or not.

This is particularly useful for template type names.

In addition, the `value` field of each of these templates is a named C++ compile-time constant; they can be used in logic expressions and their values do not depend on state.

No new features are needed in ACSL++ to support static type information.

0.8.3 Casting

Need grammar additions and examples

C++ added to C a number of casting operations. Most of these may be used in logic expressions.

- **(T)e** and **T(e)** casts – When applied to primitive types, these casts perform an arithmetic conversion. Aside from potential numeric overflow errors, these are appropriate to use in logic expressions. When applied to pointers and reference expressions, these casts are equivalent to `reinterpret_casts`. The cast works fine for conversions to and from `void*` and certain pairs of related types, but it is better to use one of the C++ casts described in the following bullets. Thus the use of these C- and function-style casts on pointers and references in logic expressions is deprecated (in ACSL++) in favor of the C++-style casts.
- **dynamic_cast<T>(e)** – A dynamic cast is used to convert a value of pointer or reference type to the corresponding value of another pointer or reference type, typically within the first type's inheritance hierarchy. For pointer types, if the given expression `e` is not a pointer to a complete instance of the desired type `T`, then the result is `null`. For reference types, if the desired conversion is not allowed, a `bad_cast` exception is thrown. Thus this cast operator can be used to test type relationships.

When used in a logic expression, the semantics of a `dynamic_cast` operation applied to a pointer value is the same as in C, producing either a valid new pointer or a null value. When applied to a reference value however, if the operation would produce an exception, it instead produces an undefined value.²

- **static_cast<T>(e)** – A static cast also operates on pointer or reference values, but simply returns the value of the given expression with the given static type. If the conversion is valid, the result is the same as dynamic casting. If the result is invalid, the result is undefined and may result in runtime errors later in program execution. ACSL++ tools may check whether a static cast produces a valid value.

The semantics of `static_cast` in a logic expression is that it produces the expected value for valid conversions and is undefined for invalid conversions.

- **reinterpret_cast<T>(e)** – A reinterpret cast simply maintains the same underlying bits, but interprets them as the given type. Whether this produces valid results and what the results are is likely to be platform-dependent. Accordingly, `reinterpret_casts` are not permitted in logic expressions.
- **const_cast<T>(e)** – A const cast simply restates the type of the expression without const qualification. It may be used in a logic expression when needed for type matching. However, since logic expressions do not have side effects, no functions that actually modify non-const values can be used.³

²This is a case where the operation could be considered to be partial and ill-defined for a bad conversion. However, since the functions of ACSL++'s logic are in general total, the semantics of `dynamic_cast` is defined to be total as well.

³Even in C++, using a const cast to enable modifying another const value produces undefined results.

0.8.4 Dynamic type information

Grammar additions and examples, more about `hash_code` and `type_index` or omit them.

C++ also has means to obtain information about the dynamic type of an expression using the `typeid` operator. It can be applied to either a type name or an expression; it can be used (through `hash_code` or `type_index`) to check equality of types.

The `typeid` operator and the `hash_code` and `type_index` may all be used in logic expressions. They are applied to C++ objects, not to logic values. Example uses of these features are shown below.

```
1  #include <iostream>
2  #include <typeinfo>
3  class B {
4      public :
5          virtual const std::string who() { return "B"; }
6  };
7
8  class A: public B {
9      public:
10         virtual const std::string who() {return "A"; }
11     };
12     int main() {
13         B* b = new B();
14         B* bb = new B();
15         A* a = new A();
16         B* ba = new A();
17         const std::type_info& bt = typeid(*b); // *b, NOT b !!
18         const std::type_info& bbt = typeid(*bb);
19         const std::type_info& at = typeid(*a);
20         const std::type_info& bat = typeid(*ba); // type_info of the dynamic type
21
22         std::cout << b->who() << bb->who() << a->who() << ba->who() << std::endl;
23         std::cout << bt.name() << " " << bbt.name() << " "
24                 << at.name() << " " << bat.name() << std::endl;
25         // All comparisons are true
26         std::cout << (bt==bbt) << " " << (bt!=at) << " " << (bt!=bat)
27                 << " " << (at==bat) << std::endl;
28         return 0;
29     }
```

No additional features are needed in ACSL++ to reason about such code. What is needed is the ability to reason about the dynamic type of a given pointer or reference to an object. For this, the logical encoding of a C++ program does need a representation of C++ types and their inheritance relationships.

Such capability is additionally needed to reason about uses of `dynamic_cast`.

0.8.5 Implicit and explicit conversions

C++ defines a complex set of rules for determining implicit conversions to be applied. Besides the casts discussed in §??, there are two other conversions: the one-argument constructor and operator `T()` conversion.

These conversions do not require any additional features within ACSL++. If the program defines either a one-argument constructor or a operator `T()` conversion, then a specification

for that function should also be given, so that any ACSL++ tools can appropriately analyze the code.

These conversions may be used in logic expressions, either implicitly or explicitly, only if they are permitted by the rules of §0.11.4.

Examples need

0.9 Invariants

«Invariants have tricky semantics – when do they hold, which are applicable in a given situation, etc. – may need more discussion of this here»

0.10 Functional programming in C++

«Discuss how to specify lambda expressions and fluent programming constructs as implemented in C++»

TODO

0.11 Access control and abstraction [C++]

«Grammar additions needed»

0.11.1 Access control of ACSL++ constructs

C++ allows members of aggregate types to be declared as `public`, `protected`, or `private`. This visibility modifier affects whether the member can be accessed within other contexts.

The same access rules apply to ACSL++ constructs declared at class scope. Any name that may be declared at class scope, such as logic functions, predicates, invariants and ghost code, has associated with it the access modifier that it would if it were a C++ construct, with the same access defaults being used for struct and class declarations. Typically, `private` ACSL++ constructs would be used to aid in the proofs of the implementation of a class's members, whereas `public` ACSL++ constructs are used as class abstractions or to assist in proofs of functions that use the class's functionality.

Recall that a C++ context A can declare as a `friend` a name N from context B and thereby allow the definition of B::N to use private elements of A. Similarly, within an ACSL++ annotation, context A can declare as `friend` an ACSL++ name N from context B and thereby allow the definition of B::N (within an ACSL annotation) to use private elements of A.

«Need an example of access and motivating examples of the rule about friends.»

0.11.2 Access, specifications and information hiding

If a particular C++ name is accessible and that name has an associated ACSL++ specification, then the specification is also accessible and all the names used within the specification must be accessible. For example, if a function definition is accessible in some context, then the function contract is also accessible, so all the names used within that function contract must be accessible. This is a conceptual requirement: if a client of a class uses a function of the class, the client needs to be able to see the specification of that function, and thus it must have access to all the identifiers within the expressions of that specification.

However, this naturally expressed requirement causes problems in writing specifications that adhere to the information hiding expectations of classes. Consider a class that wraps an implementation of a simple array:

```
class Array {  
    private:  
        int [] data;  
        int data_length;  
  
    public:  
  
        /*@  
        requires 0 <= index < data_length;  
        ensures  \result == data[data_length];  
        */  
        int getValue(int index) {  
            return data[index];  
        }  
}
```

```
| }
```

Here the representation of the Array object, its data values and length, are appropriately private, while the accessor function is public. However, expressing the specification of the accessor function requires using the private members, violating information hiding and the abstraction boundary, as well as the access rule stated above.

The core problem is this: to express the valid values of `index`, one needs the concept of the length of the array; that is, one needs an abstraction, namely `length()`. In this example the abstract `length()` is represented by the data value `data_length`, but the class could have chosen some other representation.

One solution to this representation problem is to write the following:

```
class Array {
    private:
        int [] data;
        int data_length;

    public:
        /*@
         requires true;
         logic int length() = this.data_length;

         requires 0 <= index < length();
         logic int value(int index) = this.data[index];
        */

        /*@
         requires 0 <= index < length();
         ensures \result == value(index);
        */
        int getValue(int index) {
            return data[index];
        }
}
```

This example demonstrates the general solution for abstraction that respects information hiding: write public logic functions, predicates or invariants that express the abstract properties of the class, express the specifications of the public members of the class in terms of those (public) abstractions, and write the implementation of the abstractions in terms of the appropriate private aspects of the class.

0.11.3 Usability improvements

In the example above we abstracted the concept of array length by defining the public logic function `length()`. In this case, the sole purpose of `length()` is to make accessible the value of `data_length`. It may seem verbose and overkill to need to define and use `length()` solely to expose `data_length`. An alternative is to declare that although `data_length` is private for C++ implementation purposes, it is public for specification purposes. In this case we write

```
| class Array {
```

```

private:

    //@ spec_public
    int [] data;

    //@ spec_public
    int data_length;

public:

    /*@
       requires 0 <= index < this.data_length;
       logic int value(int index) = this.data[index];
    */

    /*@
       requires 0 <= index < data_length;
       ensures \result == this.value(index);
    */
    int getValue(int index) {
        return data[index];
    }
}

```

The `spec_public` access specifier applies only to the next declaration within the class declaration: in the example above both `data` and `data_length` are still `private` in C++, since that is the most recent C++ access specifier, but the declarations have public access within specifications.

The advantage of this syntax is more concise expression; there is no need to declare `length()` just to replicate `data_length`. The disadvantage is that we have lost a measure of information hiding. By using the abstraction `length()` we could have altered the representation of length within the class without affecting the public specifications and what clients of the class see. By using `spec_public`, that is no longer the case. Accordingly, `spec_public` should only be used in temporary situations, during development, or where it is clear that the representation is not going to change.

The access specifier `spec_protected` may be used (within ACSL++ annotations) to give C++ protected access to C++ constructs that would otherwise be `private`.

0.11.4 Pure functions

A second verbose aspect of the listing above is that the logic representation function `value()` is essentially a replica of the C++ function `getValue()`. Why write and maintain both of these? Can we not use at least some C++ functions in specifications?

The syntax to do so is shown in the following listing, but there are restrictions on when this is allowed.

```

class Array {

    private:

        int [] data;

```



```

    //@ spec_public

    int data_length;

public:

    /*@
      requires 0 <= index < data_length;
      ensures  \result == data[index];
    */
    //@ pure
    int getValue(int index) {
        return data[index];
    }
}

```

When designated `pure`, a C++ function may also be used in a ACSL++ specification. The conditions under which a C++ function may be declared `pure` are discussed in the following subsections.

No side-effects

The principal restriction is that the function declared `pure` must have no side-effects. It may compute a value and may have temporary stack variables, but it may not assign to any locations in the heap that are allocated in the pre-state of the function. No assignments are permitted to a pre-state location even if the assignment does not change the value stored in the location. This is partly a static condition, but when assigning to fields of an object, it must be demonstrated that the object is a local temporary or has been allocated since the pre-state; this demonstration may require reasoning about aliasing.

No heap changes

The rule prohibiting side-effects applies to the heap as well: no allocations or deallocations of memory are allowed in pure functions. Stack operations are allowed. Thus, for example, implicit copy constructors that only create and destroy stack temporaries are permitted.

0.11.5 Reading the heap

Since the pure function is indeed a C++ function, it may include read operations on the heap. When the pure function is used as a logic function, it must be invoked with respect to a given heap state. That is, the pure function has a single implicit state label. An example showing this use is given below:

```

1  class C {
2  private:
3      //@ spec_public
4      int count;
5
6  public:
7      /*@ ensures \result == count;
8         pure
9         @*/
10     int getCount() { return count; }
11

```

```
12 |
13 | void test() {
14 |     x:
15 |     count++;
16 |     //assert getCount{Here}() == 1 + getCount{x}();
17 | }
```

Is the rule concerning strong purity too draconian?

0.12 Templates [C++]

0.12.1 Class templates

C++ includes syntax for parameterized template classes and functions. ACSL++ constructs can occur within template classes, just as they can appear within non-template classes. However, note that ACSL++ (like ACSL) has its own syntax for declaring parameterized (polymorphic) types, so no additional capabilities are needed within ACSL++ to allow parameterized logic types, functions, or predicates.

Naively, the consequence of an ACSL++ declaration being within a template is that some additional names are defined in the scope in which the ACSL++ declaration is declared. In particular, most commonly there are additional type names, but there can also be values of other types. Any ACSL++ contracts within the template can use the template parameters, just like any C++ construct can.

However, templates have a complication for both C++ and ACSL++: until template parameters are given actual values (that is, until the template is instantiated) it cannot always be determined whether the code or specifications within the template are well-formed. This is because the declared names and the types of names and operations that are dependent on the template parameters cannot be resolved until the actual template parameters are known. If T is a template parameter, $T::a$ is not known to be syntactically valid, or, if it is, what its type is, until the value of T itself is known.

Therefore a library template cannot be fully verified as a parameterized class; only verifications of instantiations are possible.

0.12.2 Specifications of templates [C++]

Even if a parameterized template cannot be verified until it is instantiated, the template text must still contain its specification. That specification must be written using names and operations that are in scope within the template specification, even if they are instantiated along with the template itself.

TODO: Exam

0.12.3 Function templates [C++]

C++ also allows declaring function templates, either as global functions or as members of aggregate classes. The corresponding capability is already present in ACSL++ as logic predicates and functions that have polymorphic type arguments.

TODO: Exam

0.13 Specification abstraction [C++]

«What capabilities are needed to write logic abstractions in a base class and then tie them to implementation in derived classes?»

0.14 C++ types and casting [C++]

0.14.1 Logic types

ACSL already allows the definition of logic types (cf. §??). One can declare a possibly polymorphic type name and then declare its properties by axiomatization. Alternatively, one can declare a type name and give a concrete definition as either a record or sum type.

These capabilities are still present in ACSL++, but note that such types are strictly logic types; there is no inheritance relationship or dynamic dispatching of method calls.

EXPERIMENTAL

An additional means of defining types is to declare C++ class (or struct) types as logic types. The syntax for such a definition is simply to preface the declaration with the keyword **ghost** and write it within an ACSL annotation. Such a ghost class is considered a kind of logic type.

Consistent with other rules that maintain a separation between logic entities and C entities,

- ghost classes may not appear in non-ghost code
- a non-ghost class may not inherit from a ghost class
- the fields of ghost classes may only use logic types
- any implementations given for methods of ghost classes may only use logic types and may only write to ghost memory locations
- ghost classes may contain ACSL++ annotations in the same way that C classes do, with the syntax adjusted for being written within ghost code

As with C++ classes, ghost classes may inherit fields and methods from their base classes and method calls are subject to dynamic dispatch.

This concept is still experimental because use cases are still be explored that would require this feature.

0.14.2 Static type information

Need grammar additions and examples

C++ includes the `type_traits` template to determine information about types. For example, for a type name `T`, one can write `std::is_fundamental<T>::value` or `std::is_fundamental<T>()`, obtaining a `bool` value indicating whether `T` is a fundamental type or not.

This is particularly useful for template type names.

In addition, the `value` field of each of these templates is a named C++ compile-time constant; they can be used in logic expressions and their values do not depend on state.

No new features are needed in ACSL++ to support static type information.

0.14.3 Casting

Need grammar additions and examples

C++ added to C a number of casting operations. Most of these may be used in logic expressions.

- **(T)e** and **T(e)** casts – When applied to primitive types, these casts perform an arithmetic conversion. Aside from potential numeric overflow errors, these are appropriate to use in logic expressions. When applied to pointers and reference expressions, these casts are equivalent to `reinterpret_casts`. The cast works fine for conversions to and from `void*` and certain pairs of related types, but it is better to use one of the C++ casts described in the following bullets. Thus the use of these C- and function-style casts on pointers and references in logic expressions is deprecated (in ACSL++) in favor of the C++-style casts.
- **dynamic_cast<T>(e)** – A dynamic cast is used to convert a value of pointer or reference type to the corresponding value of another pointer or reference type, typically within the first type’s inheritance hierarchy. For pointer types, if the given expression `e` is not a pointer to a complete instance of the desired type `T`, then the result is `null`. For reference types, if the desired conversion is not allowed, a `bad_cast` exception is thrown. Thus this cast operator can be used to test type relationships.

When used in a logic expression, the semantics of a `dynamic_cast` operation applied to a pointer value is the same as in C, producing either a valid new pointer or a null value. When applied to a reference value however, if the operation would produce an exception, it instead produces an undefined value.⁴

- **static_cast<T>(e)** – A static cast also operates on pointer or reference values, but simply returns the value of the given expression with the given static type. If the conversion is valid, the result is the same as dynamic casting. If the result is invalid, the result is undefined and may result in runtime errors later in program execution. ACSL++ tools may check whether a static cast produces a valid value.

The semantics of `static_cast` in a logic expression is that it produces the expected value for valid conversions and is undefined for invalid conversions.

- **reinterpret_cast<T>(e)** – A reinterpret cast simply maintains the same underlying bits, but interprets them as the given type. Whether this produces valid results and what the results are is likely to be platform-dependent. Accordingly, `reinterpret_casts` are not permitted in logic expressions.
- **const_cast<T>(e)** – A const cast simply restates the type of the expression without const qualification. It may be used in a logic expression when needed for type matching. However, since logic expressions do not have side effects, no functions that actually modify non-const values can be used.⁵

⁴This is a case where the operation could be considered to be partial and ill-defined for a bad conversion. However, since the functions of ACSL++’s logic are in general total, the semantics of `dynamic_cast` is defined to be total as well.

⁵Even in C++, using a const cast to enable modifying another const value produces undefined results.

0.14.4 Dynamic type information

Grammar additions and examples, more about `hash_code` and `type_index` or omit them.

C++ also has means to obtain information about the dynamic type of an expression using the `typeid` operator. It can be applied to either a type name or an expression; it can be used (through `hash_code` or `type_index`) to check equality of types.

The `typeid` operator and the `hash_code` and `type_index` may all be used in logic expressions. They are applied to C++ objects, not to logic values. Example uses of these features are shown below.

```

1  #include <iostream>
2  #include <typeinfo>
3  class B {
4      public :
5          virtual const std::string who() { return "B"; }
6  };
7
8  class A: public B {
9      public:
10         virtual const std::string who() {return "A"; }
11     };
12     int main() {
13         B* b = new B();
14         B* bb = new B();
15         A* a = new A();
16         B* ba = new A();
17         const std::type_info& bt = typeid(*b); // *b, NOT b !!
18         const std::type_info& bbt = typeid(*bb);
19         const std::type_info& at = typeid(*a);
20         const std::type_info& bat = typeid(*ba); // type_info of the dynamic type
21
22         std::cout << b->who() << bb->who() << a->who() << ba->who() << std::endl;
23         std::cout << bt.name() << " " << bbt.name() << " "
24                 << at.name() << " " << bat.name() << std::endl;
25         // All comparisons are true
26         std::cout << (bt==bbt) << " " << (bt!=at) << " " << (bt!=bat)
27                 << " " << (at==bat) << std::endl;
28         return 0;
29     }

```

No additional features are needed in ACSL++ to reason about such code. What is needed is the ability to reason about the dynamic type of a given pointer or reference to an object. For this, the logical encoding of a C++ program does need a representation of C++ types and their inheritance relationships.

Such capability is additionally needed to reason about uses of `dynamic_cast`.

0.14.5 Implicit and explicit conversions

C++ defines a complex set of rules for determining implicit conversions to be applied. Besides the casts discussed in §??, there are two other conversions: the one-argument constructor and `operator T()` conversion.

These conversions do not require any additional features within ACSL++. If the program defines either a one-argument constructor or a `operator T()` conversion, then a specification

for that function should also be given, so that any ACSL++ tools can appropriately analyze the code.

These conversions may be used in logic expressions, either implicitly or explicitly, only if they are permitted by the rules of §0.11.4.

Chapter 1

Examples

Consider the application of the transform operation to a unary operator with side effects.

```
int sum = 0;
auto op = [](int i) mutable { sum += i; return -i; }
vector<int> a = ...
vector<int> b = ...

transform(a.begin(), a.end(), b.begin(), op)
```

Functors are applied using their `operator()` member function. In general we do not statically know what functor is being used as the actual argument. So we need to be able to express some specification properties.

- $\text{\textbackslash pre}(f,i)$ – a predicate that is true when the precondition of $f(i)$ is true
- $\text{\textbackslash post}(f,r,i)$ – a predicate that is true when the postcondition of $f(i)$ with result value r is true
- $\text{\textbackslash assigns}(f,i) - \text{\textbackslash locset}$ that contains any memory location assigned when $f(i)$ is called with argument i

So for the value `op` in the code above.

- $\text{\textbackslash pre}(op,i) == \text{\textbackslash true}$ (ignoring any overflow checks)
- $\text{\textbackslash post}(op,r,i) == (r == -i \ \&\& \ \text{sum} == \text{\textbackslash old}(\text{sum} + i))$
- $\text{assigns}(op,i) == \{ \text{sum} \}$

Now `transform` applies `op` repeatedly to the elements of `vector`. If `op` were a simple function of its input only with no side effects, we could write a postcondition easily as something like $(\text{\textbackslash forall} \text{ int } i; 0 \leq i \leq a.size(); \text{vector}[i] == op(\text{\textbackslash old}(\text{vector}[i])))$.

However, `op` may easily be more complex than that. It may have more complex inputs and it may have side-effects. In general we cannot know the effect of repeated application of `op` without simulating the loop implicit in `transform` and reasoning about it with inductive invariants.

The specification of `transform` could look like this:

```

/*@ behavior
requires  \valid (first1,last1) && \valid(result + (0.. last1-first1));
{
    InputIterator in = first1;
    OutputIterator out = result;
    /*@
    loop_invariant 0 <= \count <= last1-first1;;
    loop_invariant in == first1 + \count;
    loop_invariant out = result + \count;
    loop_modifies out[0 .. last1-first1-1];
    loop_decreases (last1 - first1) - \count;
    while (in != last1) {
        ... effect of op ...
        ++in;
        ++out;
    }
}

@*/
template <class InputIterator, class OutputIterator, class UnaryOperator>
OutputIterator transform (InputIterator first1, InputIterator last1,
OutputIterator result, UnaryOperator op);

```

Here the block within the specification, enclosed in curly braces, is a *model program*. A model program specifies the behavior of some method by writing abstracted code (pseudo-code, if you will) that summarizes the method's behavior. For small programs, this might simply nearly replicate the actual implementation. In that case, reasoning about uses of the method are tantamount to inlining the implementation.

The implicit (ghost) variable `\count` denotes the number of completed iterations of the loop, beginning with 0. For a simple loop index that counts up by 1 from 0, the loop index is equal to `\count`. For more complex loops and loops using iterators, `\count` is useful in writing specifications.

Now we need to include the effects of `op` in the above specification of `transform` without knowing what `op` actually is. We do so in terms of its own specification.

A snippet of code like `y = op(x);` can be summarized as

```

assert  \pre(op,x);
havoc  \assigns(op,x);
int temp_result;
assume \post(op,temp_result,x);
y = temp_result;

```

Putting that snippet together with the partial specification of `transform` gives us this:

```

/*@ behavior
requires  \valid (first1,last1) && \valid(result,result + (0.. last1-first1-1));
{
    InputIterator in = first1;
    OutputIterator out = result;
    /*@
    loop_invariant 0 <= \count <= last1-first1;;
    loop_invariant in == first1 + \count;
    loop_invariant out = result + \count;
    loop_modifies out[0 .. last1-first1-1], \union(int i = 0 .. \count-1; \assigns(op,in[i],out[i]));
    loop_invariant

```

```

        loop_decreases (last1 - first1) - \count;
        while (in != last1) {
            int temp_arg = *in;
            assert \pre(op,temp_arg);
            havoc \assigns(op,temp_arg);
            int temp_result;
            assume \post(op,temp_result,temp_arg);
            *out = temp_result;
            ++in;
            ++out;
        }
    }

    @*/
    template <class InputIterator, class OutputIterator, class UnaryOperator>
    OutputIterator transform (InputIterator first1, InputIterator last1,
        OutputIterator result, UnaryOperator op);

```

Although, not essential in this case, in general we separate subexpressions by evaluating each subexpression to a temporary intermediate result. That way any implicit conversions and questions of well-definedness can be made explicit with appropriate assertions.

1.1 Misc stuff

1.2 Other topics

- inline functions - repeated specs