3-4-1999

Larch/C++ Reference Manual

Gary T. Leavens

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Software Engineering Commons, and the Theory and Algorithms Commons

Recommended Citation

http://lib.dr.iastate.edu/cs_techreports/357

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
Chapter 1: Introduction

1 Introduction

Larch/C++ is a notation for formally specifying the behavior and interfaces of C++ classes and functions. C++ is the programming language defined in Stroustrup’s book [Stroustrup91] and more fully in Ellis and Stroustrup’s book [Ellis-Stroustrup90] and the draft C++ standard [ANSI95]. These contain the reference manual for C++. We will often refer to sections of the reference manual by citing one of these. (These sections are roughly correlated: Section r.7.1 in [Stroustrup91] is the same as Section 7.1 in [Ellis-Stroustrup90].)

The goal of this reference manual is to precisely record the design of Larch/C++. We try to give examples and explanations, and we hope that these will be helpful to readers trying to learn about formal specification using Larch/C++. However, this manual is not designed to give all the background needed to write Larch/C++ specifications, nor to give the prospective user an overview of a useful subset of the language. For this background, we recommend the following. The reader new to formal specification or new to the Larch approach to behavioral interface specification is advised to read chapters 1-4 of [Guttag-Horning93] first. Such a reader might also want to consult [Leavens98], where more tutorial material on Larch can be found. Once the reader has the necessary background, an overview of a useful subset of Larch/C++ can be found in the paper [Leavens96]. There is also a “poster” that ships with the release. (A useful, but somewhat obsolete overview can also be found in [Cheon-Leavens94].)

Readers with the necessary background, and users wanting more details may, we hope, profit from reading this manual. We suggest reading in this manual starting with chapters 1-3, skimming chapter 4 quickly, skimming chapter 5 to get the idea of what declarations mean in Larch/C++ (paying a bit more attention in section 5.4 to the sorts of declarations), and then reading the chapters on function and class specifications, paying particular attention to the examples. After that, one can use the rest of this manual as a reference.

1 See Appendix B [Bibliography], page 319, for details on references to the literature, cited in this manual.

2 Currently the semantics are stated informally. We will eventually give a more formal semantics to Larch/C++, but as with any large design project, there are benefits to sketching the design informally first. This is similar to the idea of designing a program before coding it. In the case of Larch/C++, the size and complexity of C++ make this approach imperative.
1.1 Larch-style Specifications

Larch/C++ is a Larch-style behavioral interface specification language [Guttag-Horning93]. In this style of specification, which might be called model-oriented [Wing90a], one specifies both the interface of a procedure or abstract data type (see Section 2.2 [Interfaces], page 15), and its behavior. The behavior of a procedure is specified by describing: for what states the procedure is defined, what the procedure is allowed to change, and the relation between the states before and after the procedure is invoked. The states for which the procedure is defined is described by a logical assertion, called the precondition; the allowed relationships between these states and the states that may result from a call are described with another logical assertion called the postcondition. The pre- and postconditions are expressed using a mathematical vocabulary, which has a formal meaning specified, in part by the user of Larch/C++, in the Larch Shared Language (LSL) [Guttag-Horning93].

The behavior of an abstract data type (ADT), which is implemented by a class in C++, is specified by describing a set of abstract values for its objects, and by specifying each of its operations (C++ member functions) as a procedure. (Unlike other specification languages, such as Z [Hayes93] [Spivey92], or Fresco [Wills92b] Larch/C++ does not require the user to model an instance as a collection of data members. However, that is allowed and, as will be explained below, is the default if the user does not specify some other model.)

For example, consider the following Larch/C++ specification of a simple C++ class IntHeap. This file could be used as a header file in a C++ program. (An explanation of the notation follows the specification.)

```cpp
//@ uses IntHeapTrait; // 3 connection to LSL
//@ abstract @*/
class IntHeap { // 5 no constructors, C++ interface
public: // 6
virtual int largest() const throw() = 0; // 7 C++ interface
//@ behavior { // 8 starts largest’s specification
//@ requires len(self^) >= 1; // 9 precondition
//@ modifies nothing; // 10 what can change
//@ ensures result = head(self^); // 11 postcondition
//@ }
};
```
In C++, comments start with either //, and extend to the end of a line, or they start with /* and extend to the next */. In this specification the forbidding first line is a comment that is part of a version control system, and should be ignored. If, however, the first character of what looks to C++ like a comment is the at-sign character (@), then the comment start sequence, and the at-sign are all dropped by Larch/C++. For example, on line 3, the characters //@ are dropped by Larch/C++. Thus, when this line is used in a C++ program, the keyword uses is considered to be part of a comment, but it is significant when passed to Larch/C++. The comment sequences /*@ and @*/ are also treated in this way. Thus what Larch/C++ sees is the following input. (If one wishes to have specifications in separate files from header files, then one can also pass this form directly to Larch/C++.)

```cpp
//@ uses IntHeapTrait;  // 3 connection to LSL

abstract class IntHeap {  // 5 no constructors, C++ interface
public:  // 6
virtual int largest() const throw() = 0;  // 7 C++ interface
behavior {  // 8 starts largest's specification
  requires len(self^) >= 1;  // 9 precondition
  modifies nothing;  // 10 what can change
  ensures result = head(self^);  // 11 postconditon
}
};
```

Line 3 of this specification says that abstract values of IntHeap objects are specified in the LSL trait named IntHeapTrait; this gives a mathematical vocabulary for talking about such values. Line 5 of this specification states that the class being specified is an abstract class; that is, that it is a class for which no constructors are provided. This line is also part of the class’s interface; it gives the name of the C++ class that implements this specification: IntHeap. Line 7 gives the C++ interface of a member function. Note that this includes C++ details such as that the member function is virtual and const, and that it throws no exceptions. (The C++ syntax = 0 says that derived classes must implement this function, see Section 7.10 [Abstract Classes], page 234 for details.) Line 8 starts the specification of the behavior of the member function largest. Line 9 gives the precondition for the member function, and line 11 the postcondition; both of these are written using the vocabulary of IntHeapTrait, which gives a meaning to the trait functions len and head (and =). Line 10 specifies that the implementation of this member function cannot change any variables; in particular, the value of the default parameter, *this, cannot be changed.
LSL comes with a wide variety of useful built-in traits, which form a so-called LSL “Handbook” (see [Guttag-Horning93], appendix A). The abstract values of \texttt{IntHeap} can thus be described by including the trait \texttt{PriorityQueue} (found on page 175 of [Guttag-Horning93]) and renaming its sorts. The other trait included, \texttt{NoContainedObjects}, says that the abstract values of \texttt{IntHeap} objects do not contain embedded objects. See Section 7.5 [Contained Objects], page 207 for details of the trait \texttt{NoContainedObjects}. In more realistic examples, writing the trait might not be so easy.

The Larch style of specification goes back to Hoare’s work. Hoare used pre- and postconditions to describe the semantics of computer programs in his famous article [Hoare69]. Later Hoare adapted these axiomatic techniques to the specification and correctness proofs of abstract data types [Hoare72a]. To specify an abstract data type, Hoare described a set of abstract values for the type, and then specified pre- and postconditions for each of the operations of the type in terms of how the abstract values of objects were affected. For example, one might specify the type \texttt{IntHeap} using abstract values of the form \texttt{empty} and \texttt{add(i,h)}, where \texttt{i} is an \texttt{int} and \texttt{h} is an \texttt{IntHeap}. (The detailed description of such a type’s abstract values, could be recorded in an LSL trait, such as \texttt{IntHeapTrait}.)

There are three advantages to using abstract values instead of directly using C++ variables and data structures. The first is that such a specification is simpler, because it has fewer details. This makes specifications easier to read and understand, provided that one is comfortable with the mathematical vocabulary used to describe the abstraction. The second, and more important, reason is that by using a mathematical abstraction to specify behavior, the specification does not have to be changed when the particular data structure used in the program is changed. This permits different implementations of the same specification to use different data structures. Therefore the specification forms a contract between the rest of the program in the implementation, which ensures that the rest of the program is also independent of the particular data structures used [Parnas72] [Liskov-Guttag86] [Meyer88] [Meyer92]. Finally, it allows the specification to be written even when there are no implementation data structures, as is the case for \texttt{IntHeap}.

This idea of model-oriented specification has been followed in VDM [Jones86b], Z [Hayes93] [Spivey92], and Larch [Guttag-Horning93]. The essential elaboration of Hoare’s original idea is that the abstract values also come with a set of operators, which we will refer to as \textit{trait functions}, to avoid confusion with C++ operators. The trait functions are used to precisely describe the set of
abstract values. In Z one builds abstract values using tuples, sets, relations, functions, sequences, and bags; these all come with pre-defined trait functions that can be used in assertions. In VDM one has a similar collection of mathematical tools to describe abstract values, and another set of pre-defined trait functions. In the Larch approach, there are some pre-defined kinds of abstract values (found in Guttag and Horning’s LSL Handbook, Appendix A of [Guttag-Horning93]), but these are expected to be extended as needed. (The advantage of being able to extend the mathematical vocabulary is similar to one advantage of object-oriented programming: one can use a vocabulary that is close to the way one thinks about a problem.)

The extension mechanism in LSL is called a trait. A trait is like an equational specification, augmented with some additional constructs. These additional constructs allow one to state specific kinds of induction and deduction principles for reasoning about abstract values. (See section 4.2 in [Guttag-Horning93] for details.)

For historical reasons, the types in a trait are called sorts. Each abstract value thus has a sort. The trait specifies the sorts of arguments that each trait function takes, and the sort of its result. The axioms of a trait specify what can be proven about an expression involving trait functions. Each well-formed expression built from the trait functions has a sort, which is the result sort of the outermost trait function. The abstract values of a given sort can be thought of as equivalence classes of expressions having that sort. (One can use a partitioned by clause in LSL to make this equivalence the same as observable equivalence if desired. See pages 38-39 in [Guttag-Horning93] for details.)

The division between the two layers (or tiers) of a Larch/C++ specification is thus as follows. The shared (or “bottom”) layer of a specification consists of LSL traits, which describe abstract values. (These traits could be shared among many specifications for different interface layers.) The abstract values are purely functional; that is, there is no concept of mutation, state, aliasing, storage, non-termination, or nondeterminism in a trait.

The interface layer consists of Larch/C++ specification modules, which specify pieces of C++ programs. This is where one deals with the concepts of mutation, state, aliasing, storage, termination, nondeterminism, finiteness, and so on.

The separation of these layers is enforced in Larch/C++. That is, C++ functions cannot be used to specify other C++ functions. (Also, of course, trait functions cannot be used in C++ programs.) See Chapter 7 [Class Specifications], page 167 for further discussion on this point.

The picture just described is accurate in theory. However, in practice, users do not always write their own traits when specifying a concrete class in Larch/C++. They need not always write such
a trait, since, by default, Larch/C++ will automatically construct a trait that models the abstract values of a class as a tuple of its data members (as objects). This default can be overridden by using a trait, such as `IntHeapTrait` that defines a sort with the same name as the class being defined, such as `IntHeap`. See Chapter 7 [Class Specifications], page 167 for several examples. See Section 5.4.4 [Structure and Class Declarations], page 69 for more details about the automatically-constructed abstract model.

1.2 What is Larch/C++ Good For?

Larch/C++ is a formal specification language tailored to C++. Its basic use is thus the formal specification of the behavior of C++ program modules. As it is a behavioral interface specification language, Larch/C++ specifies how to use such C++ program modules from within a C++ program; hence Larch/C++ is not designed for specifying the behavior of an entire program. So the question “what is Larch/C++ good for?” really boils down to the following question: what good is formal specification for C++ program modules?

The main benefit in using Larch/C++ is precise, unambiguous documentation of the behavior C++ program modules (functions, classes, etc.). A Larch/C++ specification can be a completely formal contract about an interface and its behavior. Because it is an interface specification, one can record all the C++ details about the interface, such as the parameter mechanisms, whether the function is virtual, const, etc.; if one used a specification language such as VDM-SL or Z, which is not tailored to C++, then one could not record such details of the interface, which could cause problems in code integration.

One can use Larch/C++ either before coding, or as documentation of the code. (The notation is indifferent to the methodological questions; designing before coding is recommended, but documentation after the fact is better than none.)

Reasons for formal documentation of interfaces and their behavior, using Larch/C++, include the following.

- One can ship the object code for a class library to customers, without sending the source code. Customers would have documentation that is precise, unambiguous, but not overly specific. Customers would not have the code, protecting proprietary rights. In addition, customers would not rely on details of the implementation of the library that they might otherwise glean from the code, easing the process of improving the code in future releases.
- One can use a formal specification to analyze certain properties of a design carefully or formally
In general, the act of formally specifying a program module has salutary effects on the quality of the design.

- One can use the Larch/C++ specification as an aid to careful reasoning about the correctness of code, or even for formal verification. (We are not claiming that Larch/C++ has progressed to the point where we have used it for formal verification, but in principle, nothing stops one from using Larch/C++ for formal verification. The Larch Prover, see Chapter 7 of [Guttag-Horning93], can be used as an aid in this process, as noted in [Guaspari-Marceau-Polak90].)

There is one additional benefit from using Larch/C++. It is that Larch/C++ allows one to record not just public interfaces and behavior, but also some detailed design decisions. That is, in Larch/C++, one can specify not just the public interface of a C++ class (see Section 2.2 [Interfaces], page 15), but also behavior of a class’s protected and private interfaces (see Section 10.3 [Specifying Protected and Private Interfaces], page 263 for more information about how this is done). Formally documenting a base class’s protected interface is a step towards the day when users can implement derived classes of such a base class without looking at its code. (There are still other problems to be worked out in this regard, however.) Recording the private interface of a class may be helpful in program development or maintenance. Usually one would expect that the public interface of a class would be specified, and then separate, more refined specifications would be given for use by derived classes and for detailed implementation (and friend classes). (See Section 10.2 [Class Refinement], page 260 for how to record each level in Larch/C++.)

Our specific goals for Larch/C++ are as follows [Leavens-Cheon92b].

- To have a syntax that is intuitive for C++ programmers. The C++ syntax for declarations is used almost without change (see Section C.2 [CPP Differences], page 327 for the small differences).
- To aid the specification of modules that use common C++ idioms. These include subtype polymorphism, virtual functions, mutation, aliasing, reference and pointer types, and the various kinds of inheritance in C++.
- To promote inheritance of specifications. That is, one should be able to specify a class’s interface by stating how it differs from another class’s interface (see Section 7.9 [Inheritance of Specifications and Subtyping], page 215 for how to do this).
- To allow the specification of several interfaces for each class: the public interface (for clients), the protected interface (for derived classes), and the private interface (for implementations and friends).

Originally, it was also a goal of Larch/C++ to have no unmotivated differences from LCL, a specification language for C (see [Guttag-Horning91b] and Chapter 5 of [Guttag-Horning93]). This was intended to avoid rethinking various issues in Larch/C++. Larch/C++ owes much of its syntax
and semantics to LCL. However, for two reasons Larch/C++ is not compatible with LCL. The main reason is that LCL has a particular way to specify ADTs, and Larch/C++ has a completely different style for specifying C++ classes. This is not just a matter of syntax; LCL has a specific method for designing programs that engenders many design decisions. In particular the way LCL models the built-in types (of C) is influenced by its design method. One aspect of the LCL design method, not needed in C++, is that it overcomes the lack of reference parameters in C. This difference leads to the second major reason Larch/C++ is incompatible with LCL: the models of the built-in C++ types are quite different from the corresponding models in LCL. In general the models used by Larch/C++ are more detailed and more faithful to the semantics of C++ (and C). See Section C.1 [LCL Differences], page 326 for a more detailed comparison.

In addition, we have some other goals that are motivated by ideas on object-oriented design [America87] [Meyer88] [Leavens-Weihl90] [Leavens90] [Leavens91] [Leavens-Weihl95]. These ideas use the concept of supertype abstraction, which is the ability to reason about a program based on static (or nominal) type information by letting supertypes stand for all their behavioral subtypes. Informally, a behavioral subtype is a type such that each of its objects acts like some object of its supertypes. Supertype abstraction allows one to reason about a program in terms of these (hypothetical) supertype objects instead of reasoning about the details of the subtype objects. To support supertype abstraction, our goals are as follows [Leavens-Cheon92b].

- To help designers use supertype abstraction in the sense that when behavioral subtypes of existing types are specified, already specified types and functions do not need to be respecified.
- To be able to specify properties needed for the modular verification of C++ programs. That is, when new behavioral subtypes are added to a program, unchanged modules should not have to be reverified [Leavens-Weihl90] [Leavens-Weihl95].

There is still some work remaining on the second point above, particularly with respect to types with mutable objects (see [Dhara-Leavens94b] [Dhara-Leavens96]).

1.3 Status and Plans for Larch/C++

Larch/C++ is still in development. As you can see, this reference manual is still a draft, and there are some holes in it. [[[And some notes for the author by the author that look like this.]]] Most of the small scale parts of the language are adequately documented now. The documentation of classes is in pretty good shape, but still needs work. A lot of work is needed on the documentation of templates and specification modules. Current work on the manual focuses on details of class specifications.
Chapter 1: Introduction

The major work remaining on the design of Larch/C++ is related to: aliasing, subtyping for mutable types, and subclassing. The idea with subclassing is to record enough information in the specification of a class so that someone can program a derived class without looking at the code of the base class. (This work was supported in part by NSF grant CCR-9593168.)

Influences on Larch/C++ that may lead to changes in its design include the evolution of LCL [Tan94] [Chalin95] [Chalin-etal96] and the evolution of C++ itself. Another influence is the ongoing effort to use Larch/C++ on examples, in designing the Larch/C++ tools, and to (ultimately) give a formal semantics to Larch/C++.

The current release of Larch/C++ is not all we would hope for, especially with respect to the tools. It contains just a checker for Larch/C++ and a tool for making HTML files from specifications. The checker does checking that the used LSL traits exist, and checks their syntax, but does not yet use that information to do sort checking within Larch/C++ specifications. (That is coming, however.)

In [Cheon-Leavens94] we had described the automatic generation of a header file (the ‘.lh’ file) from a specification, as was done in LCL (see [Guttag-Horning91b] and Chapter 5 of [Guttag-Horning93]). However, we have abandoned that plan, and now follow the latest version of LCL [Evans96b], by allowing Larch/C++ to be used as an annotation language. The current release allows one to use special comments (of the form //@ and /*@ ... @@*/ that enclose specification constructs. This allows specifications to be added directly to C++ header files (the ‘.h’ files) if desired. See Section 2.4 [Modules and Files], page 17, for more information.

Our highest priority project now is a sort-checker for Larch/C++ specifications, and tools to generate LSL traits that Larch/C++ implicitly creates (such as those used in inheritance of specifications, and those implicitly generated by mentioning C++ types). In the more distant future, we plan some tools to aid formal verification. However, tools to aid formal verification would need to come after a formal semantics for Larch/C++ and probably more research into verification of (a subset of) C++ programs.

Included in the current release of Larch/C++ is a document ‘TODO.ps’ which gives more details on our plans.

1.4 Larch/C++ Tools

There are two currently available tools that you should have to work with Larch/C++: the Larch/C++ checker and the LSL checker. Both of these tools come with examples and document-
tation. Also available is the Larch Prover (LP), which can help debug LSL traits, and could be useful in proving properties of Larch/C++ specifications or in program verification.

The following describes how to obtain these tools, and how to use them.

1.4.1 Obtaining and Installing the Larch/C++ Release

To install the checker for Larch/C++ you will need a C++ compiler. The current release has been tested with the GNU g++ compiler under HP-UX 10.10 and Windows 95 under DJGPP. Older versions have been ported to various flavors of Linux, HP-UX, Sun OS (version 4), Solaris 5.3, and Ultrix with AT&T C++ 3.0, SparcWorks CC 3.0. (We are interested in other ports; let us know if you port it somewhere else.) If you do not have a C++ compiler, you can still get the release for the manual and the examples, but you will not be able to check any examples you write.

If you want to use Windows 95, Windows NT, or some similar system, instead of following the steps below, you can get a version of Larch/C++ with pre-compiled executables by getting ‘LCPPWin.5.12.zip’ by anonymous ftp using the following URL.


Then unzip the files, and read the ‘README’ file. This works under DJGPP, and has been tested under Windows ’95. It should certainly work under Windows NT. It should work to some extent under MS-DOS, but because that system does not support long file names, you may have problems. It’s unclear to what extent it will work under OS/2.

For Unix systems, you can get the current beta-test release of Larch/C++ by following the steps below.

- Create a directory to hold Larch/C++ and other Larch tools. (You will probably want at least the LSL checker, and you might also want LP. You will need at least version 3.1beta10.) On Unix, we suggest something like ‘/u/Larch’, or ‘/usr/unsup/larch’. We will use ‘/u/Larch’ as an example below. Substitute whatever directory you actually use. That is, execute something like:

  mkdir /u/Larch

- Put the gzipped tar file for the release in your directory, by using anonymous ftp from the ftp directory at ftp.cs.iastate.edu. It is found in the following URL.

(If for some reason that isn’t the right file name, or if you want to look at other versions, use the URL

\texttt{ftp://ftp.cs.iastate.edu/pub/larchc++/}

to look at the directory itself.)

- You now have the compressed (by \texttt{gzip}) tar file in your directory ‘/u/Larch’. Use \texttt{gunzip} to uncompress it, and use \texttt{tar} to untar it with commands such as the following (on Unix). (If you do not have the GNU command \texttt{gunzip}, you can get it by anonymous ftp from \texttt{prep.ai.mit.edu} in directory ‘/pub/gnu’, or at any of several mirror sites.)

\begin{verbatim}
gunzip LCPP.5.12.tar.gz
tar -xvf LCPP.5.12.tar
\end{verbatim}

- Change to the directory ‘LCPP’, and read the file ‘README’ for further instructions. For example, on Unix do the following.

\begin{verbatim}
cd LCPP
more README
\end{verbatim}

- Please send us e-mail to let us know that you installed Larch/C++, what kind of machine and operating system you installed it on. If you have trouble in the installation, please let us know so we can try to help you. We are also interested in your feedback on Larch/C++: things you like, do not like, find confusing, etc. Please let us know how we can improve Larch/C++ for you. Our e-mail address is:

\texttt{lc++@cs.iastate.edu}

\section*{1.4.2 Obtaining LSL and LP}

You will need to get the LSL checker, so that the Larch/C++ tools can check on any traits you specify. It can be obtained by anonymous \texttt{ftp} from the following URL.

\texttt{ftp://larch.lcs.mit.edu/pub/Larch/}

Get the ‘LSL-\texttt{README.new}’ file in that directory for detailed instructions.

After you install LSL, some steps have to be taken to ensure that everyone using both Larch/C++ and LSL can find the traits that come with Larch/C++. You can take care of it yourself as follows. On Unix, set your \texttt{LARCH\_PATH} environment variable to include the directory where the Larch/C++ built-in traits live, followed by the directory where the LSL handbook traits live. In our example installation, you would set your \texttt{LARCH\_PATH} environment variable to the following value.

\texttt{.:/u/Larch/LCPP/lib:/u/Larch/LSL/lib}
If you wish to do more extensive debugging of specifications or theorem proving, you may want to also get LP. You can get this from the global home page for Larch at the following URL.

http://www.sds.lcs.mit.edu/spd/larch/

You can also get it from the MIT ftp directory at the following URL.

ftp://larch.lcs.mit.edu/pub/Larch/lp/

1.4.3 Typical Use of the Tools

With our current tools, typical use consists of the following steps.

- Writing a trait, into a file ‘foo.lsl’, and a Larch/C++ behavioral interface specification, into a file ‘foo.h’ (or ‘foo.lh’).
- Checking the trait using lsl.
  
  lsl foo.lsl
- Checking the Larch/C++ specification.
  
  lcpp foo.h
- Think, perhaps use the Larch Prover (LP) to debug, revise, and repeat until satisfied.

After writing the specifications, if you do not have C++ code already written, one would typically write header files and C++ code for what was specified.

1.5 Acknowledgements

This work was supported in part by the United States National Science Foundation under Grants CCR-9108654, CCR-9503168, and CCR-9803843.

I gratefully acknowledge the large contributions made by Yoonsik Cheon. Yoonsik has been there from the beginning, helped design Larch/C++, helped implement the initial versions of the Larch/C++ parser, helped design and manage the release, and helped write several technical papers and the initial versions of this manual. His advice on technical matters is always valuable, and in large part Larch/C++ would not exist without his aid.
I am also very grateful to Clyde Ruby, who during 1998 and early 1999 spent large amounts of time wrestling with the difficult C++ grammar, and getting our tools into better shape. He also updated the grammar in this reference manual, and has helped with various new parts of the syntax, notably the calls-clause, the accesses-clause, the represents-clause, and behavior programs. Thanks Clyde!

Special thanks to Jeannette Wing for suggesting this research project, and for providing hints for Larch/C++ in her paper on Avalon/C++ [Wing90b].

Thanks to Jim Horning, John Guttag, and Yang Meng Tan for their design of LCL (a Larch/C). Thanks to Yoonsik Cheon, Krishna Kishore Dhara, Matt Markland, Clyde Ruby, Hua Zhong, Albert Baker, and Tim Wahls for many discussions on technical matters related to Larch/C++. Al Baker’s insistence that specifiers should be able to do a lot without writing complex traits is, I hope finally realized in of Larch/C++; thanks! Thanks also to Al for help with the syntax and semantics of redundancy. Thanks to participants in the Object-Oriented Formal Methods seminar during Fall 1993, the Formal Methods seminar of Summer 1992, the seminar on the Development of Larch-style Interface Specification Languages in Fall 1994, and the seminar on Specification and Verification with Larch/C++ in Fall 1995. Thanks to Mark Vandevoorde, Dave Detlefs, and Yang Meng Tan for a series of personal communications about the semantics of trashed. Thanks to Dave Evans and especially to Patrice Chalin for a series of personal communications about the semantics of trashed and the modifies-clause. Patrice finally convinced me that his semantics was better; thanks! Thanks to Rustan Leino for his work on the semantics of dependencies, and several discussions about it and its implications for Larch/C++. Thanks to Peter Mueller for a correction to the semantics of invariants and their explanation, which was also applied to the semantics of constraints. Thanks to Dave Musser and participants at the Dagsthul workshop on generic programming for discussions about generic programming and templates.

Thanks to Radhika Allada and Bob Lavey for help with some of the tools. Thanks to Dave Egle for being an excellent user and evaluator of Larch/C++. Thanks to Kurt Bischoff for developing his attribute grammar evaluator, Ox, and for his support of that software. Thanks to Kelvin Nilsen, who supervised the development of Ox, and who helped with using it. Thanks to Hans Boehm for his conservative garbage collector software, and his help in using it correctly. Thanks to Matt Markland for getting namespace support and checking of LSL traits into the Larch/C++ checker. Thanks to Clyde Ruby for much work on caching of traits in the implementation, and for helping with the calls-clause.

Thanks to Piero Colagrosso, K. Kishore Dhara, Dave Egle, Sudipto Ghosh, Marybeth Gurski, John Guttag, Mike Haverdink, Jim Horning, Steve Jenkins, Kevin Jones, Bob Lavey, Rustin Leino, Jacek Leszczylowski, Matt Markland, Tong Mei, John Penix, Antonios Protopsaltou, Arnd Poetzsch-Heffter, John Rood, John Rose, Pierre-Yves Schobbens, Ulf Schuenemann, Tapas Shomee,
David Sims, Gowri Sivaprasad, Yang Meng Tan, Tim Wahls, Jeannette Wing, and others we may have forgotten for comments about Larch/C++ and drafts of this manual. Thanks to Patrice Chalin for comments about LCL [Chalin95] [Chalin-etal96] and for email discussions, which lead to some significant semantic improvements in Larch/C++. Thanks to Hua Zhong for improvements to the traits used in htis manual that have emerged from her proofs of their implications.

Thanks to Matt Markland, David Cok, Dave Egle, Albert Esterline, Steve Freeman, Ward Harold, Bob Lavey, Thomas Lindner, Tim Wahls, and others for bug reports and help with portability. Thanks to Madhu Rajmani for finding out about the texi2html program.
2 Fundamental Concepts

There are certain fundamental semantic concepts, mostly relating to C++, that must be discussed before the semantics of Larch/C++ can be presented in detail.

2.1 Viewpoint

Larch/C++ specifies C++ modules (see Section 2.4 [Modules and Files], page 17) from the point of view of clients. In general a module’s client is some other C++ module, a piece of code. Sometimes we speak of a client as though it were the person writing that other piece of code.

Larch/C++ specifications are written from the viewpoint of a client; that is, they specify what interface and behavior a client sees. This is similar to most specification languages. However, Larch/C++ differs from most other specification languages in that, like C++, it distinguishes several different kinds of clients. Some clients have more access than others: these are subclasses (C++ derived classes), member functions of a class, and friends. Since friends can access all of the representation of a class, Larch/C++ allows one to specify three interfaces to each class, including private details (see Section 2.2 [Interfaces], page 15).

2.2 Interfaces

A function’s interface consists of the function’s name, its return type, the number and types of its arguments, and the types of exceptions it may throw. A Larch/C++ specification of a function can only be implemented by a C++ function, because the interface specified includes the C++ calling sequence, the way that function affects the C++ type checker, and so on. The concept of a behavioral interface specification language is discussed further in [Guttag-Horning-Wing85b] [Wing87] and [Lamport89]. (Also see Section 1.1 [Larch-style Specifications], page 2.)

In C++ a class has three interfaces (see Section 9.1c of [Ellis-Stroustrup90]).

- A public interface which can be used by all clients of the class (including subclasses, member functions of the class, and friends). The public interface consists of the name of the class, the public subclass relationships of that class, and the interfaces of the public members of the class.

In the following example, the class date has as public members: the constructor date, and the member functions day_of_month, month, and year, with the return types and parameters
listed following the C++ keyword public:. The class date is a public subclass of printable, and thus any public members of printable are also public members of date.

```cpp
class date : public printable, protected storable, private fast {
    public:
        date(int day, int month, int year);
        int day_of_month();
        int month();
        int year();
    protected:
        void set_day(int day);
        void set_month(int month);
        void set_year(int year);
    private:
        int dy, mo, yr;
};
```

- A protected interface can be used by member functions of the class, friends, and subclasses. The protected interface consists of the public interface plus all protected subclass relationships and protected members. In the example above the protected members are set_day, set_month, set_year and any protected members inherited from the superclasses printable and storable. The fact that date is derived from storable, is a protected subclass relationship, and thus part of the protected interface.

- A private interface can only be used by member functions and friends (see Section 5.4.1 of [Stroustrup91]), not by subclasses or clients. The private interface consists of the protected interface (including the public interface) and all private subclass relationships and private members. In the example above the private members are the integer variables dy, mo, and yr. The fact that date is derived from fast is a private subclass relationship, and thus part of the private interface.

### 2.3 Accessibility of Class Members in Specifications

Larch/C++, unlike most other specification languages, allows you to record implementation design decisions. (LM3, following Modula-3, also has a similar ability, see Chapter 6 of [Guttag-Horning93] and [Jones91].) Such detailed design decisions are recorded by specifying private and protected members of a class. For example, one might want to specify that the date class was privately derived from the class fast, if that captured some aspect of a design that could not otherwise be specified in Larch/C++ (such as some aspect of a time budget). Other uses for this would be making two (or more) specifications for a class such as date; one specification would be a normal one, specifying only the public interface, while a refinement of this specification could be used to specify the protected member functions. The specification with protected members would
be useful for those programming subclasses. See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for a more extended discussion on this point.

Unlike very early versions of Larch/C++ [Leavens-Cheon92b], all specified class members are accessible within the specification of public (and protected and private) member functions. As in C++, protected members are accessible from within the specification of a derived class (see Section 7.8 [Specifying Derived Classes], page 214 for details).

As a matter of style, you should specify types for which you wish to record design decisions in several layers. Doing this allows each kind of client to read a specification that is at the appropriate level of detail. For example you might first specify the public member functions so that their specifications can be used without the details of how they affect the private or protected members. Then you might specify the protected data members and any protected member functions, while at the same time giving additional details of the public member functions. Another such layer can be added for private members if desired. See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for how to use the Larch/C++ notion of refinement to do this.

2.4 Modules and Files

As in C++, each Larch/C++ specification file is a module. (Note that a class is not a module, but may sometimes be the only thing in a module.) A module may include other modules, using the usual \#include syntax of C++. A module may also contain statements about what trait(s) to use, and specifications.

It was an early convention, still preserved in many examples, to have the names of a Larch/C++ specification file ends in the suffix `.lcc`. For example ‘date.lcc’ might be the name of a file that contained the module specifying the class date. However, this is no longer, strictly speaking, necessary. Now one may use any suffix desired, such as ‘date.h’ or ‘date.lh’. One of these is more appropriate if the specification can also be used directly as a C++ header file.

As in C++, declarations in a module may be hidden or exported. A hidden declaration should be marked with the C++ keyword static. Exported declarations should not be so marked. See Chapter 9 [Specification Modules], page 254 for an example.

The \#include mechanism has the same semantics as C++ (see section r.16 of [Stroustrup91]). The Larch/C++ tools use a C++ preprocessor as a first phase, exactly as in C++. 
While Larch/C++ uses the C++ \#include syntax, one does not necessarily have to use separate “header” modules for Larch/C++ specifications. Typically Larch/C++ specification will be found in C++ header modules.

### 2.5 Declarations and Definitions

Larch/C++ adopts the distinction between declarations and definitions from C++ (Section 3.1 of [Ellis-Stroustrup90]). That is, a declaration tells the reader about the properties of a name. A definition, in addition, specifies the value to be associated with a name, or in the case of a variable definition, specifies that storage should be allocated. More formally, we adopt the exact distinction from C++ except that instead of a function body, a function specification makes what would otherwise be only a function declaration into a function definition.

For example, the following are definitions.

```cpp
int i;
struct rat { public: int num; int denom; };
int inc(int count) { count = count + 1; }
```

However the following are only declarations.

```cpp
extern int i;
struct rat;
extern int inc(int count);
```

In any scope unit (see Section 2.6 [Scope Rules], page 18), there can be only one definition of a C++ name, or a function with a given list of argument types, in a Larch/C++ specification. However, declarations can be repeated, and LSL trait functions can have multiple overloading with the same list of argument types.

### 2.6 Scope Rules

A scope unit is an area of specification text within which declarations may have effect, within which multiple definitions of the same name are prohibited, and outside of which declarations inside the unit do not have effect. The scope units in Larch/C++ are as in C++ (Section 3.2 of [Ellis-Stroustrup90] and Section r.3.3.1 of [Stroustrup94]), with the addition of a scoping unit for function specifications, spec-cases, and quantifiers.
The new scope units in Larch/C++ are as follows.

- [function specification] Larch/C++ function specifications (see Chapter 6 [Function Specifications], page 82) define a scope unit that is the same as a C++ local scope unit.
- [specification case] A spec-case (see Chapter 6 [Function Specifications], page 82) is a scope unit.
- [quantifier] An equality-term (see Section 6.1.3 [Equality Terms and Quantifiers], page 87) of the form (see Chapter 3 [Syntax Notation], page 35 for details on the notation):
  \[\text{quantifier} \ [\text{quantifier} \ ] \ldots (\text{term})\]
  is a scope unit.

The scope rules of Larch/C++ are exactly as in C++ (Section 10.4 of [Ellis-Stroustrup90] with the addition of namespaces as in [Stroustrup94]), with the following additions.

- [function specification scope] In a function specification (see Chapter 6 [Function Specifications], page 82), the declarations of the formals of the function, and all global variables (see Section 6.7 [Global Variables], page 134) in the function’s specification have the usual lexical scope, analogous to C++ local scope.

  For example, scope of the declaration of the variable `counter` is the body of the specification of `inc_counter`.

  ```
  // @(#)$Id: inc_counter.lh,v 1.8 1997/06/03 20:30:07 leavens Exp $
  extern int inc_counter(void) throw();
  //@ behavior {
  //@ extern int counter;
  //@ requires assigned(counter, pre) \ counter^ < INT_MAX;
  //@ modifies counter;
  //@ ensures counter’ = counter + 1;
  //@ }
  ```
  
  (Unlike previous versions of Larch/C++, no exception to the usual C++ scope rules is made for function specifications. That is, global variables are visible within function specifications, and do not necessarily need to be declared as above. However, it is considered good style in a function specification to declare global variables that may be read or written.)

- [specification case scope] In a function specification, each spec-case may have a let-clause (see Section 6.8 [Let Clauses], page 135) which declares abbreviations for terms used in the rest of that spec-case. These declarations have the usual lexical scope, analogous to C++ local scope.

- [quantifier scope] In an assertion, quantified logical variable declarations have the usual lexical scope, analogous to C++ local scope. For example, in the following assertion the scope of `i:int` is the rest of the line following `i:int`.

  ```
  \A i: int (i < INT_MAX \implies (i < i + 1))
  ```
2.7 Types and Sorts

In Larch/C++, the interface layer specifies types, which can be used in C++ programs. The shared layer specifies sorts, which are used in traits. Each type identifier specified in a Larch/C++ specification must also be the name of an LSL sort. However, compound C++ type expressions like `unsigned int[]` are not directly the names of sorts, but instead have names like `Arr[Obj[unsigned]]` (see Chapter 5 [Declarations], page 54 for the details on the names of sorts for such types). One other exception is that the C++ type `bool` is modeled by the LSL sort `Bool` (see Section 11.4 [bool], page 276). There can also be sorts that are not names of types; these auxiliary sorts may ease the task of writing a specification. Larch/C++ also automatically introduces certain auxiliary sorts to model some features of C++. An example is the sort `Obj[int]` which is a sort for objects (i.e., variables, see Section 2.8 [Objects and Values], page 21) whose abstract values are of sort `int`.

For types with simple names, like `int`, the type's abstract values are the equivalence classes of LSL terms of the sort with the same name (`int`) that also satisfy the type's invariant (see Section 7.3 [Class Invariants], page 196). For classes and structures, Larch/C++ can automatically construct a trait that defines the abstract values. Alternatively, the user can explicitly specify the abstract values by naming a LSL trait that defines the type name as a sort in the `uses` clause in a specification. For example,

```plaintext
//@ uses MyTrait;
```

If, as often happens, the C++ type name is different than the sort name used in the trait, one can change the name in the trait in the `uses` clause. In effect this constructs a new trait on the spot, with the sort renamed to be the desired type name. One does this using a `replace-list` in the `uses` clause. (See Section 9.2 [The Uses Clause], page 255 for details on the `replace-list` syntax.) For example, one might write the following `uses` clause, to rename the sort `D` to be the type name `date` in the trait `DateTrait`.

```plaintext
//@ uses DateTrait(date for D);
```

In this example the abstract values of the type `date` are the equivalence classes of the sort `date`, in the newly constructed trait; in essence the abstract values are those of sort `D` in the trait `DateTrait`, but it is convenient to think of the process as producing a new trait.

The names of C++ template types are not legal as LSL sorts, but by changing the `< and >` to `[ and ]` they become LSL sorts. Thus a C++ type such as `Set<int>` is based on a sort named `Set[int]`. This translation, mapping the C++ `< and >` to the LSL `[ and ]`, happens automatically when moving from C++ to LSL, and users normally need not be concerned with it.
However, this translation does not work in the other direction. That is, just because there is a sort named \( \text{Obj}[\text{int}] \), does not mean that there is a C++ type named \( \text{Obj}<\text{int}> \). Furthermore, in Larch/C++, the sort-name \( \text{Obj}[\text{int}] \) is written just like that: \( \text{Obj}[\text{int}] \). (See Section 6.1.3.2 [Quantifiers], page 88 for details on the syntax for sort-name.)

Traits for all of the C++ built-in types, such as \( \text{int} \), are automatically used in a Larch/C++ specification. Each C++ built-in type name, such as \( \text{int} \), is also the name of a sort. See Chapter 11 [Built-in Types], page 269 for the details on how the abstract values of these types are modeled in LSL.

Equivalent C++ type-ids, such as \( \text{unsigned} \) and \( \text{unsigned int} \), are considered to be equivalent sort names for the purpose of sort checking. (Unlike LCL [Guttag-Horning93], but like C++, in Larch/C++ the types \( \text{char}^* \) and \( \text{char}[] \) are the same when used as formal parameters [Chalin95] [Chalin-etal96].)

C++ typedefs (see Section 5.2.5 [Typedef Specifiers], page 63) are expanded before being used in sort checking, and so do not introduce new types or sort names.

2.8 Objects and Values

As in Section 3.7 of [Ellis-Stroustrup90], “an object is a region of storage.” Note that a C++ variable of type \( \text{int} \) is an object in this sense, although it does not respond to messages. We use the term \text{instance} for emphasis when discussing an object of a class; instances are what are usually called objects in object-oriented programming. Instances respond to messages. A variable of type \( \text{int} \) is not an instance. All objects in C++ have an address, or location, which we will sometimes call an \text{object identifier}.

Values are stored in objects. To emphasize that a value is the value of an instance of a class, we sometimes use the term \text{instance value}. In contrast to the abstract values used in specifications, the values manipulated by C++ programs, are thought of as concrete values. Each concrete value, a bit pattern, is thought of as a representation of an abstract value. For example, a C++ \text{int} variable holds a bit pattern that corresponds to an abstract value which is a mathematical integer, specified by the trait \text{int} (see Section 11.1 [Integer Types], page 269).

As an illustration of the concept of objects and values, the following is a picture of the Larch/C++ model of an integer variable, \( i \). In the picture, \( i \) is an object, which contains the abstract value 228. To be more specific, the following is a picture of \( i \) in some program state. A \text{state}, as in the picture, associates to each object, an abstract value. The abstract value of an assignment may
change from state to state, as the program executes (for example, the value of \(i\) may be changed by an assignment statement in C++). For the most part, Larch/C++ specifications only describe states just before and just after the call of a C++ function.

\[
\begin{align*}
\text{i: } & \quad 228 \\
\end{align*}
\]

The Larch/C++ model of an integer variable, containing the abstract value 228.

Unlike C++, in the Larch/C++ model, an object can contain an arbitrarily complex abstract value. That is, a state may map an object to any value defined by a trait. For example, the Larch/C++ picture of a variable \texttt{myHeap} of type \texttt{IntHeap} might be as follows. See Section 1.1 [Larch-style Specifications], page 2 for the specification of the class \texttt{IntHeap} and the trait that defines its abstract values.

\[
\begin{align*}
\text{myHeap: } & \quad \text{add}(6, \text{add}(7, \text{empty})) \\
\end{align*}
\]

The Larch/C++ model of an IntHeap variable, containing the abstract value add(6, add(7, empty)).

Finally, in Larch/C++, values can contain objects, which enables complex, even circular, object structures to be modeled. (One way to think of this is that objects are just a special kind of abstract value.) For example, the following is a picture of variable \texttt{mySet}, whose abstract value is a set of point objects. The picture represents the abstract value of the set as \{\*,\*,\*\}, where the *s are the tails of three arrows that show what object is contained in the set. Each of the point objects has an abstract value that is a LSL tuple of two integers. In general, users of Larch/C++ will model objects and values in such layers.
The Larch/C++ model of a Set variable with an abstract value that contains three other objects.

The word *lvalue* means a term (something like a C++ expression, see Section 6.1 [Predicates], page 86 for details) that denotes an object or a function. Recall that in C++ an array name is *not* an lvalue.

In contrast to C++ (but like LCL), Larch/C++ does not consider formal parameters passed by value to be objects. This is because parameters passed by value cannot be changed from the point of view of the client (the caller), and so are not objects from the point of view of the client. Therefore in Larch/C++, only formal parameters passed by reference are objects. Pointers that are passed by value are not objects but do point to objects. For example, within the specification of a function with the following heading the expressions *cr* and *ip* are lvalues (denote objects), but *i* and *ip* are not.

```c++
int foo(int i, char & cr, int * ip)
```

Whether or not a formal parameter is declared to be const does not affect this distinction. For example, within the specification of a function with heading

```c++
int foo(const int ci, const char & ccr, const int * cip)
```

the expressions *ccr* and *cip* are lvalues, but *ci* and *cip* are not.

The name *result* is used in postconditions to stand for what is returned by the C++ function being specified. It is considered to be an object only if the return type of the function is a reference type. That is, *result* is treated in the same way as a formal parameter. For example, within a function specification with the above header, *result* is *not* an lvalue. However, if the return type were specified as *int &*, then *result* would be an lvalue.
Values are formally modeled by various traits, most of which are specified by Larch/C++ users in LSL. Objects are formally modeled by the traits described below. These traits build on the more primitive notion of a state; states are formally modeled by another trait described below.

(Some of the discussion in the subsections below is quite technical. If you are new to Larch/C++, feel free to skim these subsections.)

### 2.8.1 Formal Model of Objects

Objects in Larch/C++ are modeled using several traits. The main trait is TypedObj, which handles the translation between typed objects and values and the untyped objects and values used in the trait State (see Section 2.8.2 [Formal Model of States], page 30). This model builds on the work of Wing [Wing87], Chen [Chen89], Lerner [Lerner91], and most recently has benefited discussions with Chalin and his work [Chalin95].

The formal model of this trait (and its relation to the sorts in the trait State) may be explained with the help of the following (crudely-drawn) picture. This picture shows a sort $T$ of abstract values, with a representative element, $tval$. The trait function $\text{injectTVal}$ maps this into the sort $\text{WithUnassigned}[T]$, a sort that also includes the special value $\text{unassigned}$. The trait function $\text{extractTVal}$ is its (near) inverse. The sort $\text{Loc}[T]$ of typed objects containing $T$ values (or unassigned) has $\text{loc}$ as a typical element. The sorts $\text{WithUnassigned}[T]$ and $\text{Loc}[T]$ in the trait TypedObj are the typed counterparts of the sorts Value and Object in the trait State. The overloaded trait functions named $\text{widen}$ map typed to untyped values and objects. Their inverses are the trait functions named $\text{narrow}$. In each trait, the $\text{eval}$ mapping takes a second argument which is a state, written as $\text{st}$ in the picture.
The TypedObj trait itself includes several other traits, and uses them to define the sort \( \text{Loc}[T] \). The included traits will be explained individually below.

A picture of the sorts in the traits TypedObj and State, and some of the mappings between them. The second argument to eval is a state, st.
TypedObj(Loc, T): trait
   includes State, WithUnassigned(T), WidenNarrow(Loc[T], Object),
   WidenNarrow(WithUnassigned[T], Value), TypedObjEval(Loc, T),
   AllocatedAssigned(Loc, T), ModifiesSemantics(Loc, T),
   FreshSemantics(Loc, T), TrashesSemantics
   asserts
   sort Loc[T] generated by narrow
   sort Loc[T] partitioned by widen

The conversions to and from typed and untyped versions of objects and values are defined by
the two inclusions of the trait WidenNarrow given below.

% @(#)$Id: WidenNarrow.lsl,v 1.3 1997/02/13 00:21:25 leavens Exp $
% Maps between untyped and typed values.
% This could be used to describe any partially inverse pair of mappings.
WidenNarrow(Typed, Untyped): trait
   introduces
   widen: Typed -> Untyped
   narrow: Untyped -> Typed
   asserts
   \forall t: Typed
   narrow(widen(t)) == t;
   implies
   \forall u: Untyped
   narrow(widen(narrow(u))) == narrow(u);

The sort WithUnassigned[T] is specified by the following trait. (Those who are familiar with
denotational semantics [Schmidt86] will recognize this as the “lift” of T, with unassigned used in
place of the usual notation for a bottom element. The mappings injectTVal and extractTVal
are explicit conversions to and from this lifted set.)

% @(#)$Id: WithUnassigned.lsl,v 1.1 1995/11/06 05:12:17 leavens Exp $

WithUnassigned(T): trait
   introduces
   injectTVal: T -> WithUnassigned[T]
   extractTVal: WithUnassigned[T] -> T
   unassigned: -> WithUnassigned[T]
   isUnassigned: WithUnassigned[T] -> Bool
The trait `TypedObjEval` is defined below. Evaluation is, as in the picture above, defined by widening the typed object to an untyped object, using the untyped `eval` to get the untyped object's value, and narrowing that value to a `WithUnassigned[T]` value, then extracting that to a value of type `T`.

```lsl
TypedObjEval(Loc, T): trait
  assumes State, WithUnassigned(T), WidenNarrow(Loc[T], Object),
           WidenNarrow(WithUnassigned[T], Value)
  introduces
    eval: Loc[T], State -> T
  asserts
    \forall loc: Loc[T], st: State
    \hspace{1em} eval(loc, st) == extractTVal(narrow(eval(widen(loc), st)));
  implies
    converts
    eval: Loc[T], State -> T
    \hspace{1em} exempting \forall loc: Loc[T], st: State, typs: Set[TYPE]
    \hspace{1em} eval(loc, bottom), eval(loc, emptyState),
    \hspace{1em} eval(loc, bind(st, widen(loc), widen(unassigned), typs))
```

The trait `AllocatedAssigned` defines notions of when a typed object is allocated in a state, and when it is assigned a well-defined value (i.e., is not unassigned). See Section 6.2.2 [Allocated and Assigned], page 107 for details.
The trait \texttt{ModifiesSemantics} defines trait functions that help give a semantics to the Larch/C++ modifies clause. See Section 6.2.3 [The Modifies Clause], page 110 for details.

The trait \texttt{FreshSemantics} defines trait functions that help in giving the semantics of the Larch/C++ built-in \texttt{lcpp-primary fresh}. See Section 6.3.1 [Fresh], page 123 for details.

The trait \texttt{TrashesSemantics} defines trait functions that help in giving the semantics of the Larch/C++ \texttt{trashes-clause}. See Section 6.3.2 [The Trashes Clause], page 126 for details.

Objects in Larch/C++ come in two flavors, mutable and constant (immutable). Mutable objects include global variables and reference parameters. Constant objects include global variables declared using the C++ \texttt{cv-qualifier const}.

\textbf{2.8.1.1 Formal Model of Mutable Objects}

Mutable objects are modeled by sorts with names of the form \texttt{Obj[T]}, which is the sort of an object containing abstract values of sort \texttt{T}. The trait \texttt{MutableObj} gives the formal model of mutable objects by adding the capability of mutation to the trait \texttt{TypedObj}. (See Section 2.8.2 [Formal Model of States], page 30 for the definition of \texttt{updateValue} for untyped objects.) Having the trait function \texttt{contained_objects} defined for mutable objects is useful in specifying template container classes (see Chapter 8 [Template Specifications], page 239). In any case, all sorts of values must have the trait function \texttt{contained_objects} defined, and objects are considered to be values in Larch/C++, so such a definition is necessary. See Section 7.5 [Contained Objects], page 207 for more details on contained objects.
**Chapter 2: Fundamental Concepts**

2.8.1.2 Formal Model of Constant Objects

Constant objects are modeled by sorts with names of the form `ConstObj[T]`, which is the sort of a constant object containing abstract values of sort `T`. The trait `ConstObj` gives the formal model of constant objects. See Section 7.5 [Contained Objects], page 207 for the details of the trait `contained_objects`. The trait function `contained_objects` is defined so that constant objects work correctly with various sugars for C++ structs.

```latex
\forall cobj: ConstObj[T], st: State
\text{contained_objects}(cobj, st)
== \text{if} \text{assigned}(cobj, st)
\text{then} \text{contained_objects}(\text{eval}(cobj, st), st)
\text{else} \{};
\text{implies}
\text{converts contained_objects: ConstObj[T], State -> Set[TypeTaggedObject]}
```
2.8.2 Formal Model of States

States (sort \texttt{State}) can be thought of as finite mapping from untyped objects (sort \texttt{Object}) to untyped values (sort \texttt{Value}) and sets of types (sort \texttt{Set[TYPE]}). The trait \texttt{State_Basics} gives the basic operators on states used by Larch/C++. The state \texttt{bottom} represents the state that results from a nonterminating computation or error. The state \texttt{emptyState} is the empty finite mapping. The trait function \texttt{bind} adds a mapping from an object to a value, overriding any previous mapping for that object. The trait function \texttt{eval} gives the value associated with an object by a state.

%% 
\% @(#)$Id: State_Basics.lsl,v 1.11 1997/02/13 00:21:15 leavens Exp $
\% The sort State is the sort of C++ states, which this formalizes.
\% This is adapted from traits used by GIL [Chen89] and GCIL [Lerner91].
% Hua Zhong helped revise and improve this trait, and proved its implications.

\texttt{State_Basics}: trait
\begin{verbatim}
introduces
 emptyState, bottom: \rightarrow State
 bind: State, Object, Value, Set[TYPE] \rightarrow State
 allocated: Object, State \rightarrow Bool
 isBottom: State \rightarrow Bool
 eval: Object, State \rightarrow Value
\end{verbatim}

\begin{verbatim}
asserts
 State generated by emptyState, bottom, bind
 State partitioned by allocated, eval, isBottom
 \forall obj, obj1: Object, st: State, v: Value, typs: Set[TYPE]
  \sim allocated(obj, emptyState);
  \sim allocated(obj, bottom);
  allocated(obj, bind(st, obj1, v, typs))
   == \sim isBottom(st) \backslash ((obj = obj1) \backslash allocated(obj, st));
  \sim isBottom(st) \Rightarrow
       (eval(obj1, bind(st, obj, v, typs))
        = (if obj1 = obj
            then v
            else eval(obj1, st)));
  \sim isBottom(emptyState);
  isBottom(bottom);
  isBottom(bind(st, obj, v, typs)) == isBottom(st);
  bind(bottom, obj, v, typs) == bottom;
\end{verbatim}
implies
\forall obj, obj1:Object, st:State, v,v1: Value,
typs,typs1: Set[TYPE]
emptyState ~= bottom;
isBottom(st) == (st = bottom);
~isBottom(st) => (bind(st,obj,v,typs) ~= bottom);
~isBottom(st) =>
  (eval(obj1, bind(st, obj1, v, typs)) = v);
~isBottom(st) =>
  (eval(obj1, bind(st, obj, v, typs))
   = (if obj1 = obj
      then eval(obj1, bind(emptyState, obj1, v, typs))
      else eval(obj1, st)));
~isBottom(st) =>
  (allocated(obj1, bind(st, obj, v, typs))
   = (if obj1 = obj
      then allocated(obj1, bind(emptyState, obj1, v, typs))
      else allocated(obj1, st)));
bind(bind(st, obj, v, typs), obj, v1, typs1)
  == bind(st, obj, v1, typs1);
bind(bind(st, obj, v, typs), obj1, v1, typs1)
  == if obj1 = obj
      then bind(st, obj1, v1, typs1)
      else bind(bind(st, obj1, v1, typs1), obj, v, typs);
converts allocated, eval, isBottom
exempting \forall obj:Object
eval(obj,bottom), eval(obj,emptyState)

For each allocated object, a state associates with it a set of types. This set records the names of
the sorts to which the object can be sensibly narrowed, using the trait function \texttt{narrow}. Another
way to look at this is that each untyped object can be viewed from a limited number of typed
perspectives from a C++ program (in a given state). One can imagine the compiler recording this
information for each object, potentially updating this information when new aliases are developed
to the object with different type perspectives. See Section 6.2.2 [Allocated and Assigned], page 107
for one way this information is used. The formal model of this information is specified by the trait
\texttt{TypePerspectives} below. An object that is not allocated has no type perspectives.

% @(#)Id: TypePerspectives.lsl,v 1.10 1996/11/15 12:30:46 leavens Exp $
% What a state knows about the "types" an object may have
% This trait was corrected and proved by Hua Zhong.
TypePerspectives: trait
assumes State_Basics
includes Set(TYPE, Set[TYPE], int for Int)
introduces
types_of: Object, State -> Set[TYPE]
hasType: Object, State, TYPE -> Bool
asserts
\forall obj, obj1: Object, t: TYPE, typs: Set[TYPE],
v: Value, st: State
\begin{align*}
types_of(obj, emptyState) &= 
\{};
\end{align*}
\begin{align*}
types_of(obj, bottom) &= 
\{};
\end{align*}
\begin{align*}
\neg\text{isBottom}(st) \Rightarrow
\begin{align*}
types_of(obj, bind(st, obj1, v, typs))
&= \begin{cases} 
\text{if} \ (obj= obj1) \ \text{then} \ typs \\
\text{else} \ types_of(obj, st));
\end{cases}
\end{align*}
\end{align*}
\begin{align*}
\text{hasType}(obj, st, t) &= t \ \text{in} \ types_of(obj, st); 
\end{align*}
implies
\forall obj, obj1: Object, t: TYPE, v: Value, st: State
\begin{align*}
\neg\text{allocated}(obj, st) \Rightarrow (types_of(obj, st) = \{}); 
\end{align*}
converts types_of, hasType

An untyped object may have its value and type perspectives updated independently. For example, in mutation of a typed object, loc, the type perspectives recorded for the underlying untyped object, widen(loc), do not change. The trait State_Updates specifies these updates.

State_Updates: trait
assumes State_Basics, TypePerspectives
introduces
updateValue: State, Object, Value -> State
updateTypes: State, Object, Set[TYPE] -> State
asserts
\forall obj, obj1: Object, typts, typs1: Set[TYPE],
v, v1: Value, st: State
\begin{align*}
\text{updateValue}(emptyState, obj1, v1) &= emptyState; 
\end{align*}
\begin{align*}
\text{updateValue}(bottom, obj1, v1) &= bottom; 
\end{align*}
\begin{align*}
\text{updateValue}(bind(st, obj, v, typs), obj1, v1)
&= \begin{cases} 
\text{if} \ obj = obj1 \ \text{then} \ bind(st, obj, v1, typs) \\
\text{else} \ bind(\text{updateValue}(st, obj1, v1), obj, v, typs); 
\end{cases}
\end{align*}
\begin{align*}
\text{updateTypes}(emptyState, obj1, typs1) &= emptyState; 
\end{align*}
\begin{align*}
\text{updateTypes}(bottom, obj1, typs1) &= bottom; 
\end{align*}
\begin{align*}
\text{updateTypes}(bind(st, obj, v, typs), obj1, typs1)
&= \begin{cases} 
\text{if} \ obj = obj1 \ \text{then} \ bind(st, obj, v, typs1) \\
\text{else} \ bind(\text{updateTypes}(st, obj1, typs1), obj, v, typs); 
\end{cases}
\end{align*}
implies
\forall obj: Object, typs: Set[TYPE], v: Value, st: State
\neg \text{allocated}(obj, st) \Rightarrow \forall v (\text{updateValue}(st, obj, v) = st);
\neg \text{allocated}(obj, st) \Rightarrow \forall v (\text{updateTypes}(st, obj, typs) = st);
converts \text{updateValue}, \text{updateTypes}

The trait \text{State} below has the complete model of states for Larch/C++. It includes \text{State_Basics} and the other traits defined above.

\% @(#)\$Id: State.lsl,v 1.23 1997/02/13 00:21:14 leavens Exp $

\text{State: trait}
\quad \text{includes} \text{State_Basics, TypePerspectives, State_Updates,}
\quad \text{Set(Object, Set[Object], int for Int)} \quad \% \text{from LSL handbook}
\introduces
\quad \text{domain}: \text{State} \rightarrow \text{Set[Object]}
\asserts
\quad \forall obj: Object, st: \text{State}
\quad \text{obj} \in \text{domain(st)} = \text{allocated(obj, st)};
\implies
\quad \text{equations}
\quad \text{domain(\text{emptyState})} = \{\};
\converts \text{domain}

\section{2.9 Satisfaction}

The meaning of a Larch/C++ specification module is a set of C++ code and header files that satisfy it. (This set may be empty if one cannot write a C++ module to satisfy the specification.) A C++ header (‘.h’) file, and an optional C++ code (‘.c’, ‘.cc’, etc.) file, satisfy their specification if and only if the following conditions hold.

- The header file contains as code all the C++ declarations specified (with the exception of spec-decls, see Section 9.1 [Ghost Variables], page 254), and C++ declarations for each of the definitions specified (including all classes specified).
- For each function, class, etc., definition specified, the code file contains a C++ definition that satisfies the specification of that function, class, etc. The appropriate notions of satisfaction are described in the bulk of this manual. Note that the code file may obtain some or all of the specified implementations by including the header file. If no implementations are called for, then the C++ code file may be omitted.
- The C++ header file, and the C++ code file when present, are legal C++ code.
The C++ header file can be legally included by multiple modules in a single program. Thus the header file should not, usually, contain code for C++ functions and the member functions of classes. (An exception would be for inline functions.) Also the header file should not contain a definition of a global variable. (It should only contain a declaration of such.)

If a header file and an optional code file satisfies a specification $S$, then we say that they implement $S$, or for emphasis, that they correctly implement $S$.

The reason for using both a header and a code file (the latter only if needed) to implement each Larch/C++ specification is to promote reuse of C++ code. Thus each C++ implementation has a header file which can be included by other C++ program parts to obtain the declarations of interface. This makes it easier to get the C++ types correct. The Larch/C++ release has several examples of such code with implementations.
3 Syntax Notation

We use an extended BNF grammar to describe the syntax of Larch/C++. The extensions are as follows [Ledgard80].

- Nonterminal symbols are written as follows: *nonterminal*. That is, nonterminal symbols appear in an *italic* font (in the printed manual).
- Terminal symbols are written as follows: *terminal*. In a few cases it is also necessary to quote terminal symbols, such as when using ‘|’ as a terminal symbol instead of a meta-symbol (see Section 4.10 [Special Symbols], page 43).
- Square brackets ([ and ]) surround optional text. Note that [ and ] are terminals.
- The notation … means that the preceding nonterminal or group of optional text can be repeated zero (0) or more times.

For example, the following gives a production for a non-empty list of *init-declarators*, separated by commas.

\[
\textit{init-declarator-list} ::= \textit{init-declarator} \ [ \ , \ \textit{init-declarator} \ ] \ldots
\]

To remind the reader that the notation ‘…’ means zero or more repetitions, we try to use ‘…’ only following optional text, although in cases such as the following the brackets and the enclosed nonterminal could have been omitted.

\[
\textit{type-specifier-seq} ::= \textit{type-specifier} \ [ \ \textit{type-specifier} \ ] \ldots
\]

As in the above examples, we follow the evolving C++ standard [ANSI95] in using nonterminal names of the form X-list to mean a comma-separated list, and nonterminal names of the form X-seq to mean a sequence not separated by commas.

We use “//” to start a comment (to you, the reader) in the grammar.
4 Lexical Conventions

This chapter presents the lexical conventions of Larch/C++; that is, the microsyntax of Larch/C++. At the end of the chapter, support for international character sets is described.

Throughout this chapter, grammatical productions are to be understood lexically. That is, no white-space (see Section 4.1 [White Space], page 36) may intervene between the characters of a token.

The microsyntax of Larch/C++ is described by the production microsyntax below; it describes what a program looks like from the point of view of a lexical analyzer [Watt91].

\[
\text{microsyntax ::= lexeme } [ \text{ lexeme } ] \ldots \\
\text{lexeme ::= white-space } \mid \text{ comment } \mid \text{ annotation-marker} \\
\mid \text{ pragma } \mid \text{ token}
\]

4.1 White Space

Blanks, horizontal and vertical tabs, carriage returns, formfeeds, and newlines, collectively called white space, are ignored except as they serve to separate tokens. Newlines are special in that they cannot appear in some contexts where other whitespace can appear, and are also used to end C++-style comments (see Section 4.2 [Comments], page 36).

\[
\text{white-space ::= non-nl-white-space } \mid \text{ newline} \\
\text{non-nl-white-space ::= a blank, tab, vertical tab, carriage return, or formfeed character} \\
\text{newline ::= a newline character}
\]

4.2 Comments

Both kinds of C++ comments are allowed in Larch/C++: old C-style comments and new C++-style comments. However, if what looks like a comment starts with the at-sign (@) character, then it is considered to be the start of an annotation by Larch/C++, and not a comment. (See Section 4.3 [Annotations], page 38, for details.) Comments that are not annotations do not affect the meaning of a Larch/C++ specification, and should be included for the benefit of the human reader.
• In a C-style comment, all the text between /* and the next */ is a comment. This comment pair can appear anywhere a white-space character is permitted and can span multiple lines of specification. Such comments cannot be nested.

• New C++-style comments extend from a double slash, //, to the end of the line of the specification file in which the // appears.

```
comment ::= C-style-comment | C++-style-comment
C-style-comment ::= /* [ C-style-body ] C-style-end
    | stars-non-slash [non-star-slash] ... |
    stars-non-slash ::= * [ ] ... non-slash
    non-star-slash ::= any character except @ or *
    stars-non-slash ::= any character except *
    non-slash ::= any character except /
    C-style-end ::= [ ] ... */
C++-style-comment ::= // newline
    | // non-at-newline [ non-newline ] ... newline
    non-newline ::= any character except a newline
    non-at-newline ::= any character except @ or newline
```

The character sequences //, //@, /*, */*, /*@, and @/* have no special meaning within a // comment and treated just like other characters. Similarly, the character sequences //, //@, /*@, and */ have no special meaning within a /* comment.

The following are an examples of comments.

```plaintext
// @(#)$Id: comments.lh,v 1.1 1997/01/10 23:33:17 leavens Exp $

/* a C-style comment looks like this
   and may continue for several lines */

// A C++-style comment looks like this.
// (The first line is one too.)

// a */weird*/ case of a C++-style (//) comment
/* an equally strange // C-style (*/) comment /**
// the following is not an annotation: //@ ok?
/* and */@ neither is this part */
```
4.3 Annotations

If what looks to C++ as a comment starts with an at-sign (@) as its first character, then it is not considered a comment by Larch/C++. We shall refer to the tokens between //@ and the following newline, and between pairs of /*@ and @*/ as annotations. Annotations look like comments to C++, and are thus ignored by it, but they are significant to Larch/C++. This is achieved by having Larch/C++ drop (i.e., do nothing with) the character sequences that are annotation-markers: //@, /*@, and @*/.

annotation-marker ::= //@ | /*@ | @*/

Within an annotation, all of the syntax of Larch/C++ is recognized, including comments and even annotations. The following are examples of C++ style annotations, that show some comments within the annotations.

//@ behavior { // this is the behavior of returns7
//@ ensures result = 7; /* a gratituous comment */
//@ }

4.4 Pragmas

C++ pragmas and gcc/g++ attribute declarations are ignored by Larch/C++. They have the following syntax. The first form of pragma must look like a C++ macro definition in the sense that the # can only be preceded on its line by whitespace.

pragma ::= # non-nl-white-space pragma [ non-newline ] . . .
          | __attribute__ [ non-semi-newline ] . . .
          | __asm__ | __const__ | __inline__
          | __signed__ | __typeof__ | __volatile__
          | __extension__

          non-semi-newline ::= any character except a semicolon (;) or newline

In the following, the text ignored by Larch/C++ is described in the comments.
4.5 Tokens

Larch/C++ lexical conventions are similar to those of C++. The different kinds of tokens are given in the grammar below (and in see Section 4.15 [Alternative Tokens], page 52); they are described in the rest of this chapter.

\[
\text{token ::= identifier | simple-id} \\
\quad | \text{keyword | context-dependent-keyword} \\
\quad | \text{special-symbol | predicate-keyword} \\
\quad | \text{informal-comment | literal | lsl-constant}
\]

4.6 Identifiers

As in C++, identifiers are used to name functions, variables, etc.

\[
\text{identifier ::= letter [ letter-or-digit ] ...} \\
\quad | \text{ident( letter [ letter-or-digit ] ... )}
\]

\[
\text{letter ::= \_ | a through z, or A through Z} \\
\text{digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9} \\
\text{letter-or-digit ::= letter | digit}
\]

The usual form of an identifier is an arbitrary long sequence of letters and digits. The first character of an identifier must be a letter; the underscore (\_) is treated as a letter. (The alphabetic letters only include ASCII codes, no codes for accented characters are included.) Identifiers are case sensitive; i.e., \text{identifier}, \text{Identifier}, and \text{IDENTIFIER} are three distinct identifiers.

Some names that would otherwise be identifiers are reserved by Larch/C++ (see Section 4.8 [Keywords], page 40 for a list of Larch/C++ keywords). In a few rare cases, one needs to use a
Larch/C++ keyword as the name of something to be specified. This is the reason for the other form of identifier. For example, `ident(reach)` means the identifier `reach`, not the Larch/C++ keyword `reach`. (Note that no whitespace is allowed between the parts of such an identifier.) This syntax allows one to use the C++ macro defined in the following, so that C++ will not gag on this form of identifier.

```c
// @(#)Id: ident_macro.lh,v 1.2 1997/05/25 00:19:37 leavens Exp $
#ifndef __ident_macro_lh__
#ifndef ident_macro_lh
#define ident_macro_lh
#define ident(x) x
#endif
#endif
```

Identifiers may live in several different worlds in Larch/C++. There are 3 different such worlds: trait names, names of specified C++ entities, and all other names. The last world includes trait function names (including prefix, infix, and postfix trait functions) and sort names that are not type names. The same identifier may thus be used in these worlds with different meanings. For example, in Larch/C++, `int` is both the name of a trait and the name of a C++ type.

### 4.7 Simple-Ids

A simple-id has the same syntax as an identifier, but is only used for names of LSL traits or formals that denote such traits.

\[ identifier ::= \text{letter} [ \text{letter-or-digit} ] . . . \]

### 4.8 Keywords

Several identifiers are reserved for use as keywords in Larch/C++. (See Section 4.6 [Identifiers], page 39, however, for a mechanism for using a Larch/C++ keyword as an identifier.) To be accurate, however, not all such identifiers are always recognized by Larch/C++ as keywords; only an always-keyword is recognized as a keyword everywhere.

\[ keyword ::= \text{term-only-keyword} | \text{stmt-exp-only-keyword} \]
\[ | \text{always-keyword} \]
The following keywords are only recognized as keywords in places where a Larch/C++ predicate or term (see Section 6.1 [Predicates], page 86) can appear. Therefore, within a C++ statement or expression each is recognized as an identifier. (The keyword on, while not technically appearing within a term, is treated the same way.)

\[
\text{term-only-keyword} ::= \text{fresh} \mid \text{informally} \mid \text{liberally} \\
\mid \text{on} \mid \text{nothing} \mid \text{post} \\
\mid \text{pre} \mid \text{redundantly} \mid \text{result} \\
\mid \text{returns} \mid \text{self} \mid \text{then} \\
\mid \text{thrown} \mid \text{throws} \mid \text{trashed} \\
\mid \text{unchanged}
\]

The following keywords are not recognized as keywords within a Larch/C++ term or predicate; in such places they are recognized as identifiers. Outside of a term, they are recognized as keywords, and can thus be used as keywords within a C++ statement or expression.

\[
\text{stmt-exp-only-keyword} ::= \text{break} \mid \text{case} \mid \text{catch} \\
\mid \text{const\_cast} \mid \text{continue} \mid \text{default} \\
\mid \text{delete} \mid \text{do} \mid \text{dynamic\_cast} \\
\mid \text{goto} \mid \text{new} \mid \text{reinterpret\_cast} \\
\mid \text{return} \mid \text{static\_cast} \mid \text{switch} \\
\mid \text{throw} \mid \text{try} \mid \text{typeid} \\
\mid \text{while}
\]

The following keywords are recognized as keywords in every context. Note that the C++ keywords if, else, and for are used differently within specification terms in Larch/C++ than in C++.
always-keyword ::= abstract | accesses | also
| any     | asm    | assert |
| auto    | be     | behavior |
| by      | calls  | char   |
| class   | constraint | constructs |
| const   | depends | double |
| else    | ensures | enum   |
| everything | example | expects |
| explicit | extern | float  |
| for     | friend | if     |
| inline  | int    | invariant |
| is      | let    | long   |
| modifies | mutable | namespace |
| operator | private | program |
| protected | public | reach |
| refine  | register | represents |
| requires | satisfies | short |
| signature | signed | simulates |
| sizeof  | spec   | static |
| struct  | template | this |
| throw   | trashes | typedef |
| typename | union | unsigned |
| uses    | using  | virtual |
| void    | volatile | wchar_t |
| weakly  | where  | with   |

See Section 4.15 [Alternative Tokens], page 52, for other reserved words, used to support limited character sets.

### 4.9 Context-Dependent Keywords

As in C++, new context-dependent keywords are introduced by various declarations. We use a few more categories of such keywords than the proposed standard grammar does (see Section A.1 of [ANSI95]), because we want to distinguish type definition names and template names for classes from non-classes. Also, note that an original-class-name is the name of something that is a declared as a class. See Section 5.2.3.2 [Class and Namespace Names], page 61 for the syntax of class-name and namespace-name. See Section 4.6 [Identifiers], page 39 for the syntax of identifier.
context-dependent-keyword ::= typedef-non-class-or-enum-name
  | typedef-class-name | typedef-enum-name
  | original-namespace-name | namespace-alias-name
  | original-class-name | original-enum-name
  | template-non-class-name | template-class-name
typedef-non-class-or-enum-name ::= identifier
typedef-class-name ::= identifier
typedef-enum-name ::= identifier
original-namespace-name ::= identifier
namespace-alias-name ::= identifier
original-class-name ::= identifier
original-enum-name ::= identifier
template-non-class-name ::= identifier
template-class-name ::= identifier

An identifier is recognized as a typedef-class-name if it was declared (in an appropriate scope) by a declaration using the typedef decl-specifier (see Section 5.2 [Declaration Specifiers], page 57), and if it declares a class. Similarly, an identifier is recognized as a typedef-enum-name if it was declared as a typedef to an enum. Other identifiers declared in a typedef declaration are recognized as typedef-non-class-or-enum-names when used later. The same distinction is made in template declarations (see Chapter 8 [Template Specifications], page 239) between a template-non-class-name and a template-class-name.

Similarly, an identifier is recognized as an original-enum-name if it was declared with an enum-specifier (see Section 5.3 [Enumeration Declarations], page 64). Again, an identifier is recognized as an original-class-name if it was declared with a class-specifier (see Chapter 7 [Class Specifications], page 167). Similar remarks hold for identifiers recognized as original-namespace-names or namespace-alias-names (see Section 5.5 [CPP Namespace and Using Declarations], page 78).

4.10 Special Symbols

There kinds of special symbols that Larch/C++ recognizes vary with what is being parsed. Details are given in the following subsections.

special-symbol ::= always-special-symbol
  | C++-decl-symbol | C++-operator-symbol
  | predicate-special-symbol | lsl-op
4.10.1 Always Special Symbols

The following are used as punctuation symbols in Larch/C++, regardless of context. Of course, they may have different meanings in different contexts. (See Section 4.15 [Alternative Tokens], page 52, for synonyms for {, }, [, and ].)

\[
\text{always-special-symbol ::= ( | ) | { | } | [ | ] | = | ; | : | :: | , | ? | . | .* | \ldots'}
\]

4.10.2 C++ Declaration Symbols

The following symbols are used as special symbols in most contexts, but not within a predicate or term (see Section 6.1 [Predicates], page 86), where are instead recognized as lsl-op symbols, (see Section 4.10.5 [LSL Operators], page 45).

\[
\text{C++decl-symbol ::= < | > | * | & | ~}
\]

(See Section 4.15 [Alternative Tokens], page 52, for synonyms for &, and ~.)

4.10.3 C++ Operator Symbols

The following are special within the declaration of a C++ operator (see Section 5.4 [Declarators], page 65), and within a C++ statement or expression.

\[
\text{C++operator-symbol ::= new | delete}
\]

See Section 4.15 [Alternative Tokens], page 52 for alternatives for the characters &, l, ~, !, ^, [ and ] used in such operators. The alternative token correspondence is the same as in Section r.2.4 of [Stroustrup95].
4.10.4 Predicate Special Symbols

The predicate-special-symbol characters are used as punctuation symbols within a predicate or term (see Section 6.1 [Predicates], page 86). See Section 6.2.1 [State Functions], page 103 for the meaning of the first two of these symbols; the second two are standard LSL symbols.

\[ \text{predicate-symbol} ::= \neg | \prime | \langle | \rangle \]

Larch/C++ recognizes a single quote (’) as a predicate-symbol instead of as the beginning of a character-constant when it is not immediately followed by a single char-const-char and another single quote (see Section 4.13.3 [Character Constants], page 49).

4.10.5 LSL Operators

In LSL, a trait may introduce infix, prefix, and postfix trait function symbols. These are recognized as such in Larch/C++ predicates and terms (see Section 6.1 [Predicates], page 86). The following grammar details how such symbols are made into tokens. Briefly, one can use strings of operator characters (such as `!==? and `<>`, or symbols that start with a backslash (such as `\U` and `\union`). The token extends as far to the right as possible, so sometimes white space must be used to delimit such tokens.

See Section 4.11 [Keywords in Predicates], page 46 for a list of such forms that are reserved by Larch/C++.

\[ \text{lsl-op} ::= \backslash \text{identifier} | \star-or-op-char \ [ \text{op-char} ] \ldots \]
\[ \star-or-op-char ::= \ast | \text{op-char} \]
\[ \text{op-char} ::= \neg | = | < | > | + | - | / \]
\[ | ! | | \# | $ | & | ? | @ | ' | \]

Note that in the definition of op-char, the `|` in `'| above is not a meta symbol, but a terminal representing the symbol `|`.

Many C++ operator symbols, for example, `&`, `||`, `++`, `<`, `>`, `->`, `->*`, are parsed as lsl-ops within a predicate or term. These can be defined by the user to mean various things, as usual in LSL.

The tokens `&` and `~` may be written as `bitand` and `compl`, respectively. This follows the proposed C++ standard's support for international character sets [Stroustrup95] [ANSI95] (see Section 4.15
Chapter 4: Lexical Conventions

[Alternative Tokens], page 52). However, tokens consisting of more than one op-char, such as \&\& or ~~, have no such equivalent. (But \(!=\) can be written as not_eq, see Section 6.1.3 [Equality Terms and Quantifiers], page 87).

In LSL one can customize such things as the set of operator characters, and which characters are opening and closing bracket symbols. Such freedom is not allowed in Larch/C++, where these sets are fixed to the defaults in LSL version 3.1. When you wish a special symbol, it is best to use one that starts with a backslash (such as \(\backslash\)). Furthermore, although LSL allows names that do not start with a backslash (\(\backslash\)) to be used as prefix, infix, or postfix operators, Larch/C++ does not recognize a name such as intersection as an lsl-op. Hence, for prefix, infix, and postfix operators you should also use a name beginning with a backslash.

4.11 Keywords in Predicates

The following forms are recognized as keywords within a LSL (and Larch/C++) predicate or term (see Section 6.1 [Predicates], page 86). Since they have special meaning in Larch/C++, those that would otherwise be lsl-op (see Section 4.10.5 [LSL Operators], page 45) may not be used as such within predicates and terms.

\[
predicate-keyword ::= \\backslash A | \\backslash and | \\backslash any | \\backslash E | \\backslash eq | \\backslash exists | \\backslash forall
| \\backslash implies | \\backslash angle | \\backslash neq | \\backslash obj | \\backslash or | \\backslash pre | \\backslash post | \\backslash range
| = | == | != | =\neq | \\backslash \slash | \\backslash \backslash | =>
\]

The predicate-keywords \(\angle, \rangle\), and each form that Larch/C++ uses as a logical-opr (see Section 6.1.2 [Logical Connectives], page 87) may be given different meanings for different sorts in a LSL trait. The meaning of the others is fixed either by LSL or by Larch/C++.

4.12 Informal Comments

Informal comments, are not really comments, but do resemble C-style comments. They look like \%( ... %\), where the body of the comment can contain any characters except the string \%(\). These tokens are recognized only within a LSL (and Larch/C++) predicate or term (see Section 6.1 [Predicates], page 86). See Section 6.1.4 [Informal Descriptions], page 91 for how these are used.
informal-comment ::= (% | informal-comment-body | %)
informal-comment-body ::= non-percent-right-paren [ non-percent-right-paren ] ... 
non-percent-right-paren ::= non-percent
  | percents-non-right-paren 
non-percent ::= any character except %
percents-non-right-paren ::= % [ % ] ... non-right-paren 
non-right-paren ::= any character except )

4.13 Literals

All the C++ literals are built-in to Larch/C++. A literal can be viewed as a trait function (with zero arity) of the sort which the literal’s type is based on. For example, one can view the integer constant 27 as a trait function with the signature 27: -> int. As in C++, adjacent string-literals are concatenated into a single string (see section 2.5.4 of [Ellis-Stroustrup90]).

literal ::= integer-constant | floating-constant 
  | character-constant | string-literal [ string-literal ] ... 
  | abstract-string-literal

4.13.1 Integer Constants

Larch/C++ integer constants are exactly the same as those of C++. See Section 4.6 [Identifiers], page 39 for the lexical syntax of digit.

integer-constant ::= decimal-constant | octal-constant | hex-constant
decimal-constant ::= one-to-nine [ digit ] ... [integer-suffix]
one-to-nine ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
octal-constant ::= 0 [ octal-digit ] ... [integer-suffix]
octal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex-constant ::= 0 hex-indicator hex-digit [ hex-digit ] ... [integer-suffix]
hex-indicator ::= x | X
hex-digit ::= digit | a | b | c | d | e | f | A | B | C | D | E | F
integer-suffix ::= long-suffix | unsigned-suffix 
  | long-suffix unsigned-suffix | unsigned-suffix long-suffix
long-suffix ::= 1 | L
unsigned-suffix ::= u | U
Some examples of \textit{integer-constants} follow.

\begin{verbatim}
227
228U
0342L
0xFFFFFFFFul
\end{verbatim}

The meaning of an \textit{integer-constant} is as in C++ (See Section 2.5.1 of \cite{Ellis-Stroustrup90}), except that the sort of an \textit{integer-constant} is always \texttt{int}, unless a suffix is present. This makes the specification independent of a particular C++ implementation of integers.

\subsection{4.13.2 Floating Constants}

Larch/C++ floating constants are the same as C++ floating numbers (see Section 2.5.3 of \cite{Ellis-Stroustrup90}).

\begin{verbatim}
floating-constant ::= fractional-constant [exponent-part] [float-suffix]
    | digit [ digit ] \ldots exponent-part [float-suffix]
fractional-constant ::= [ digit ] \ldots digit [ digit ] \ldots
    | digit [ digit ] \ldots exponent-part ::= exponent-indicator [ sign ] digit [ digit ] \ldots
exponent-indicator ::= e | E
sign ::= - | +
float-suffix ::= f | F | l | L
\end{verbatim}

Some examples of \textit{floating-constant} follow.

\begin{verbatim}
123.456
.123
123.
1234.456e-7
12345E7
1234.456e+7F
1234eE7L
\end{verbatim}

A \textit{floating-constant} has sort \texttt{double}, unless the type is explicitly specified by a suffix. The suffixes \texttt{f} and \texttt{F} mean \texttt{float} while the suffixes \texttt{l} and \texttt{L} mean \texttt{long double}. 
Chapter 4: Lexical Conventions

4.13.3 Character Constants

Larch/C++ character constants are the same as C++ (single) character constants.

\[\text{character-constant ::= [ L ] ' char-const-char '}
\]
\[\text{char-const-char ::= normal-char | std-esc | ''}
\]
\[\text{normal-char ::= any character except ', '', \ or a newline}
\]
\[\text{std-esc ::= \n } // \text{ newline} \LF
\]
\[\text{ | \t } // \text{ horizontal tab} \HT
\]
\[\text{ | \v } // \text{ vertical tab} \VT
\]
\[\text{ | \b } // \text{ backspace} \BS
\]
\[\text{ | \r } // \text{ carriage return} \CR
\]
\[\text{ | \f } // \text{ form feed} \FF
\]
\[\text{ | \a } // \text{ alert} \BEL
\]
\[\text{ | \\ } // \text{ backslash} \\\
\]
\[\text{ | \? } // \text{ question mark} ?
\]
\[\text{ | \'} // \text{ single quote} '
\]
\[\text{ | \" } // \text{ double quote} "
\]
\[\text{octal-code ::= octal-digit | octal-digit octal-digit}
\]
\[\text{ | octal-digit octal-digit octal-digit}
\]

For example, ‘l’, ‘\t’, and ‘\032’ are character constants. The sort of a single character constant is char. The meaning of a character constant such as 'a' is not uniquely defined by C++ but depends on “the machine’s character set” (section 2.5.2 [Ellis-Stroustrup90]). Therefore in Larch/C++, 'a' has sort char, but its numerical value is not uniquely defined. A constant that starts with L, for example L’a’, is a wide character constant of sort wchar_t. Larch/C++ does not support multicharacter constants, as they are not well-defined. (See Section 2.5.2 of [Ellis-Stroustrup90] for a description of multicharacter constants.)

Larch/C++ supports the standard C++ escape sequences. The meaning of such a std-esc is as in C++. In an escape sequence, the octal (or hexadecimal) digit string ends with the first character that is not an octal-digit (or a hex-digit) (see Section 2.5.2 of [Ellis-Stroustrup90]).
### 4.13.4 String Constants

The syntax of string constants in Larch/C++ includes all of the syntax for C++ string literals (see Section 2.5.4 of [Ellis-Stroustrup90]). It also allows some non-standard escape sequences. See Section 4.13.3 [Character Constants], page 49 for details of `std-esc`.

\[
\begin{align*}
\text{string-literal} &::= [ \text{L} ] " [ \text{string-character} ] \ldots " \\
\text{string-character} &::= \text{normal-char} \mid \text{escape-sequence} \mid \text{'}' \\
\text{escape-sequence} &::= \text{std-esc} \mid \text{non-std-esc} \\
\text{non-std-esc} &::= \backslash \\text{non-escape-code} \\
\text{non-escape-code} &::= \text{any character that cannot follow \ in a std-esc}
\end{align*}
\]

Some examples of `string-literals` follow.

```
"
"the previous line contains the empty string"
"the standard escape characters are:\n\t\v\b\f\a\\?\'"\032\xff"
"\in Larch/C++, use non-standard escapes \ not /\ it's okay"
L"the line above means the following"
"\\\in Larch/C++, use non-standard escapes \\ not /// it’s okay"
```

The meaning of a `non-std-esc`, such as \g is the character backslash followed by the next character (i.e., the meaning of \ \ followed by the meaning of g). Thus \ \ means the same thing as \ ///. This is a convenience in writing informal descriptions (see Section 6.1.4 [Informal Descriptions], page 91), where one may want to use \ \ and \ /// to mean “or” and “and”. (However, one must be careful, not to put \ /// just before the closing double quote of such a string.)

In Larch/C++, a distinction is made between array types and pointer types (as in LCL [Guttag-Horning93]). So unlike C++, a `string-literal` in Larch/C++ has sort `Arr[Obj[char]]` (which is considered different from `Ptr[Obj[char]]`).\(^1\) A string constant preceded by L is a wide-character string, and has sort `Arr[Obj[wchar_t]]` (see Section 2.5.4 of [Ellis-Stroustrup90]). Note the difference between a character constant and a string that contains a single character: 'x' is not the same as "x". The former is a single character of sort `char` while the latter is a string constant of sort `Arr[Obj[char]]`.

---

\(^1\) Although the C++ standard does not define the effect of modifying a string-literal, it does state that the type is not an array of constant char objects (`const char[]`), but just “array of char”, i.e., `char[]`. 
See Section 11.9 [Character Strings], page 296 for some traits that are helpful in specifying functions dealing with C++ strings.

### 4.13.5 Abstract String Constants

In specifications, one sometimes needs a constant of sort `String[char]`. Such constants are provided by the syntax for an `abstract-string-literal`. The syntax for such an `abstract-string-literal` is nearly the same as a C++ string literal (see Section 4.13.4 [String Constants], page 50) except that it has a leading `A`.

\[
\text{abstract-string-literal} ::= \text{A} \ [ \text{string-character} ] \ldots \text{"}
\]

Some examples of `abstract-string-literal`s follow.

\text{A}"
\text{A}"the previous line contains the empty abstract string"

Such constants are sometimes useful in writing down examples in function specifications (see Section 6.9.1 [Examples in Function Specifications], page 137).

The sort `String[char]` is defined in the trait `AbstractStringTrait`, which follows.

\% @(#)$Id: AbstractStringTrait.lsl,v 1.2 1995/06/29 15:42:12 leavens Exp $
AbstractStringTrait: trait
  includes int, String(char, String[char], int for Int)

### 4.14 LSL Constants

In a renaming (see Section 9.2 [The Uses Clause], page 255), one can pass to LSL an `lsl-constant` of the following form.

\[
lsl-constant ::= \text{decimal-constant} | \text{character-constant}
\]
4.15 Alternative Tokens

As in the coming C++ standard [Stroustrup95] [ANSI95], support for international character sets (i.e., keyboards without the full suite of ASCII characters) is provided in Larch/C++.

The following table gives the alternative tokens, and the primary form of the token to which they are equivalent. Only the primary form is used in the syntax in this manual, but the alternatives are recognized by the Larch/C++ tools in all contexts. (The \_\_wchar\_t token is a concession to GNU C++.)

<table>
<thead>
<tr>
<th>Alternative</th>
<th>Primary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;%)</td>
<td>{</td>
</tr>
<tr>
<td>(%)&gt;</td>
<td>}</td>
</tr>
<tr>
<td>(&lt;)</td>
<td>[</td>
</tr>
<tr>
<td>(&gt;)</td>
<td>]</td>
</tr>
<tr>
<td>(\text{bitand})</td>
<td>&amp;</td>
</tr>
<tr>
<td>(\text{compl})</td>
<td>~</td>
</tr>
<tr>
<td>(\text{not_eq})</td>
<td>!=</td>
</tr>
<tr>
<td>(\text{__wchar_t})</td>
<td>wchar_t</td>
</tr>
</tbody>
</table>

In the definition of a C++ operator function interface, and in C++ statements and expressions, but not in a predicate or term (see Section 6.1 [Predicates], page 86), the following are recognized.

<table>
<thead>
<tr>
<th>Alternative</th>
<th>Primary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{and})</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>(\text{bitor})</td>
<td>|</td>
</tr>
<tr>
<td>(\text{or})</td>
<td></td>
</tr>
<tr>
<td>(\text{xor})</td>
<td>~</td>
</tr>
<tr>
<td>(\text{and_eq})</td>
<td>&amp;=</td>
</tr>
<tr>
<td>(\text{or_eq})</td>
<td></td>
</tr>
<tr>
<td>(\text{xor_eq})</td>
<td>%=</td>
</tr>
<tr>
<td>(\text{not})</td>
<td>!</td>
</tr>
</tbody>
</table>
In a predicate or term, one can use as tokens the LSL symbols \ for “or” and \ for “and”. For negation, use compl, which works as a synonym for ~ in all contexts. In predicates and terms, various trait functions provide synonyms for !. In predicates and terms, there is no standard symbol for “exclusive-or”; the token ~ is used in predicates and terms to mean “the pre-state value of” (see Section 6.2.1 [State Functions], page 103), but it has a synonym \pre.
5 Declarations

In a Larch/C++ specification, a declaration plays two roles. As usual, it introduces one or more names into a specification and gives information about the name’s type and other attributes, which are used in giving meaning to occurrences of the name. But in addition a declaration specifies that the C++ module that implements the specification must have the same declaration. (The only exceptions are ghost variable declarations, see Section 9.1 [Ghost Variables], page 254, and the declarations in the declaration-seq of a Larch/C++ function specification, see Section 6.7 [Global Variables], page 134.) This is in accord with the principle of behavioral interface specification (see Section 2.2 [Interfaces], page 15). Therefore, by giving a declaration in Larch/C++, one can specify the storage class, type, and linkage for an object, and the type and linkage for functions. One can think of the declaration as going directly into the C++ code. (This is easily accomplished if the specification is given as annotations in a C++ header file.) Because of this, the declaration must satisfy the restrictions on C++ code in Section r.7 of [Stroustrup91].

The syntax of a declaration in Larch/C++ is nearly identical to the syntax of a declaration in C++. The only difference is that additional forms that are useful in specifications are permitted, and specification information can be attached to various productions. In particular, Larch/C++ adds the ability to declare the behavior of functions (see Chapter 6 [Function Specifications], page 82) and classes (see Chapter 7 [Class Specifications], page 167). Larch/C++ also adds the ability to pass traits to templates (see Chapter 8 [Template Specifications], page 239) and higher-order functions (see Section 6.13 [Specifying Higher-Order Functions], page 157).

Thus the Larch/C++ syntax is a superset of the C++ syntax for declarations. The Larch/C++ syntax is given below.
Chapter 5: Declarations

```
declaration ::= [ decl-specifier-seq ] [ init-declarator-list ] ; [ fun-spec-body ]
| [ decl-specifier-seq ] ctor-declarator [ fun-spec-body ]
| block-declaration
| function-definition
| template-declaration
| explicit-instantiation
| explicit-specialization
| linkage-declaration
| namespace-definition
| refinement-declaration
| extern everything ;
block-declaration ::= asm-definition
| namespace-alias-definition
| using-declaration
| using-directive
init-declarator-list ::= init-declarator [ , init-declarator ] …
init-declarator ::= declarator [ initializer ]
```

Other sections of this chapter give more information on the parts of a standard C++ declaration: decl-specifier-seq, declarator, and initializer. The decl-specifier-seq give the type and other information about the thing being declared. The declarator gives the name of the thing being declared, and may also give information about whether the thing being declared is a pointer, array, or function. An initializer allows one to declare an initial value.

Other parts of the C++ declaration syntax that recursively use declarations are treated in subsections of this chapter: linkage-declaration, namespace-definition, and namespace-alias-definition. See Section 5.4.6 [Function Declarations], page 72 for the syntax and semantics of function-definition and ctor-declarator. See Chapter 6 [Function Specifications], page 82 for the syntax and semantics of fun-spec-body. See Section 6.7 [Global Variables], page 134 for the meaning of a declaration of the form extern everything;, which is used in function specifications. See Chapter 8 [Template Specifications], page 239 for the syntax and semantics of template-declaration. See Chapter 10 [Refinement], page 257 for the syntax and semantics of refinement-declaration.

For example, after the initial comment, each (nonblank) line and group of lines of the following is an example of a declaration.

```
// @(#)$Id: declaration.lh,v 1.8 1997/06/03 20:30:01 leavens Exp $
int i;
int j = 3, k = 4; // declaration with initializers
int *ip = &i;
extern double Sqrt(double x);
```
5.1 Initializers

In a \textit{init-declarator-list}, one can specify an initial value for a variable just as in C++. The syntax for an \textit{expression}, \textit{constant-expression}, and \textit{assignment-expression} is exactly as in C++ (see Sections 17.2 of [Ellis-Stroustrup90] and [ANSI95]), and so is not given here.

\begin{verbatim}
initializer ::= = constant-expression | = { initializer-list }
   | ( expression-list )
constant-expression ::= exactly as in C++
initializer-list ::= initializer-clause [ , initializer-clause ] ... [ , ]
initializer-clause ::= assignment-expression
   | { [ initializer-list ] }
assignment-expression ::= exactly as in C++
expression-list ::= expression [ , expression ] ... 
expression ::= exactly as in C++
\end{verbatim}

For example, one can use each of the following lines as an \textit{initializer}.
= 7
= 6.5e-7
= 'c'
= "a string"

If one wishes to abstractly specify the initial value, instead of giving C++ code, then one can write a `fun-spec-body` to specify the value. For example, if the class `IntHeap` has been specified with abstract values from trait `IntHeapTrait` (see Section 1.1 [Larch-style Specifications], page 2), then one can write a declaration such as the following to specify the initialization of the variable `myHeap` to the abstract value `add(3, (add 4, empty))`.

```c++
#include "IntHeap.lh"
IntHeap myHeap;
//@ behavior { ensures myHeap' = add(3, add(4, empty)); }
```

### 5.2 Declaration Specifiers

The syntax of declaration specifiers is as in C++. This is a correction from previous versions of Larch/C++ which restricted the order of `decl-specifier-seq`. Note that the new ANSI standard no longer supports the "implicit int" rule of C. See Section 5.2.3 [Type Specifiers], page 59 for the syntax of `type-specifier-seq`.

\[
decl-specifier ::= storage-class-specifier
| type-specifier | function-specifier
| friend | typedef
\]
\[
decl-specifier-seq ::= decl-specifier [ decl-specifier ] …
\]

For example, one can write `decl-specifier-seqs` such as the following.
int
extern int
static double
friend void
inline double
typedef int
unsigned long int

5.2.1 Storage Class Specifiers

The syntax of storage class specifiers is the same as in C++. (The keyword mutable is described in Section r.7.1.1 of [Stroustrup95]. It should only be applied to non-const and non-static data members of a class.)

storage-class-specifier ::= static | extern | mutable | auto | register

Declarations marked static are not made available by an import in Larch/C++.

Note that an object declaration such as

int x;

is a definition, but the addition of extern, as in the following,

extern int x;

turns it into a declaration (see Section 2.5 [Declarations and Definitions], page 18).

The linkage (see Section r.3.3 and r.7.1 of [Stroustrup91]) specified by static and extern is as in C++.

The only use for register would be to record an implementation design decision.
5.2.2 Function Specifiers

The following is the syntax of function-specifiers.

\[ \text{function-specifier} ::= \text{virtual} \mid \text{inline} \mid \text{explicit} \]

The virtual specifier may be used only in declaration of a nonstatic class member function within a class specification (see Section 7.1.2 of [Stroustrup91]). To specify a pure virtual member function in an abstract class, one should use an initializer of the form =0 (see Section 10.3 of [Ellis-Stroustrup90]). A class with any pure virtual functions is an abstract class. See Section 7.10 [Abstract Classes], page 234 for more information about abstract classes.

The inline specifier should only be used when one desires to record an implementation design decision, as it does not usually concern the clients of a C++ function. A C++ function satisfies a specification that does not use inline regardless of whether inline appears in the implementation. However a C++ function satisfies a specification that uses inline only if the function also uses inline, or a sugar for it such as giving the definition of a member function in the class declaration (Section 9.3.2 [Ellis-Stroustrup90]).

5.2.3 Type Specifiers

The syntax of type specifiers is as in C++ (see Section r.7.1.6 of [Stroustrup91]).

\[ \text{type-specifier-seq} ::= \text{type-specifier} \left[ \text{type-specifier} \right] \ldots \]
\[ \text{type-specifier} ::= \text{simple-type-name} \]
\[ \mid \text{enum-specifier} \mid \text{class-specifier} \]
\[ \mid \text{elaborated-type-specifier} \mid \text{cv-qualifier} \]
\[ \text{cv-qualifier} ::= \text{const} \mid \text{volatile} \]

Usually only one type-specifier may be given in a declaration. But const or volatile may be added to the others (see Section r.7.1.6 of [Stroustrup91]), and some combinations of built-in type names may be used to make types such as unsigned int. In the following, each line is a legal type-specifier-seq.
The Larch/C++ meaning of `const` is discussed below (see Section 5.4.7 [Constant Declarations], page 75).

The type specifier `volatile` says that the value of an object may be changed by concurrent processes, etc.

The `type-specifier` may be omitted from a declaration, and defaults to `int`. However, omitting the type specification is bad style in a Larch/C++ specification. This default is also slated to be removed from C++, and hence from Larch/C++. In short, do not use this default.

See Chapter 7 [Class Specifications], page 167 for details on the `class-specifier`. See Section 5.3 [Enumeration Declarations], page 64 for details on the `enum-specifier`.

### 5.2.3.1 Simple Type Names

The syntax of `simple-type-name` is as in C++. The nonterminals `typedef-non-class-or-enum-name`, `typedef-enum-name`, and `original-enum-name` are all previously-declared identifiers (see Section 4.9 [Context-Dependent Keywords], page 42). See Section 5.2.3.2 [Class and Namespace Names], page 61 for the syntax of `complete-class-name` and `nested-name-specifier`. 

```
double               // simple type names
MyClass
unsigned int
long double
const int            // cv-qualifier + simple type name
const unsigned int
struct myStruct { int x,y; } // class-specifier
enum color {red,white,blue}  // enum-specifier
struct MyStruct       // elaborated-type-specifiers
enum color
typename MyStruct
```
simple-type-name ::= complete-type-name
    | [ :: ] nested-name-specifier template template-class-instance
    | built-in-type-name

complete-type-name ::= complete-class-name
    | complete-non-class-type-name

complete-non-class-type-name ::= [ :: ] [ nested-name-specifier ] non-class-type-name

non-class-type-name ::= enum-name | typedef-non-class-or-enum-name

built-in-type-name ::= char | short | int | long | signed | unsigned | float | double
    | bool | void

enum-name ::= original-enum-name | typedef-enum-name

See Section r.7.1.6 of [Stroustrup91] for restrictions on the use of long, short, double, signed, unsigned, and char.

Each of the built-in types of C++ is automatically associated with a trait of the same name in Larch/C++. This trait defines the abstract values of the C++ values of such types. For example, the abstract values int is modeled by abstract values of sort int in trait int. See Chapter 11 [Built-in Types], page 269 for the details on how the abstract values of these types are modeled in LSL.

### 5.2.3.2 Class and Namespace Names

The syntax of a complete-class-name, class-name, and a complete-namespace-name, is as in the proposed C++ standard [ANSI95]. The nonterminals original-class-name, original-namespace-name, and namespace-alias-name are all previously-declared identifiers (see Section 4.9 [Context-Dependent Keywords], page 42).

complete-class-name ::= [ :: ] [ nested-name-specifier ] class-name

complete-namespace-name ::= [ :: ] [ nested-name-specifier ] namespace-name

nested-name-specifier ::= class-name :: | namespace-name ::
    | nested-name-specifier class-name ::
    | nested-name-specifier namespace-name ::
    | nested-name-specifier template template-class-instance ::

class-name ::= original-class-name | typedef-class-name | template-class-instance

namespace-name ::= original-namespace-name | namespace-alias-name

The scope resolution operator, ::, has the same meaning as in C++ (see Section r.3.2 of [Stroustrup91]).
For example, if `TerminalUtilities` is a `original-namespace-name` containing the declarations of `TerminalControllerClass` and `TermControl`, if `TerminalControllerClass` is an `original-class-name`, if `TermControl` is a `typedef-class-name`, and if `Set<int>` is a `template-class-instance`, then each line of the following is a `complete-class-name`.

```
Set<int>
::Set<int>
TerminalUtilities::TermControl
::TerminalUtilities::TermControl
TerminalUtilities::TerminalControllerClass
```

However, `TermControl` is not a `complete-class-name` except within the scope of the declaration of `TerminalUtilities`.

Continuing the above example, each line of the following is a `complete-namespace-name`.

```
TerminalUtilities
::TerminalUtilities
```

Continuing the above example, each line of the following is a `nested-name-specifier`.

```
Set<int>::
TerminalUtilities::
TerminalUtilities::TermControl::
TerminalUtilities::TerminalControllerClass::
```

### 5.2.3.3 Elaborated Type Specifiers

The syntax of `elaborated-type-specifier` is the same as in C++ (the use of `typename` is found in the coming C++ standard [ANSI95] [Stroustrup95]), with the addition of signatures [Baumgartner-Russo95], which are used in Larch/C++ to help specify requirements on template parameters (see Chapter 8 [Template Specifications], page 239).

```
elaborated-type-specifier ::= class-key [ :: ] [ nested-name-specifier ] identifier
| enum [ :: ] [ nested-name-specifier ] identifier
| typename [ :: ] nested-name-specifier identifier
| typename [ :: ] nested-name-specifier [ template ] template-class-instance
class-key ::= class | struct | union | signature
```
An *elaborated-type-specifier* can by used instead of just the *identifier* either for emphasis, or because the *identifier* has been used in another declaration that would otherwise hide the name of the type. For example in the following, a class *Foo* is declared, along with a variable of that class of the same name. The variable name makes a hole in the scope of the class name, hence the *elaborated-type-specifier* must be used to name the class in the following code.

```c
// @(#)$Id: elaborated.lh,v 1.1 1997/01/10 23:35:29 leavens Exp $
class Foo { } Foo;
class Foo aFoo; // needs elaborated type specifier
```

### 5.2.4 Friend

As in C++, a *friend* specifier may only be used within the specification of a class. It may only be used in the declaration of a function or a class. The *friend* specifier says that the function or class named in the declaration has access to the private members of the class in which it occurs. (This is automatic if you use annotations to a header file for your specification.) For example, one could put the declaration

```c
friend ostream& operator<<(ostream&, date);
```

in the class *date* to say that this overloading of the C++ “put to” operator is to have access to the private members of the class *date*. Such a declaration is implemented by a matching definition in the header file where the declaration of class *date* occurs. See Section 7.12 [Specifying Friends], page 237 for a discussion of when to grant friendship.

### 5.2.5 Typedef Specifiers

As in C++, a *typedef* declaration makes synonyms for types. For example, after the following declarations:

```c
typedef struct { double re, im; } complex;
typedef int (*pif)(int);
```

the names *complex* and *pif* can be used as the names of types (see Section 5.2.3 [Type Specifiers], page 59).
Since a `typedef` is an abbreviation, uses of the name defined in a `typedef` are semantically equivalent to the use of the type’s meaning. Sometimes this meaning will be, as in the case of the declaration of `complex` above, something that does not otherwise have a name.

In C++ and Larch/C++ there is no need to use `typedefs` to define `struct` or `union` types. So one should write the above declaration of `complex` as follows.

```c
struct complex { double re, im; }
```

### 5.3 Enumeration Declarations

The syntax of enumeration declarations is as in C++ (see Section 7.2 of [Stroustrup91]).

```c
enum-specifier ::= enum [ identifier ] { [ enumerator-definition-list ] }
enumerator-definition-list ::= enumerator-definition [, enumerator-definition] . . .
enumerator-definition ::= identifier [ constant-initializer ]
```

See Section 7.2 [Class Member Specifications], page 186 for the syntax of `constant-initializer`. The `constant-expressions` used in `constant-initializers` must be of an integral sort or the sort of some other enumerator [section 7.2, ANSI95].

The identifiers in an `enumerator-definition` are called enumerators. As in C++, the enumerators are constants in Larch/C++. If the `identifier` is used in the declaration of an `enum-specifier`, then the sort of the enumerators is that `identifier`, otherwise a unique sort name is generated for them. These new constants only have equal abstract values if they are defined with the same initializers (either explicitly or implicitly). For example, in the following:

```c
enum color { red, orange, yellow, green, blue, indigo, violet=5 };
enum day_of_week { sun=1, mon, tues, wed, thurs, fri, sat };```

the enumerators `indigo` and `violet` are equal. However, neither of them is equal to `thurs`, although they have the same values when converted to integers in C++ (see Section 7.2 of [Stroustrup91]).

By using the trait function `to_int`, an enumerator can be converted to its integer value. With the above example, `to_int(blue)` has value 4 and sort `int`, whereas `blue` has sort `color`.

For example, the trait for the `color` example above would be the following. See Section 11.1 [Integer Types], page 269 for the trait `int`. 
% @(#)Id: color_Trait.lsl,v 1.6 1997/06/03 20:29:59 leavens Exp $

color_Trait: trait
includes int, NoContainedObjects(color)
introduces red, orange, yellow, green, blue, indigo, violet: -> color
to_int: color -> int

asserts
color generated by red, orange, yellow, green, blue, indigo, violet
color partitioned by to_int: color -> int
equations
to_int(red) == 0;
to_int(orange) == 1;
to_int(yellow) == 2;
to_int(green) == 3;
to_int(blue) == 4;
to_int(indigo) == 5;
to_int(violet) == 5;

implies
\forall c1, c2: color
 (c1 = c2) == (to_int(c1) = to_int(c2))
converts to_int: color -> int

The LSL enumeration of shorthand (see page 49 of [Guttag-Horning93]) is not used in the general construction of such traits. However, in many examples it could be combined with the Handbook trait Enumeration (see page 165 of [Guttag-Horning93]) and some renamings to construct an equivalent trait.

See Section 11.6 [Enumeration Types], page 278 for the trait constructed for the day_of_week example.

The trait corresponding to an enumeration declaration is implicitly used in any specification module in which the declaration appears. (See Section 2.7 [Types and Sorts], page 20 for more details on using a trait in a specification.)

5.4 Declarators

In a declaration (see Chapter 5 [Declarations], page 54), a declarator declares a single object, function, or type. The optional declaration specifiers (see Section 5.2 [Declaration Specifiers], page 57) describe its type, storage class, and other attributes. A declarator gives it a name and may refine its type with operators such *, as described in Section r.8 of [Stroustrup91]. The following table, from page 132 of [Ellis-Stroustrup90], summarizes these operators.
### Operator Meaning

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Pointer</td>
</tr>
<tr>
<td>::*</td>
<td>Pointer to Member</td>
</tr>
<tr>
<td>&amp;</td>
<td>Reference</td>
</tr>
<tr>
<td>[]</td>
<td>Array</td>
</tr>
<tr>
<td>()</td>
<td>Function</td>
</tr>
</tbody>
</table>

The syntax of declarators and abstract declarators is as in C++, except that when declaring a function, a *param-qualifier* may also declare LSL traits that are expected. See Section 6.13 [Specifying Higher-Order Functions], page 157 for the syntax and use of the *expects-clause*. See Section 5.1 [Initializers], page 56 for the syntax of initialization within declarators. See Section 6.11 [Exceptions], page 148 for the syntax of *exception-decl*. See Section 5.2.3.2 [Class and Namespace Names], page 61 for the syntax of *complete-class-name* and *complete-namespace-name*. See Section 5.2.3 [Type Specifiers], page 59 for the syntax of *cv-qualifier*.

```
declarator ::= direct-declarator | ptr-operator declarator
direct-declarator ::= id-expression | direct-declarator declarator-qualifier
                      | ( declarator )
declarator-qualifier ::= [ [ constant-expression ] ] | param-qualifier
param-qualifier ::= ( [ parameter-declaration-clause ] [ expects-clause ] )
                    [ cv-qualifier-seq ] [ exception-decl ]
id-expression ::= unqualified-id | qualified-id
unqualified-id ::= identifier
                  | operator-function-id | conversion-function-id
                  | template-instance
qualified-id ::= nested-name-specifier | template | unqualified-id
operator-function-id ::= operator C++-operator-symbol
conversion-function-id ::= operator type-specifier-seq [ ptr-operator ]
conversion-type-id ::= type-specifier-seq [ conversion-declarator ]
conversion-declarator ::= ptr-operator [ conversion-declarator ]
ptr-operator ::= * [ cv-qualifier-seq ] | &
                 | [ :: ] nested-name-specifier * [ cv-qualifier-seq ]
cv-qualifier-seq ::= cv-qualifier [ cv-qualifier ] ...
abstract-declarator ::= ptr-operator [ abstract-declarator ]
direct-abstract-declarator ::= [ direct-abstract-declarator | declarator-qualifier
                      | ( abstract-declarator )
```
The semantics of each of these forms of declarator are described below.

In the context of a declaration, each declarator in a Larch/C++ specification associates a sort with an id-expression. This sort is based on the declared C++ type name, and sort generators are used to deal with the C++ concepts of variables, pointers, etc. With each such sort, there is an LSL trait that gives its meaning. For example, the name Obj[int] is used as the sort for integer objects. An integer object in Larch/C++ is either a global variable (declared as int i or int& i) or a reference parameter (declared as int& i). Note that non-reference parameters in C++ are passed by value, and so are not treated as objects by Larch/C++ (see Section 6.2.1 [State Functions], page 103).

To be more concrete, consider the following table.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Name</th>
<th>Its Sort (when used as a global variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 7;</td>
<td>i</td>
<td>Obj[int]</td>
</tr>
</tbody>
</table>

The sort of i, when used as a global variable, is Obj[int], which means that i is an object containing an int. An object may have a value in a given state; typically in Larch/C++ the states of interest are those just before or just after the C++ function being specified. As explained below (see Section 6.2.1 [State Functions], page 103), the value of i in the state before a function call is written i^, and value afterwards is written i'. The sort of both i^ and i' is int.

The semantics of object sorts (such as Obj[int]) are described using the trait MutableObj (see Section 2.8.1.1 [Formal Model of Mutable Objects], page 28), which is instantiated with a value sort (such as int). That is, when a global int variable is declared, Larch/C++ implicitly uses the trait MutableObj(int). Similarly, for a global declaration of a variable of type T, Larch/C++ implicitly uses the trait MutableObj(T). Exceptions to this rule are made for global declarations of structs and classes (see Section 5.4.4 [Structure and Class Declarations], page 69) and arrays (see Section 5.4.3 [Array Declarations], page 69).

The meaning of declarations using operators such as * is explained in the subsections below.

### 5.4.1 Reference Declarations

Reference variables are much like normal variables, when used as globals in Larch/C++. Consider the following table.
Chapter 5: Declarations

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Name</th>
<th>Its Sort (when used as a global variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 7;</td>
<td>i</td>
<td>Obj[int]</td>
</tr>
<tr>
<td>int &amp; ir = i;</td>
<td>ir</td>
<td>Obj[int]</td>
</tr>
</tbody>
</table>

As a reference variable, `ir` is an alias for `i`. Hence when used as a global variable it has the same sort as `i`, `Obj[int]`. Thus the same trait is included by Larch/C++, `MutableObj(int)` (see Section 2.8.1.1 [Formal Model of Mutable Objects], page 28). The sort of `ir` is `int`. References are treated differently in function specifications, however (see Section 6.1.8.1 [Sorts for Formal Parameters], page 94).

5.4.2 Pointer Declarations

Pointers are somewhat more complex than references. Consider the following.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Name</th>
<th>Its Sort (when used as a global variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 7;</td>
<td>i</td>
<td>Obj[int]</td>
</tr>
<tr>
<td>int * ip = &amp;i;</td>
<td>ip</td>
<td>Obj[Ptr[Obj[int]]]</td>
</tr>
</tbody>
</table>

The sort of `ip`, when used as a global variable, is `Obj[Ptr[Obj[int]]]`, which means that `ip` is an object containing a pointer to an integer variable. (That is, `ip` is a pointer variable.) The sort of `ip'` is `Ptr[Obj[int]]`, which means that `ip'` is a pointer value (address) that points to an integer object. The sort of `*(ip')` is `Obj[int]` as the `*` dereferences the pointer value (the address contained in `ip` in the state after the call). The sort of `(*(ip'))` is `int`, which is the integer value in the object pointed to by the address contained in `ip` in the state after the call.

A global variable declaration such as `T *tp;`, makes Larch/C++ implicitly use the following traits: `MutableObj(Ptr[Obj[T]])` (see Section 2.8.1.1 [Formal Model of Mutable Objects], page 28), and `Pointer(Obj, T)`. See Section 11.8 [Pointer Types], page 287 for details on the trait `Pointer` used to define the abstract values of pointers, and the instantiations used for various pointer types.

See Section 11.8 [Pointer Types], page 287 for the traits that define pointers to members. They are the same as the pointer traits, except that instead of using sorts of the form `Ptr[T]`, they use sorts of the form `PtrMbr[T]`. 
5.4.3 Array Declarations

Consider the following.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Name</th>
<th>Its Sort (when used as a global variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ai[3];</td>
<td>ai</td>
<td>Arr[Obj[int]]</td>
</tr>
</tbody>
</table>

The sort of $ai$, when used as a global variable, is $Arr[Obj[int]]$, which means that $ai$ is an array of integers. In C++, an array variable can be thought of as the name of a sequence of objects; however the array name is not itself an object. Thus $ai[0]$ is an object, and so $(ai[0])'$ is the value of the zero-th element of the array in the state after the specified function is called. The sort of $(ai[0])'$ is $int$, while the sort of $ai[0]$ is $Obj[int]$, which means that $ai[0]$ is an integer object.

Since $ai$ is not an object, strictly speaking $ai'$ should not have a meaning. However, as in LCL (see Chapter 5 of [Guttag-Horning93]), $ai'$ stands for an abstract value that maps each index $i$ of $ai$ to $(ai[i])'$. See Section 6.2.1 [State Functions], page 103 for details.

A declaration such as $int ai[3]$ makes Larch/C++ implicitly include the trait $Array(Obj[int], int)$. See Section 11.7 [Array Types], page 279 for details of the traits defining the abstract values of arrays, and the instantiations used for various array types.

5.4.4 Structure and Class Declarations

Consider the following C++ type definition.

```cpp
struct Entry { char *sym; int val; };
```

According to the semantics of C++, one can equivalently also write the following.

```cpp
class Entry {public: char *sym; int val; };
```

The sort of a global variable $e$ of this type is $ConstObj[Entry]$.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Name</th>
<th>Its Sort (when used as a global variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry e;</td>
<td>e</td>
<td>$ConstObj[Entry]$</td>
</tr>
</tbody>
</table>
That $e$ has sort `ConstObj[Entry]` means that it is an object whose abstract value cannot be changed. This may seem strange, but the explanation reveals a lot about how Larch/C++ automatically models C++ objects.

A picture of how C++ stores $e$ in memory follows. In this picture, the crudely-drawn box and the two sub-boxes represent locations. The point is that $e$ and $e.sym$ share the same address in memory.

```
!-----------------------------------!
 e: ! ! !
!-----------------------------------!
 e.sym e.val
```

A picture of the storage set aside for $e$ by C++.

The Larch/C++ model, on the other hand, has a single abstract value inside each location. Even though there is more than one location within $e$, namely $e.sym$ and $e.val$, Larch/C++ does not model the abstract value of $e$ as two locations. Instead, Larch/C++ must model the abstract value of $e$ as a single mathematical entity: a tuple containing the locations for $e.sym$ and $e.val$. The picture for Larch C++ thus looks as follows.

```
!-----------------------------------!
 ! [ * , * ] !
!-------|------------|--------------!
 e | | 
 v v
!-----------------------------------!
 ! ! !
!-----------------------------------!
 e.sym e.val
```

A picture of the Larch/C++ model of $e$.

The above picture shows that $e$ is modeled in Larch/C++ as a location, which contains as its abstract value a tuple (drawn as $[*,*]$, where the two asterisks, *, are the tails of two pointers). This tuple contains two locations, namely the locations for $e.sym$ and $e.val$. However, this tuple of locations is fixed, because $e$ always has the same two fields in C++. So since the abstract
value of e cannot change, it is modeled in Larch/C++ as a constant object. Hence the sort of e is ConstObj[Entry], not Obj[Entry]. The sort Entry is thus a tuple of two locations. See Section 11.10 [Structure and Class Types], page 299 for details on the trait that specifies the sort Entry. The meaning of the sort ConstObj[Entry] is given by the trait ConstObj(Entry) which Larch/C++ implicitly uses when it encounters the declaration of e. See Section 2.8.1.2 [Formal Model of Constant Objects], page 29 for the trait ConstObj.

All of this is rather more complex than one might like, but in actual use it's not so bad. Before getting to the reasons for the complexity, it is important to point out how it can be avoided: one can use a trait that defines the sort Entry (and the trait function contained_objects). When one specifies such a trait, one can specify an abstraction that hides the details of locations, etc. By using a trait with a sort of the same name as the struct, the specifier prevents Larch/C++ from automatically constructing the (complex) model, and instead directly supplies the model. This enables the specifier to use something simple as the abstract value of an instance of a class, such as a tuple of a character string and an integer. (Compare that to a tuple of locations, the first of which contains a pointer to an array containing objects containing characters, and the second of which contains a location containing an integer!) See Chapter 7 [Class Specifications], page 167 for details on how to specify classes and structs with such models. See Section 7.1.1 [A First Class Design (Person)], page 168 for a comparable example, the class Person.

Now for the reasons for the complexity of the automatically-supplied Larch/C++ model of structs. The main reason is that, if one is to model C++ faithfully, then one has to model a struct variable, such as e, as an object. This is because in C++ one can, for example, take the address of e. So without further information from the specifier, Larch/C++ cannot know how e is to be used. Because Larch/C++ cannot make any assumptions about how e is to be used, no simpler model will do.

The second reason for this complexity is that Larch/C++ generally models objects and their values in layers. That is, informally, an object contains a single abstract value, an abstract value may contain objects, and these objects may each contain a single abstract values, etc. The illusion of multiple abstract values may be imposed using a tuple for the abstract value, but mathematically there is only one abstract value.

Returning to more mundane considerations, consider applying state functions to e (see Section 6.2.1 [State Functions], page 103). The sort of e' is Entry, which is a (LSL) tuple of two objects (see Section 11.10 [Structure and Class Types], page 299 for details). The field sym denotes an object of sort Obj[Ptr[Obj[char]]] and the field val denotes an object of sort Obj[int]. That is, the term e'.sym has sort Obj[Ptr[Obj[char]]] and e'.val has sort Obj[int].
### 5.4.5 Union Declarations

The sorts for union types are defined in the same way as for class and struct types. That is, if the user explicitly uses a trait that defines a sort with the same name as the union (and gives a definition of the `contained_objects` trait function that takes that sort), then Larch/C++ does not automatically construct a trait for the union type’s abstract values. However, as with struct and class types, the user may choose to provide such a trait. See Section 11.11 [Union Types], page 302 for details on the automatically-constructed traits for unions.

With the following type definition

```plaintext
union U {int i_var; char *char_p;};
```

consider the following table.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Name</th>
<th>Its Sort (when used as a global variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>U x;</td>
<td>x</td>
<td>Obj[U]</td>
</tr>
</tbody>
</table>

The sort of `x`, when used as a global variable, is `Obj[U]`, which means that `x` is a variable that holds a `U` value. The sort of `x'` is `U`, which is a (LSL) union of two locations that share storage. Assuming the automatically-constructed trait for this type (see Section 11.11 [Union Types], page 302), the tag `i_var` can be used to refer to an object of sort `Obj[int]`, and the tag `char_p` denotes an object of sort `Obj[Ptr[Obj[char]]]`. That is, the term `x'.char_p` has sort `Obj[Ptr[Obj[char]]]` and `x'.i_var` has sort `Obj[int]`.

When a union variable, such as `x` above, is declared Larch/C++ implicitly uses an appropriate instantiation of the trait `MutableObj`. In this example, the included trait is: `MutableObj(U)` (see Section 2.8.1 [Formal Model of Objects], page 24).

See Section 11.11 [Union Types], page 302 for details on how the tags of a union are used to refer to the different parts of the union value in the automatically constructed trait for a union.

### 5.4.6 Function Declarations

The syntax for declaring and defining C++ functions is the same as in C++ [ANSI95] with some additions. See Chapter 6 [Function Specifications], page 82 for the syntax of `fun-spec-body`, and for how to specify (and not just declare or define the interface of) a function. The syntax for a
compound-statement is as in C++ (see Section 17.6 of [Ellis-Stroustrup90][ANSI95]), except that
traits can be passed to higher-order functions (see Section 6.13 [Specifying Higher-Order Functions],
page 157 for the syntax of using-trait-list), and specification-statements are allowed (see Section 6.14
[Behavior Programs], page 164, for these). Since, aside from this addition to the syntax for postfix-expression, the syntax of statements and expressions is identical to that in C++, it is not given in full here.

\[
function-definition ::= fun-interface \[ fun-spec-body \] \[ ctor-initializer \] fun-body \\
| fun-interface \[ fun-spec-body \] function-try-block
\]

fun-interface ::= \[ decl-specifier-seq \] declarator \\
| \[ decl-qualifier-seq \] ctor-declarator \\
| \[ decl-qualifier-seq \] special-function-declarator

decl-qualifier ::= storage-class-specifier | function-specifier | friend | typedef

decl-qualifier-seq ::= decl-qualifier \[ decl-qualifier \] ... 

ctor-initializer ::= : mem-initializer \[ , mem-initializer \] ... 

mem-initializer ::= mem-initializer-id \( \{ \) expression-list \( \) \)

mem-initializer-id ::= complete-class-name | identifier

expression-list ::= expression \[ , expression \] ... 

expression ::= exactly as in C++, but add the following 

postfix-expression ::= postfix-expression \( \{ \) expression-list \( \) \[ using-trait-list \] \( \) 
| simple-type-name \( \{ \) expression-list \( \) \[ using-trait-list \] \( \)

function-try-block ::= try \[ ctor-initializer \] fun-body handler-seq 

handler-seq ::= handler \[ handler \] ...

handler ::= catch \( ) \) exception-declaration \) compound-statement

exception-declaration ::= type-specifier-seq declarator \\
| \[ abstract-declarator \]

compound-statement ::= \{ \) statement-seq \( \}

statement-seq ::= statement \[ statement \] ...

statement ::= exactly as in C++, but add specification-statement

ctor-declarator ::= complete-class-name param-qualifier \\
| complete-template-class-name param-qualifier

complete-template-class-name ::= \[ :: \] \[ nested-name-specifier \] template-class-name

special-function-declarator ::= \[ :: \] nested-name-specifier dtor-name param-qualifier \\
| dtor-name param-qualifier | declarator-id param-qualifier

dtor-name ::= ~ original-class-name | ~ template-class-name

fun-body ::= compound-statement

The form of a fun-interface starting with an optional explicit followed by a complete-class-name or a template-ctor-name is for the constructor of a class (see Section 7.2.1 [Constructors],
A normal fun-interface has the following form, just as in C++ (see Section 8.2.5 of [Stroustrup90]).

\[
[ \text{decl-specifier-seq} ] \text{declarator} ( [ \text{parameter-declaration-clause} ] ) [ \text{cv-qualifier-seq} ]
\]

That is, one specifies the function’s return type (in the \text{decl-specifier-seq} and parts of the \text{declarator}), name (in the \text{declarator}), and formal arguments (in the \text{parameter-declaration-clause}). The use of \text{const} and \text{volatile} in the \text{cv-qualifier-seq} is restricted as in C++ (see Section 8.2.5 of [Stroustrup90]). Finally, one can declare the exceptions that may be raised by the function in the optional \text{exception-decl} (see Section 6.11 [Exceptions], page 148). See Section 5.4 [Declarators], page 65 for details of the syntax.

The syntax for declaring the arguments to a C++ function is also the same as in C++, with a slight addition. See Section 5.4 [Declarators], page 65 for the syntax of \text{declarator}, which is used to declare a function interfaces by using a \text{param-qualifier}. The syntax for a \text{param-qualifier} allows one to specify what traits should also be passed to the function. See Section 6.13 [Specifying Higher-Order Functions], page 157 for how to use this extension.

\[
\text{parameter-declaration-clause} ::= \text{parameter-declaration-list} [ \ldots ]
\]
\[
| \ldots | \text{parameter-declaration-list} , \ldots
\]
\[
\text{parameter-declaration-list} ::= \text{parameter-declaration} , \text{parameter-declaration-list}
\]
\[
\text{parameter-declaration} ::= \text{decl-specifier-seq} \text{declarator} [ \text{parameter-initializer} ]
\]
\[
| \text{abstract-declarator} [ \text{parameter-initializer} ]
\]
\[
\text{parameter-initializer} ::= \text{assignment-expression}
\]
\[
\text{assignment-expression ::= exactly as in C++}
\]

Note that the three dots in ‘\ldots’ are not a meta-notation; they form a single token in the C++ syntax (see Section 8.2.5 in [Ellis-Stroustrup90]). For example, one would write:

\[
\text{int printf(char *fmt, \ldots);}
\]

The sort of a global function name is \text{ConstObj[cpp\_function]}. Consider the following.
### Chapter 5: Declarations

#### 5.4.7 Constant Declarations

The use of `const` in a declaration changes the sort of the object declared. In Larch/C++ the convention is to use the sort generator `ConstObj` instead of `Obj`. However, `structs` and `unions` are more complex, because `const` declarations for them mean that their top-level fields cannot be changed. Also the usual exception has to be made for arrays, because arrays are not themselves objects. For example, consider the following.
### 5.4.8 Summary of Declarations

For C++ type names $T$ and $S$, the above discussion is summarized in the following table.
### Form of Decl. | Sort of x (used as global) | Sort of x^, x' (when x is global)
--- | --- | ---
T x | Obj[T] | T
const T x | ConstObj[T] | T
T & x | Obj[T] | T
const T & x | ConstObj[T] | T
T & const x | Obj[T] | T
T * x | Obj[Ptr[Obj[T]]] | Ptr[Obj[T]]
const T * x | Obj[Ptr[ConstObj[T]]] | Ptr[ConstObj[T]]
T * const x | ConstObj[Ptr[Obj[T]]] | Ptr[Obj[T]]

```cpp
struct STS {
  T t; S s;
};
STS x; ConstObj[STS] STS
const STS x; ConstObj[Const[STS]] Const[STS]
union UST {
  S s; T t;
};
UST x; Obj[UST] UST
const UST x; ConstObj[Const[UST]] Const[UST]
int x(int i); ConstObj[cpp_function] cpp_function
```

Note that the use of a state-function such as ' and ^ (see Section 6.2.1 [State Functions], page 103) peels off the outermost Obj or ConstObj generator from the sort. See Section 6.2.1 [State Functions], page 103 for an explanation of the shorthands x^ and x' for an array name x. See Section 11.7 [Array Types], page 279 for the traits that define array sorts.

When these types are composed, the sort becomes the composite of the sorts given. The only apparent exception is the interaction of `const` and references. When composing arrays with references and pointers, one has to take the C++ precedence rules into account (e.g., [] has higher precedence than *). The following are a few examples.
Chapter 5: Declarations

Declaration Sort of x (used as global)
------------- --------------------------
const int &const x; ConstObj[int]
const int *const x; ConstObj[Ptr[ConstObj[int]]]
const int x[3][4]; Arr[Arr[ConstObj[int]]]
int *x[10]; Arr[Obj[Ptr[Obj[int]]]]
int (*x)[10]; Obj[Ptr[Arr[Obj[int]]]]
int *x[10][4]; Arr[Arr[Obj[Ptr[Obj[int]]]]]
int (*x)[10][4]; Obj[Ptr[Arr[Arr[Obj[int]]]]]
int *const x[10]; Arr[ConstObj[Ptr[Obj[int]]]]
int ***x; Obj[Ptr[Obj[Ptr[Obj[Ptr[Obj[int]]]]]]]

struct Str {
    char *s;
    int len;
};
Str x; ConstObj[Str]

struct IntList {
    int val;
    IntList *next;
};
IntList x; ConstObj[IntList]

const IntList x; ConstObj[Const[IntList]]
const int& x(int a[]); ConstObj[cpp_function]
int (**x)(int i); Obj[Ptr[ConstObj[cpp_function]]]
int (* const x)(int i); ConstObj[Ptr[ConstObj[cpp_function]]]
int (*x[10])(int i); Arr[Obj[Ptr[ConstObj[cpp_function]]]]

5.5 C++ Namespace and Using Declarations

The grammar for the C++ namespace constructs is the same as in the coming C++ standard (see section r.3.3.1.1 of [Stroustrup95] and [ANSI95]). (See Section 5.2.3.2 [Class and Namespace Names], page 61 for the syntax of namespace-name.) The namespace constructs affect the visibility of names as in C++, and must be implemented by the same constructions in the C++ code.

```plaintext
namespace-definition ::= namespace [ identifier ] { [ declaration-seq ] }
    | namespace original-namespace-name { [ declaration-seq ] }
description-seq ::= [ description-seq ] description
namespace-alias-definition ::= namespace identifier = complete-namespace-name ;
```
using-declaration ::= using qualified-id ;
    | using typename [ :: ] nested-name-specifier type-name ;
type-name ::= class-name | non-class-type-name
using-directive ::= using namespace complete-namespace-name ;

See Section 9.1 [Ghost Variables], page 254 for the syntax and semantics of spec-decl.

The following are examples of namespace-defineds.

namespace TerminalUtilities {
    enum WhatToDo { flash, freeze, thaw, home, beep };
    int x_pos = 0;
    int y_pos = 0;
}

//@ uses NoInformation(TerminalControllerClass);
namespace TerminalUtilities { // extension of existing namespace
class TerminalControllerClass {
    public:
        void TermControl(WhatToDo what) throw();
        int num_lines;
        // ... 
    };
typedef TerminalControllerClass TermControl;
namespace AGratuitousNamespace {
    int x;
}

The following are examples of namespace-alias-defineds.

#include "namespace_definition.lh"
namespace Term = TerminalUtilities;
namespace TermAGNamespace = TerminalUtilities::AGratuitousNamespace;
namespace TG = Term::AGratuitousNamespace;

The following are examples of using-declarations.
The following is an example of a using-directive.

The following is an example of a using-directives.

5.6 Linkage Declarations

The syntax of a linkage-declaration is the same as in C++ [ANSI95]; these provide the ability to declare (and specify) the interfaces of functions written in other languages. See Section 5.5 [CPP Namespace and Using Declarations], page 78 for the syntax of declaration-seq.

\[
\text{linkage-declaration ::= } \text{extern string-literal \{ [ declaration-seq ] \} } \\
\text{ | extern string-literal declaration}
\]

For example, the following are examples of linkage-declarations. (See Chapter 6 [Function Specifications], page 82 for explanations of the notation in the specifications.)

extern "C" double ceil(double x);

extern "C" double floor(double x);
//@ behavior {
//@ ensures returns \( x - 1 < \text{result} \) \( \text{result} \leq x \);
//@ }

extern "C" {
// @spec int seed_value;
void srand(int seed);
//@ behavior {
//@ modifies seed_value;
//@ ensures returns \ assigned(seed_value, post) \ seed_value' = seed;
//@ }

int rand();
//@ behavior {
//@ requires assigned(seed_value, pre);
//@ ensures returns \ 0 < result \ result <= INT_MAX;
//@ }
}

5.7 Asm Definitions

The syntax of a \textit{asm-definition} is the same as in C++ [ANSI95]. Such a declaration would presumably only be used to specify implementation design decisions.

\textit{asm-definition} ::= \texttt{asm ( string.literal )} ;
6 Function Specifications

A Larch/C++ function specification has two parts: an interface specification, and a behavioral specification. The interface is specified by a C++ declaration, and the behavioral specification is attached to it, following the keywords behavior or behavior program. Technically, a fun-spec-body may be written in the places noted in the syntax for declaration above. (See Chapter 5 [Declarations], page 54 for the syntax.) Many examples are given in this chapter.

fun-spec-body ::= behavior { [ uses-seq ] [ declaration-seq ] spec-case-seq }

| behavior program compound-statement
uses-seq ::= uses-clause [ uses-clause ] . . .
spec-case-seq ::= spec-case [ also spec-case ] . . .
spec-case ::= [ let-clause ] req-frame-ens [ example-seq ]
            | [ let-clause ] [ requires-clause-seq ] { spec-case-seq } [ ensures-clause-seq ] [ example-seq ]
req-frame-ens ::= [ requires-clause-seq ] ensures-clause-seq
                | [ requires-clause-seq ] frame [ ensures-clause-seq ]
requires-clause-seq ::= requires-clause [ requires-clause ] . . .
requires-clause ::= requires [ redundantly ] pre-cond ;
pre-cond ::= predicate
frame ::= accesses-clause-seq [ modifies-clause-seq ] [ trashes-clause-seq ] [ calls-clause-seq ]
        | modifies-clause-seq [ trashes-clause-seq ] [ calls-clause-seq ]
        | trashes-clause-seq [ calls-clause-seq ]
        | calls-clause-seq
ensures-clause-seq ::= ensures-clause [ ensures-clause ] . . .
ensures-clause ::= ensures [ redundantly ] post-cond ;
                | ensures [ redundantly ] liberally post-cond ;
post-cond ::= predicate

The behavior of a function is specified in the fun-spec-body, which consists of either the keyword behavior and several parts enclosed in a pair of braces ({ and }) or the keywords behavior program and a compound-statement. The first form is more abstract and preferable in situations where higher-order behavior is not required. See Section 6.14 [Behavior Programs], page 164 for a discussion of the second form.

Although Larch/C++ has many parts available to specify the behavior of a C++ function, only a few are necessary in most cases. In this chapter, these simple cases are described first, and more complex variations later. Most of what you need to specify most functions in the behavior form is a single req-frame-ens, which can consist of an optional requires-clause (the precondition), an optional frame (that tells what can be modified, deallocated, or called), and an ensures-clause
(the postcondition). To simplify things even further, we will postpone consideration of redundant clauses and the parts of the frame until later in the chapter.

The foundation of behavioral specification is the use of pre- and postconditions [Hoare69]. The precondition \((\text{pre-cond} \text{ in the grammar})\) is a predicate that follows the Larch/C++ keyword \textit{requires}. It specifies what is required of the client that calls the function. The postcondition \((\text{post-cond} \text{ in the grammar})\) is a predicate that follows the Larch/C++ keyword \textit{ensures}. It specifies what is ensured by the implementation of the function for calls that satisfy the precondition. The predicates are written with the vocabulary of the used trait or traits (see Chapter 9 [Specification Modules], page 254).

An example is the following. (An explanation follows.)

```c++
// @(#)$Id: isqrt.lh,v 1.8 1997/09/16 03:03:30 leavens Exp $
extern int isqrt(int x) throw();
//@ behavior {
//@ requires x >= 0;
//@ ensures (result-1)*(result-1) < x
//@ /
//@ } x < (result+1)*(result+1);
//@ }
```

An informal interpretation of this specification is that it specifies a C++ function named \textbf{isqrt}, which takes one \textbf{int} argument value, and returns an integer approximation to its square root. This function cannot throw any exceptions, because its interface is specified with the C++ syntax \textbf{throw()}. (See Section 6.11 [Exceptions], page 148 for how to specify functions that can throw exceptions.)

In the behavioral part of the specification, the Larch/C++ keyword \textbf{result} stands for the result of a call. The formal argument \textbf{x} stands for the value of the argument. In the above example, the trait used is the built-in trait \textbf{int} (see Section 11.1 [Integer Types], page 269). This trait gives meaning to the trait functions used, including \(\ast\), \(-\), and \(+\). The interface specified is that of a function named \textbf{isqrt}, which takes one \textbf{int} argument, and returns an \textbf{int}, and throws no exceptions. When one calls \textbf{isqrt} with argument \textbf{x}, if \textbf{x} is positive, then the \textbf{result} is some number that satisfies the postcondition. The postcondition has two disjuncts, separated by \(\lor\), which means “or” (see Section 6.1.2 [Logical Connectives], page 87). The first disjunct says that if the result squared is not less than \textbf{x}, then subtracting one from the result gives an approximation that is too small; hence the approximation returned may be larger than the true root, but not too large. Similarly, the second says that if the result squared is not greater than \textbf{x}, then adding one to the result gives an approximation that is too large.
As illustrated by the specification of `isqrt`, one can specify C++ functions that are not easily thought of as mathematical functions of their arguments. That is, a mathematical function, f, would be such that f(28) = f(28), but `isqrt(28)` might return, according to the specification, either 5 or 6, so that two calls `isqrt(28)` might give different results. For example, an implementation of `isqrt` that returned the larger of the two acceptable values every other time it was called would satisfy the above specification. (See Section 6.9.1 [Examples in Function Specifications], page 137 for how to write such examples into the specification using the example-seq.)

Therefore it is better to think of a C++ function not as a mathematical function, but as a relation among its arguments and result. One can think of a relation as a set-valued function, so that such a relation is also a function from a (mathematical) tuple of argument (values) to a set of results. Viewed as a set-valued function, a relation has a domain, which is the set of all argument tuples for which the set of results is non-empty.

The precondition of a function specification (that does not use the keyword `liberally`) describes the argument values that are guaranteed to be related to some result. The postcondition describes the relation itself; that is, it describes the set of results that are related to the given arguments.

Still ignoring the keywords `redundantly` and `liberally`, the declaration-seq, and frame, a C++ function satisfies its specification if and only if for each type-correct function call, if the precondition predicate is satisfied by the arguments, then the function call terminates and the postcondition is satisfied by the result and the arguments. See Section 6.2 [Mutation], page 101 for a more accurate definition that takes side-effects into account, including the declaration-seq and modifies-clause. See Section 6.12 [Liberal Specifications], page 151 for a yet more accurate definition that takes non-termination (specified by the use of `liberally`) into account.

Note that if the precondition is not satisfied by the arguments of a call, nothing is said either about termination or about whether the postcondition is satisfied (if the function terminates). One way to interpret this is that client code should not call a function unless its precondition is satisfied.

The requires-clause-seq is optional; an omitted requires-clause-seq places no requirements on client code. Logically, an omitted requires-clause-seq is equivalent to a requires-clause-seq with one requires-clause of the following form.

requires true;

Either the frame or the ensures-clause-seq may be omitted, but not both. An omitted ensures-clause-seq is equivalent to stating that the function returns normally in a way that satisfies the
frame. Logically, an omitted *ensures-clause-seq* is equivalent to a single *ensures-clause* of the following form.

\[
\text{ensures true;}
\]

An omitted *frame* is equivalent to a *frame* of the following form, which means that all objects may be accessed, no objects can be mutated, no objects may be trashed, and all functions may be called.

\[
\text{accesses everything;}
\text{modifies nothing;}
\text{trashes nothing;}
\text{calls everything;}
\]

(See Section 6.5 [The Accesses Clause], page 133 for details on the *accesses-clause-seq*. See Section 6.2 [Mutation], page 101 for details on the *modifies-clause-seq*. See Section 6.3 [Allocation and Deallocation], page 123 for details on the *trashes-clause-seq*. See Section 6.4 [The Calls Clause], page 132 for details on the *calls-clause-seq*.)

See Section 6.9 [Redundancy in Function Specifications], page 137 for how to write optional examples into a specification, and for how to use the *redundantly* keyword. We call a clause *redundant* if it uses the *redundantly* keyword or if it is an example in an example-seq (see Section 6.9.1 [Examples in Function Specifications], page 137). All non-redundant clauses of a given kind at each level of a *spec-case* must precede all redundant clauses of that kind at that level. For example, one cannot put a redundant *ensures-clause* before a non-redundant one. Multiple non-redundant requires and ensures clauses act as if they were single clauses with their *pre-conds* or *post-conds* conjoined.

See Section 6.10 [Case Analysis], page 143 for the meaning of a specification with multiple *spec-cases*. That section also describes the meaning of a *spec-case* with sub-cases.

See Section 6.12 [Liberal Specifications], page 151 for the meaning of a specification that uses the keyword *liberally*.

The Larch/C++ keyword *result* can only be used in the postcondition. The sort of *result* is the sort associated with the return type specified for the function. In the above specification, *result* is of sort *int*. (See Section 6.1.10 [Larch/C++ Special Primaries], page 99 for more details on the sort of *result*.)
The predicates in the pre- and postconditions principally use the following identifiers and keywords.

- the names of formal arguments,
- \texttt{result} (in the postcondition),
- LSL constants and trait functions from any used traits,
- names of global variables or variables declared in the \texttt{declaration-seq} part of a function body.
- \texttt{this}, \texttt{self}, and the names of appropriately scoped data members can be used in the specification of a member function (see Chapter 7 [Class Specifications], page 167).

There are a few other specialized keywords that can be used in pre- and postconditions. These other keywords are described below, but the ones from the above list will suffice for simple specifications.

Arguments to functions in C++ are passed by value, so the sort of a formal parameter is not the same as the sort for an equivalent global variable declaration. However references have the same sorts. See Section 6.1.8.1 [Sorts for Formal Parameters], page 94 for more details and examples.

The following sections discuss function specifications in more detail.

\section*{6.1 Predicates}

A predicate is a \textit{term} with sort \texttt{Bool}. The syntax of terms is taken from LSL [Guttag-Horning93], with a few Larch/C++ specific additions.

\begin{verbatim}
predicate ::= term
\end{verbatim}

In general, a \textit{term} need not have sort \texttt{Bool}, but if it is used as a predicate, then it must have sort \texttt{Bool}. More detail on the various kinds of \textit{terms} is found below.

\subsection*{6.1.1 If then else}

The lowest precedence operator in a \textit{term} is \texttt{if then else}. Note that for an \texttt{if then else}, there must always be an \texttt{else} part, like the C++ operator “?:”. 
term ::= if term then term else term
     | logical-term

In an `if` `then` `else` term, the first `term` must have sort `Bool`, and the other terms must have the same sort, which is the sort of the whole term. The meaning is just what you would expect. See page 162 of [Guttag-Horning93] for a formal statement.

Note that in LSL, there are no undefined terms; so technically, even a term such as `div(5,0)` would have a value. But with the use of `if` `then` `else`, one can ignore the values of such terms. For example, the following is equivalent to $x' = 3$.

```
if true then x' = 3 else x' = div(5,0)
```

### 6.1.2 Logical Connectives

Slightly higher in precedence operators are the logical connectives.

```
logical-term ::= logical-term logical-opr equality-term
               | equality-term

logical-opr ::= \and | \or | \implies | \slash | \backslash | =>
```

The terms on either side of a `logical-opr` must have sort `Bool`. These also have the usual meaning; that is, `\slash` and `\and` mean “and”, `\or` and `\backslash` mean “or”, and `=>` and `\implies` mean “implies”. See page 161 of [Guttag-Horning93] for a formal statement.

One can also use the C++ syntax `&` and `|` as `lsl-ops` (see Section 6.1.5 [LSL Operator Terms], page 92), as these are defined in a Larch/C++ built-in trait. (See Section 11.4 [bool], page 276 for details on the trait `bool_sugars`.) However, these do not have the same precedence as the `logical-oprs` that are their equivalents `\slash` and `\or`.

### 6.1.3 Equality Terms and Quantifiers

Slightly higher in precedence than the logical connectives are the LSL `eq-oprs`, first-order quantifiers, and the `satisfies` operator. Informal descriptions also fit into the syntax at this level of precedence.
equality-term ::= lsl-op-term [ eq-opr lsl-op-term ]
   | quantifier [ quantifier ] ... ( term )
   | higher-order-comparison
   | informal-desc

eq-opr ::= = | == | \eq | ~= | != | \neq

An equality-term may have any sort, since no eq-opr need be used. See below for more details on the sorts of each form of equality-term.

See Section 6.1.5 [LSL Operator Terms], page 92 for the syntax and meaning of lsl-op-terms. See Section 6.13 [Specifying Higher-Order Functions], page 157 for the meaning of higher-order-comparison. See Section 6.1.4 [Informal Descriptions], page 91 for the syntax and meaning of informal-desc. The other equality-terms are described below.

6.1.3.1 Equality Terms

The eq-ops =, ==, and \eq all mean the same thing. For example, the equality-terms 3 = 3, 3 == 3, and 3 \eq 3 all are true. Similarly, all of ~=, !=, and \neq mean the same thing. For example, 4 ~= 5, 4 != 5, and 4 \neq 5 are all true.

The lsl-op-terms (see Section 6.1.5 [LSL Operator Terms], page 92) on either side of an eq-opr must have the same sort. The sort of a term with an eq-opr in it, such as i = 2, is Bool.

The meaning of = (and its synonyms == and \eq) is standard. The standard meaning is that the two lsl-op-terms must be equal.

The meaning of ~= (and its synonyms !=, and \neq) is the negation of the meaning of =. That is, E1 ~= E2 is true if E1 = E2 are not equal.

6.1.3.2 Quantifiers

The logical quantifiers “for all” and “there exists” are written in Larch/C++ as \A (or \forall) and \E (or \exists). See Section 5.2.3.1 [Simple Type Names], page 60 for the syntax of built-in-type-name. See Section 5.2.3.2 [Class and Namespace Names], page 61 for the syntax of class-name. See Section 8.3 [Instantiation of Templates], page 251 for the syntax of type-id.
quantifier ::= quantifier-sym quantified-list
quantifier-sym ::= \A | \forall | \E | \exists
quantified-list ::= varId : sort-name [ , varId : sort-name ] ... 
varId ::= identifier
sort-name ::= identifier [ sort-instance-actuals ] 
  | class-name | built-in-type-name 
  | typedef-non-class-or-enum-name | typedef-enum-name
sort-instance-actuals ::= [ sort-or-type [ , sort-or-type ] ... ]
sort-or-type ::= identifier [ sort-instance-actuals ] | type-id

The sort of a term with quantifiers is Bool. The term within the parentheses following the quantifiers must also have sort Bool, assuming that the identifiers introduced by the quantifiers have the declared sorts.

An identifier used as a sort-name should be the name of a sort from one of the used traits. Such a name may be parameterized as in LSL, with the sort name’s parameters following it in square brackets, such as String[char]. Note that a class-name is also considered a sort-name, and it may be an instance of a template, such as Set<int>. The grammar does not allow the two forms to be mixed: if Set is known in the specification as a template-class-name, then one must write Set<int>, or ident(Set)[int] (as Set[int] will give a parse error, see Section 4.6 [Identifiers], page 39 for the syntax ident()).

The meaning of a term with quantifiers is the usual one from first-order logic. For example, the following is true.

\A i: int (i < INT_MAX \implies (i < (i + 1)))

The following is also true.

\A i: int (i > INT_MIN \implies (\E j: int (j < i)))

A quantifier introduces a scope (see Section 2.6 [Scope Rules], page 18). For example, in the following, the scope of the i declared in \E i:int extends through (legalIndex(a,i) \&\& (a[i])^ = i) and so the function present looks for an element of a that has a value (in the pre-state) equal to its own index.
However, if it finds such, it then returns the argument \( i \), rather than this index. (See Section 11.8 [Pointer Types], page 287 for the definitions of the trait functions **validUpTo**, which defines \( \text{validUpTo}(a, \text{siz}) \) to mean that the indexes 0 through \( \text{siz} \)-1 (inclusive) are valid indexes into the array into which \( a \) points.)

Often the use of quantifiers can be avoided by writing a trait which has an appropriate trait function. For example, when \( a \) has sort \( \text{Ptr[ConstObj[int]]} \), as in the above specifications, then the following predicate can be written more succinctly as \( \text{validUpTo}(a, \text{siz}) \), because the trait function \( \text{validUpTo} \) is defined in the trait **PointerWithNull**, which Larch/C++ automatically uses when the specification deals with such pointer types (see Section 11.8 [Pointer Types], page 287 for details).

\[ \forall j: \text{int} ((0 \leq j \land j < \text{siz}) \Rightarrow \text{legalIndex}(a, j)) \]

Most other cases where quantifiers are used to iterate over a collection of abstract values can also be made into trait functions; you can write them yourself if they are not standard.
6.1.4 Informal Descriptions

An informal description can be used to escape the rigor of formal specification. In Larch/C++ an informal description can be used anywhere an equality-term can be used, which allows it to be used as a conjunct or disjunct of a predicate, for example. This syntax is also used in other places in Larch/C++ where informal specification is permitted. (See Section 9.2 [The Uses Clause], page 255 for how abstract values can be informally described. See Section 6.9.1 [Examples in Function Specifications], page 137 for how examples can be described informally.)

Informal descriptions have the following syntax.

\[
\text{informal-desc} ::= \text{informally string-literal} \mid \text{informal-comment}
\]

The first form is more wordy, but that may be useful in communicating with users who are not very familiar with Larch/C++. An informal-comment has the syntax \((\% \ldots \%)\) (see Section 4.12 [Informal Comments], page 46 for the detailed syntax), and is more useful when writing multi-line comments than the first form.

For example, the following specification uses an informal precondition, and an informal postcondition.

```c
// @(#)$Id: isqrt-informal.lh,v 1.5 1999/01/11 21:20:14 leavens Exp $
extern int isqrt(int x) throw();
//@ behavior {
//@ requires informally "x is not negative";
//@ ensures (% result is an approximation to
//@ the square root of x %);
//@ }
```

Informal descriptions are considered to have sort `Bool`, and thus should be written as true or false statements. If you find yourself describing some activity, you should rephrase your statement to describe the result of that activity.

The use of informal descriptions allows the level of formality of a Larch/C++ specification to be tuned to some extent. That is, one can write completely informal statements about the behavior of a function, later make some parts of the informal statements formal, and still later refine these into completely formal statements. For example, the following specification has a completely formal precondition, and a partly informal postcondition.
extern int isqrt4(int x) throw();
//@ behavior {
//@ requires x >= 0;
//@ ensures result >= 0
//@ /
//@ (% result is an approximation to
//@ the square root of x %);
//@ }

(See Section 10.2 [Class Refinement], page 260 for how to do this within Larch/C++.)

However, if you need to use informal descriptions to get around some limitation of Larch/C++, we ask that you tell us about the problem instead (send e-mail to lc++@cs.iastate.edu), so that we may consider enhancing the language. If you wish to add informal explanation to a formal Larch/C++ specification, you can also use comments (see Section 4.2 [Comments], page 36). See Section 10.2 [Class Refinement], page 260 for an example using informal descriptions.

Since informal descriptions have no formal semantics, nothing further can be said about them. Hence the rest of this manual largely ignores informal descriptions. (Any formal semantics of Larch/C++ would only apply to specifications that did not use informal descriptions.)

### 6.1.5 LSL Operator Terms

The next higher precedence are LSL operators. See Section 4.10.5 [LSL Operators], page 45 for the syntax of lsl-op. These can be used either as prefix, infix, or postfix operators. The prefix operators are right associative, the postfix and infix operators are left associative.

\[
lsl-op-term ::= \text{lsl-op [ lsl-op ] . . . secondary}
\]

- \text{secondary [ lsl-op secondary ] . . .}
- \text{secondary lsl-op [ lsl-op ] . . .}

The meaning and sort of an lsl-op are determined by the trait in which it is defined.

For example, \text{-i} is an example of the use of a prefix operator (-). An example with two prefix operators is \text{- *p}, note that the space is needed to separate the - from the *, otherwise Larch/C++ interprets this as the use of a single prefix operator \text{-*}. An example with a postfix operator is \text{angle \negated \sine}. An example with an infix operator is \text{s \U s2 \U s3}.
6.1.6 Brackets and Braces

Binding more tightly are the various brackets, and braces: which can be used as infix, prefix, or postfix operators, or as terms (such as tuples and set constructors).

\begin{align*}
  \text{secondary} & ::= \text{primary} \mid [ \text{primary} ] \text{sc-bracketed} \mid \text{primary} \\
  \text{sc-bracketed} & ::= [ [ \text{term-list} ] ] \mid \{ [ \text{term-list} ] \} \\
  & \quad \mid \langle [ \text{term-list} ] \rangle \mid \rangle \text{langle} \mid \langle [ \text{term-list} ] \rangle \rangle \\
  \text{term-list} & ::= \text{term} [ , \text{term} ] 
\end{align*}

The meaning and sort of such a term are determined by the trait in which the overloading of this syntax is defined.

With brackets, one can write a secondary such as \texttt{a[3]}, or \texttt{a2[3,4]}, which may be useful for various kinds of arrays (see Section 11.7 [Array Types], page 279) and pointers (see Section 11.8 [Pointer Types], page 287). One can also use brackets to write a secondary such as \texttt{[2,b]}, which the LSL syntax for tuples.

With braces, one can write a secondary such as \texttt{\{\}}, or \texttt{\{i\}}, for various sets or other containers traits, or something like \texttt{range\{10,11,12\}}. When dealing with overloaded LSL operators, such as set braces, it may sometimes be necessary to give the sort of such a term. For example, if in a certain specification one were dealing with types \texttt{SetOfInt} and \texttt{SetOfChar}, one might want to write \texttt{\{\}::SetOfInt} to be clear about which empty set is meant by \texttt{\{\}}. If the C++ types in question were named \texttt{Set<int>} and \texttt{Set<char>}, one would have to use a \texttt{typedef} to give simple names for the Larch/C++ syntax in this context (see Section 5.2.5 [Typedef Specifiers], page 63).

6.1.7 Primaries

A primary consists of a primitive, possibly followed by zero or more suffixes. The primary-suffixes associate to the left (see Section 6.1.9 [Primary Suffixes], page 98).

\begin{align*}
  \text{primary} & ::= \text{primitive} [ \text{primary-suffix} ] \ldots
\end{align*}
6.1.8 Primitives

A primitive can be a parenthesized term. A primitive can also be a varId, which is an identifier naming a formal parameter, a global variable, or a name introduced by a quantifier. Finally, a primitive can be the application of a trait function to arguments, or an lcpp-primary.

\[
\text{primitive ::= } (\text{term}) ~ | ~ \text{varId} ~ | ~ \text{qualified-id} ~ | ~ \text{fcnId} (\text{term-list}) ~ | ~ \text{lcpp-primary}
\]

\[
\text{fcnId ::= identifier}
\]

The sort of a parenthesized term is the sort of the enclosed term. Its meaning is the meaning of the term.

A fcnId is a trait function identifier; not a C++ function identifier. The sort of a term of the form fcnId (term-list) is given by the used traits as follows. Let the sorts of the terms in term-list be \(S_1, \ldots, S_n\); then there must be a unique trait function named with the identifier of the fcnId, with signature \(S_1, \ldots, S_n \rightarrow T\); and if so the sort of the whole primitive term is \(T\). That is, the sort of the whole primitive term is the return sort of the trait function overloading that matches the sorts of the arguments.

Examples of primitives include \((i + 1)\), \(i\), and \(\text{isEmpty(s1 \cup s2)}\).

6.1.8.1 Sorts for Formal Parameters

The sort of an identifier depends on how it is declared. If it is declared as a global variable, then its sort is given by its declaration (see Section 5.4 [Declarators], page 65). If it is a constant from a trait, then its sort is given in the trait. If an identifier declared in a quantifier, then the declaration explicitly gives its sort. Otherwise, for identifiers declared as formal parameters, the sort is determined from the declared type (see Section 2.7 [Types and Sorts], page 20) as for global variables (see Section 5.4 [Declarators], page 65) with the following important differences. The first difference is that since parameters are passed by value, the leading Obj sort generator (or ConstObj generator, see below) is taken off, unless the type is a reference type. The reason for taking off the leading Obj sort generator is that non-reference parameters are passed by value in C++. (In this we follow LCL, see [Guttag-Horning93].) Second, the use of array types for formal parameters (at the top level) is equivalent to the use of a pointer type. (This follows C++, but differs from LCL
Finally, the sorts of formal struct and union parameters are the corresponding value sort, not the object sort. Examples are given in the following table.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>VarId</th>
<th>Its Sort (when used as a formal parameter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i</td>
<td>i</td>
<td>int</td>
</tr>
<tr>
<td>int &amp; ir</td>
<td>ir</td>
<td>Obj[int]</td>
</tr>
<tr>
<td>int * ip</td>
<td>ip</td>
<td>Ptr[Obj[int]]</td>
</tr>
<tr>
<td>int ai[]</td>
<td>ai</td>
<td>Ptr[Obj[int]]</td>
</tr>
<tr>
<td>struct IPair {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int fst, snd;</td>
<td>sip</td>
<td>Val[IPair]</td>
</tr>
<tr>
<td>}</td>
<td>sip</td>
<td></td>
</tr>
<tr>
<td>union FI {</td>
<td>fi</td>
<td>Val[FI]</td>
</tr>
<tr>
<td>float f;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int i;</td>
<td>fi</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>fi</td>
<td></td>
</tr>
</tbody>
</table>

In the table above, the sorts of ir and i differ, unlike the case for global variables. The sort of *ip is Obj[int], which is also the sort of: ai[0], ip[0], and *ai.

The sort of ai is the same as the sort of ip, because of the C++ equivalence between array parameters and pointer parameters. Another way to think of this is that when an array is passed in C++, the address of the element with index 0 is passed as a pointer value. (Although in C++ there is no type distinction between an array name and a pointer value, so that myArray and &myArray[0] are equivalent in almost all contexts; in the Larch/C++ model these two expressions have different sorts, and thus one should think of the C++ function call f(myArray) as shorthand for f(&myArray[0]). In terms of the trait functions of the trait PrePointer (see Section 11.8 [Pointer Types], page 287), the abstract value passed is address_of(myArray, 0).)

The sort of sip is not IPair, but Val[IPair], because IPair would be a tuple of objects instead of a tuple of values (see Section 11.10 [Structure and Class Types], page 299). The sort Val[IPair] is a tuple with two fields, fst and snd, both of sort int (not Obj[int]), but int values. This models passing structures by value. Hence the sort s of the terms sip.fst and sip snd are both int.

Similarly, the sort of fi is not FI, but Val[FI], since pass by value implies the formal parameter’s model should be a union of values (not objects). The sort Val[FI] is a union with two tags: f and
When defined, the sort of \texttt{fi.f} is \texttt{float}, and the sort of \texttt{fi.i} is \texttt{int}. See Section 11.11 [Union Types], page 302 for details on how the sort \texttt{Val[FI]} is defined.

Except for reference parameters, the use of \texttt{const} does not change the sorts of formal parameters. It also does affect the C++ interface of a function, unless one uses \texttt{const} reference, pointer, or array parameters [ANSI95]. Hence its use is best avoided for formal parameters, except, of course for \texttt{const} reference, pointer, or array parameters. The sort of a formal parameter that uses \texttt{const} is derived from the sort of the corresponding global variable declaration in a way that is analogous to how the sorts of formals without \texttt{const} are derived. That is, the main idea is that the leading \texttt{ConstObj} sort generator is taken off of the sort of the corresponding global variable declaration, but this does not affect reference parameters. For example, consider the following.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>VarId</th>
<th>Its sort (when used as a formal parameter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{const int ci}</td>
<td>ci</td>
<td>int</td>
</tr>
<tr>
<td>\texttt{const int&amp; rc}</td>
<td>rc</td>
<td>\texttt{ConstObj[int]}</td>
</tr>
<tr>
<td>\texttt{int&amp; const cr}</td>
<td>cr</td>
<td>\texttt{ConstObj[int]}</td>
</tr>
<tr>
<td>\texttt{const int * pci}</td>
<td>pci</td>
<td>\texttt{Ptr[ConstObj[int]]}</td>
</tr>
<tr>
<td>\texttt{int * const cip}</td>
<td>cip</td>
<td>\texttt{Ptr[Obj[int]]}</td>
</tr>
<tr>
<td>\texttt{const int cai[]}</td>
<td>cpi</td>
<td>\texttt{Ptr[ConstObj[int]]}</td>
</tr>
<tr>
<td>\texttt{struct IPair { int fst, snd; }}</td>
<td>csip</td>
<td>\texttt{Val[IPair]}</td>
</tr>
</tbody>
</table>

In the above, the sorts of the references are the same as they would be when used as global variables. The sort of \texttt{pci} reflects its being a pointer to constant integer objects. Thus \texttt{pci}, used as a formal parameter, is considered to name a value, but \texttt{*pci} is an lvalue—a constant integer object. The sort \texttt{Val[IPair]} is a tuple of two fields \texttt{fst} and \texttt{snd}, both of sort \texttt{int}. See Section 11.10 [Structure and Class Types], page 299 for details on this sort.

C++ considers the declarations \texttt{int} and \texttt{const int} different for formal parameters, and thus for linking programs with functions so declared. When possible, avoid the use of \texttt{const} for such formals, and only use it for references and pointers (and for pointers, only in the form as shown in the declaration of \texttt{pci}).

For C++ type names \texttt{T} and \texttt{S}, the above discussion is summarized in the following table. See Section 11.10 [Structure and Class Types], page 299 for an explanation of how the sort \texttt{Val[STS]} is defined.
<table>
<thead>
<tr>
<th>Form of Decl.</th>
<th>Sort of x (used as a formal parameter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T x</td>
<td>T</td>
</tr>
<tr>
<td>const T x</td>
<td>T</td>
</tr>
<tr>
<td>T &amp; x</td>
<td>Obj[T]</td>
</tr>
<tr>
<td>const T &amp; x</td>
<td>ConstObj[T]</td>
</tr>
<tr>
<td>T &amp; const x</td>
<td>ConstObj[T]</td>
</tr>
<tr>
<td>T * x</td>
<td>Ptr[Obj[T]]</td>
</tr>
<tr>
<td>const T * x</td>
<td>Ptr[ConstObj[T]]</td>
</tr>
<tr>
<td>T * const x</td>
<td>Ptr[Obj[T]]</td>
</tr>
<tr>
<td>T x[]</td>
<td>Ptr[Obj[T]]</td>
</tr>
<tr>
<td>const T x[]</td>
<td>Ptr[ConstObj[T]]</td>
</tr>
</tbody>
</table>

struct STS {
    T t; S s
};

STS x;       Val[STS]
const STS x;  Val[STS]

When these types are composed, the sort becomes the composite of the sorts given. The only apparent exception is the interaction of `const` and references. The following are a few examples. See Section 11.10 [Structure and Class Types], page 299 discusses how the traits that define `Val[Str]` and `Val[IntList]` would be defined. Note that `Val[IntList]` is a tuple of two fields. The field `val` has sort `int`, and the field `next` has sort `Ptr[Obj[IntList]]`. See Section 5.4.8 [Summary of Declarations], page 76 to compare the following with the sorts of the corresponding global variables.
### 6.1.9 Primary Suffixes

To a *primitive* one can attach one or more of the following suffixes.

\[
\text{primary-suffix ::= selection } | : \text{sort-name} | \text{state-function}
\]

\[
\text{selection ::= . identifier}
\]

One can use `. ` to refer to a field of a tuple value, including the abstract value of a C++ `struct`. The user of Larch/C++ may also define a meaning for selections for other sorts; in that case they are sort checked like trait function applications.

The use of `: sort-name` is to help disambiguate the overloading of trait functions and other terms. It simply says that the preceding *primitive* has the given sort (see Section 2.7 [Types and Sorts], page 20).
Chapter 6: Function Specifications

The use of state-function is to obtain the value of an object in a state (see Section 6.2.1 [State Functions], page 103).

6.1.10 Larch/C++ Special Primaries

The Larch/C++ special primaries are as follows.

\[\text{lcpp-primary ::= literal | this | self | result} \]
\[\quad | \text{pre | post | any | returns} \]
\[\quad | \text{throws (type-id)} \]
\[\quad | \text{thrown (type-id)} \]
\[\quad | \text{sizeof (type-id)} \]
\[\quad | \text{fresh (term-list)} \]
\[\quad | \text{trashed (store-ref-list)} \]
\[\quad | \text{unchanged (store-ref-list)} \]

See Section 4.13 [Literals], page 47 for the syntax and meaning of C++ literals.

An lc++-primary of the form sizeof(tn), where tn is a type-id, denotes an int. Since sizes are C++-implementation dependent, this integer is not uniquely determined by Larch/C++. (See See Section 8.3 [Instantiation of Templates], page 251 for the syntax of type-id.)

See Section 6.2.3 [The Modifies Clause], page 110 for the syntax and meaning of store-ref-list and the forms of lc++-primary beginning with unchanged (see Section 6.2.3.6 [Unchanged], page 121). See Section 6.3.2 [The Trashes Clause], page 126 for the forms of lc++-primary beginning with trashed (see Section 6.3.2.1 [Trashed], page 128). See Section 6.11 [Exceptions], page 148 for the meaning of the forms of lc++-primary beginning with returns, throws, and thrown. The others are explained in this section.

6.1.10.1 This and Self

The Larch/C++ keyword this can only be used in the specification of a member function. It has the same meaning as it does in C++ (see Section r.9.3.1 in [Stroustrup91]). When used in a (non-const) member function of class T, which is associated with sort ValT (often valT is just T, but see Section 2.7 [Types and Sorts], page 20), this has sort ConstObj[Ptr[Obj[ValT]]]. When used in a const member function, the sort is ConstObj[Ptr[ConstObj[ValT]]].
For example, in the specification of a member function of a class `Person` (see Section 7.1.1 [A First Class Design (Person)], page 168 for more details on this example), the sort of `this` is `ConstObj[Ptr[Obj[Person]]]`. However, in the specification of a `const` member function of class `Person`, such as `years_old`, the sort of `this` is `ConstObj[Ptr[ConstObj[Person]]]`.

The Larch/C++ keyword `self` is a shorthand for `*(this\any)`, which is dereferencing the pointer value found in `this` in some visible state. See Section 6.2.1 [State Functions], page 103 for details about the `state-function \any` (and see Section 6.2 [Mutation], page 101 for the notion of state.) (Any visible state is acceptable, because the `this` pointer itself cannot be assigned to in a C++ program.) Thus a form such as `self\any` is shorthand for `*(this\any)`\any. The keyword `self` can only be used in the specification of a member function. When used in a member function of class `T`, which is associated with sort `ValT`, `self` has sort `Obj[ValT]`. Most often `self` is much more convenient for specification than `this`.

For example, in the specification of a member function of a class `Person` (see Section 7.1.1 [A First Class Design (Person)], page 168 for more details on this example), the sort of `self` is `Obj[Person]`. However, in the specification of a `const` member function of class `Person`, such as `years_old`, the sort of `self` is `ConstObj[Person]`.

See Chapter 7 [Class Specifications], page 167 for more examples.

### 6.1.10.2 Result

The Larch/C++ keyword `result` stands for the result of a function (when it does not throw an exception). Its sort is determined as if `result` were a formal parameter declared like the result type of the function. For example, consider the following.

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Sort of result (used in body of foo)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int foo(float x)</td>
<td>int</td>
</tr>
<tr>
<td>int &amp; foo(float x)</td>
<td>Obj[int]</td>
</tr>
<tr>
<td>int * foo(float x)</td>
<td>Ptr[Obj[int]]</td>
</tr>
<tr>
<td>void foo(int &amp; i)</td>
<td>void</td>
</tr>
<tr>
<td>void * foo(int *ip)</td>
<td>Ptr[Obj[void]]</td>
</tr>
</tbody>
</table>

See Section 6.1.8.1 [Sorts for Formal Parameters], page 94 for details on determining the sorts of formal parameters; these details are the same for determining the sort of `result`. See Section 11.5 [void], page 278 for the trait that defines the abstract values of the C++ type `void`. 
See Chapter 6 [Function Specifications], page 82 for a simple example. See Section 6.1.3.2 [Quantifiers], page 88 for a more interesting example, where result is described more indirectly.

Although \texttt{result} can be used in functions whose return type is \texttt{void}, doing so is confusing and is best avoided. The Larch/C++ model for the type \texttt{void} is a set whose only element is written \texttt{theVoid}. Because of there is only one element in this sort the term \texttt{result = theVoid} is always \texttt{true} [Jones95e]. If you want to assert that a function that returns \texttt{void} returns instead of signalling an exception, the correct term to use is \texttt{returns} (see Section 6.11 [Exceptions], page 148 for more details and examples).

6.10.3 Names of States (pre, post, and any)

The \texttt{lcpp-primary pre} stands for the state just before the invocation of the function being specified. The \texttt{lcpp-primary post} stands for the state just after that function’s return. The \texttt{lcpp-primary any} can be used when either of these will do, and in invariants. Each of these has the sort \texttt{State}, which is the sort of the formal model of states in Larch/C++. See Section 2.8.2 [Formal Model of States], page 30 for details on the trait \texttt{State} that defines the sort \texttt{State}.

This feature of Larch/C++ was used in the specification language LM3 (Chapter 6 of [Guttag-Horning93], see Section 2.3.2 of [Jones91]). In LM3 [Jones91], Jones emphasizes the use of explicit states for specifying higher-order procedures.

6.2 Mutation

Mutation of an object means changing its abstract value. This can happen in one of two ways:

- the object had a well-defined abstract value that is changed, or
- the object had no well-defined abstract value, but it became well-defined.

The word “modifies” is sometimes used as a synonym for “mutates”.

To specify a function that mutates an object, one usually has to refer to the object’s abstract value in two states: just before the function body is run (but after parameter passing) and just as it is about to return. Informally one refers to the value of an object at a given time as its state. More formally, a state is a (mathematical) function from objects to their values. An informal phrase such as “the object \texttt{x} has value 2 before the call” means that the state before the call (after parameter
passing) maps the object \( x \) to 2. See Section 2.8 [Objects and Values], page 21 for an introductory discussion of objects and values.

The states of interest in a function specification are as follows:

- the pre-state, which maps objects to their values just before the function body is run, but after parameter passing, and
- the post-state, which maps objects to their values at the point of returning from the call (or throwing an exception), but before the function parameters go out of scope.

Thus, to allow for mutation, the formal model of a \( \text{C++} \) function (that cannot throw exceptions) is as a relation between:

- the formal arguments and the pre-state, and
- the post-state and the result.

Note that the in Larch/\( \text{C++} \) even a \texttt{void} function has a result, which is \texttt{theVoid}. See Section 6.1.10.2 [Result], page 100 for more discussion on this point.

Therefore a function specification’s precondition describes the arguments and pre-states over which the function is defined (the domain of the relation). The postcondition describes the relation itself; that is, it describes the set of post-state and result pairs that are related to a given tuple of actual arguments and a pre-state.

The \textit{modifies-clause} in a specification can be used to state what objects a function is allowed to mutate, and what objects it is \textit{not} allowed to mutate. (It is part of a “frame axiom” for a specification [Borgida-etal95]. The other part of the frame axiom is the \textit{trashes-clause}, which, if omitted, says that no object can become unassigned or deallocated (see Section 6.3.2 [The Trashes Clause], page 126).

The subsections below describe: state functions, which allow one to specify the value of an object in the pre- or post-state, the trait functions \texttt{allocated} and \texttt{assigned}, which allow one to specify whether objects are allocated and well-defined, the \textit{modifies-clause}, and formal details of the \textit{modifies-clause}. 

6.2.1 State Functions

If an object is assigned in a state (see Section 6.2.2 [Allocated and Assigned], page 107), then its abstract value can be obtained by using a state-function.

\[\text{state-function} ::= \neg | \pre | \post | \any | \obj\]

The value in the pre-state will be called the pre-value and the value in the post-state will be called the post-value.

For example, consider the following.

```c
// @(#)$Id: add_one.lh,v 1.6 1997/06/03 20:29:55 leavens Exp $ external void add_one(int& x) throw();
//@ behavior {
//@ requires assigned(x, pre) // x is allocated and has a value
//@ \pre \ x^ < INT_MAX; // \x^ \ : pre-value of x
//@ modifies x; // x : an object
//@ ensures x' = x^ + 1; // x' \ : post-value of x
//@ }
```

Informally, \texttt{add\_one} takes an integer object passed by reference, and adds 1 to it. More formally, assuming that \texttt{x} has a well-defined value, and that the value of \texttt{x} in the pre-state is not too large to be incremented, the post-value of \texttt{x} is one greater than the original value of \texttt{x}.

The following table summarizes the sorts of terms using the different state functions.

<table>
<thead>
<tr>
<th>term</th>
<th>sort (assuming x declared as: int &amp; x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Obj[int]</td>
</tr>
<tr>
<td>x^</td>
<td>int</td>
</tr>
<tr>
<td>x\pre</td>
<td>int</td>
</tr>
<tr>
<td>x'</td>
<td>int</td>
</tr>
<tr>
<td>x\post</td>
<td>int</td>
</tr>
<tr>
<td>x\any</td>
<td>int</td>
</tr>
<tr>
<td>x\obj</td>
<td>Obj[int]</td>
</tr>
</tbody>
</table>

The \texttt{state-functions} \neg and \pre are synonymous, and both are used to extract the pre-value of an object. The \texttt{state-functions} \pre and \post are synonymous, and both are used to extract the...
post-value of an object. For invariants (see Section 7.3 [Class Invariants], page 196) and other situations where one wishes to extract the value of an object, but for which no particular visible state is appropriate, the \textit{state-function} \texttt{\any} can be used. By a \textit{visible state} we mean a state that a client can observe. For a class with no public data members (see Section 2.3 [Accessibility of Class Members in Specifications], page 16), the visible states are the state just after an instance is created and initialized by a creator, and the states just before and just after the call to any member function (or friend function), in which the instance is passed. For a class with public data members, every state is a visible state, which is a good reason not to have public data members.

The \textit{state-function} \texttt{\obj} can be used to explicitly refer to an object itself, instead of its value. It is essentially a no-op, and thus need only be used for emphasis.

Formally, the value of applying a \textit{state-function} to a value (which will usually be an object), is given by passing the value and the appropriate state to the trait function \texttt{eval} in the trait \texttt{TypedObj} (see Section 2.8.1 [Formal Model of Objects], page 24 for the trait \texttt{TypedObj}). For example, \texttt{i^} means \texttt{eval(i,pre)}. (See below for syntactic sugars that allow a \textit{state-function} to be applied to a value that is not an object.)

(One can also write \texttt{eval(i,pre)} in specifications. However, \texttt{i^} is shorter, and hence is the preferred form in Larch/C++)

Except for syntactic sugars discussed below, the \textit{state-functions} can only be applied to terms that denote objects; that is to terms whose sort has the form \texttt{Obj[T]} or \texttt{ConstObj[T]} for some \texttt{T}. The sort of a term with \textit{state-function} \texttt{\obj} is unchanged, but sorts \texttt{\^, \pre, \', \post, and \any} take off the leading \texttt{Obj} or \texttt{ConstObj} sort generator.

You can only use a \textit{state-function} on a formal parameter name if that name parameter is passed by reference. (More precisely, you can only use a \textit{state-function} on values for which the trait function \texttt{eval} is defined, this is usually only object sorts.) Value parameters are not considered objects in Larch/C++ (see Section 6.1.8.1 [Sorts for Formal Parameters], page 94) and so the following is an error. (See Section 11.1 [Integer Types], page 269 for the trait function \texttt{to_int}.)

```c++
bool equal(int x, int y);
//@ behavior {
//@ ensures result = (x^ = y^); // error
//@ }
```

In the above example, both \texttt{x} and \texttt{y} denote values not objects; thus \texttt{^} cannot be used. A legal \texttt{ensures} clause would be \texttt{ensures result = (x = y);}.
For C++ arrays, one can apply a state-function to the elements of the array. For example, the following specifies a function that adds one to each element of an array. Recall the sort of the formal parameter declared int ai[] is $\text{Ptr}[\text{Obj}[\text{int}]]$, because such C++ syntax means that a pointer (into the array) is passed. The term $\text{assignedUpTo}(ai,siz)$ (see Section 11.8 [Pointer Types], page 287) means that the pointer is not null, that the integers 0 to $siz-1$ (inclusive) are legal indexes for this pointer, and that each element in that range is an allocated object with a well-defined value. Thus the pointer can be treated as an initialized array of size $siz$. To avoid a syntax error, a parenthesis is needed around $ai[i]$ in terms like $(ai[i])^\wedge$. This is because $ai[i]$ is a secondary, and a state-function can only be applied to a primary. See Section 6.1.7 [Primaries], page 93 for the syntax of primary. See Section 6.1.6 [Brackets and Braces], page 93 for the syntax of secondary.

```c
// @(#)$$Id: array_add_one.lh,v 1.12 1997/06/03 20:29:57 leavens Exp $
extern void array_add_one(int ai[], int siz) throw();
//@ behavior {
//@ requires 0 <= siz \ assignedUpTo(ai, siz)
//@ /
//@   \A i:int ((0 <= i \&\& i <= (siz-1))
//@     => (ai[i])^ + 1 <= INT_MAX);
//@ modifies ai;
//@ ensures \A i:int ((0 <= i \&\& i <= (siz-1))
//@     => (ai[i])' = (ai[i])^ + 1);
//@ }
```

An array name (for global array variables), or a pointer formal, such as $ai$ in the above example, is not an object. However, as in LCL we extend the meaning of the state-functions to arrays (and to pointers into arrays) element-wise. In the context of $\text{array_add_one}$, this extension makes $ai'$ stand for the abstract value of sort $\text{Arr}[\text{int}]$ that maps each legal index $i$ of $ai$ (if any) to $(ai[i])'$. Note that the sort of $ai'$ is not a pointer sort, but an array sort. In the above example, both $(*ai)'$ and $ai'[0]$ would mean the same thing, and the syntax $(ai[i])^\wedge$ could be equivalently written as $ai^-[i]$ (or even $(*(ai + i))^\wedge$).

These syntactic sugars are defined because the trait function $\text{eval}$ is defined for pointers and arrays in the traits $\text{Pointer}$ (see Section 11.8 [Pointer Types], page 287) and $\text{Array}$ (see Section 11.7 [Array Types], page 279). That is, $ai'[i]$ is defined when the term $(\text{eval}(ai,post))[i]$ is defined, because the latter is the desugared version of the former.

The sort $\text{Arr}[\text{int}]$ is defined by the trait $\text{Array}$ (see Section 11.7 [Array Types], page 279), which is included in the trait $\text{Pointer}$ (see Section 11.8 [Pointer Types], page 287).
The same sugar applies to multi-dimensional arrays. For example, if \texttt{aai} is a two-dimensional array of\texttt{ints} (declared as a global variable), then \texttt{aai'} means \texttt{eval(aai,post)}. As defined in the trait \texttt{MultiDimensionalArray} (see Section 11.7 \[Array Types\], page 279), this is the abstract value of sort \texttt{Arr[Arr[int]]} that assigns to each pair of legal indexs, \texttt{\textit{i}} and \texttt{\textit{j}}, the abstract value \texttt{((aai[i])[j])'}.

The same kind of sugar is defined for pointers to arrays. See Section 11.8 \[Pointer Types\], page 287 for details in the trait \texttt{PointerToArray}.

For C\texttt{++} structures and classes that use the default (automatically-constructed) trait, one can apply a \textit{state-function} to the fields of its abstract value. For example, consider the following global structure variable definition.

\begin{verbatim}
struct Ratl { int num, denom; }
Ratl my_ratl;
\end{verbatim}

One can write \texttt{my_ratl^.denom^}, which has sort \texttt{int}, because \texttt{my_ratl^.denom} is an object of sort \texttt{Obj[int]}. See Section 11.10 \[Structure and Class Types\], page 299 for details on the automatically constructed traits for structures and classes.

In the default traits for structures and classes, Larch/C\texttt{++} extends the meaning of the \textit{state-functions} to the tuples that are their abstract values element-wise. Thus one can write \texttt{my_ratl^^}, which means \texttt{eval(eval(my_ratl,pre),pre)}: because the trait function \texttt{eval} is defined on the sort \texttt{Ratl} to return a value of sort \texttt{Val[Ratl]}. That is, \texttt{my_ratl^^}, applies the \textit{state-function} \texttt{\textasciitilde} to the tuple that is the result of \texttt{my_ratl^}, and is thus equal to the following.

\begin{verbatim}
[my_ratl^.num^, my_ratl^.denom^]
\end{verbatim}

This last term denotes an abstract value of sort \texttt{Val_Ratl}, containing the numerator and denominator values in the pre-state. Hence \texttt{my_ratl^.num} has sort \texttt{int}. It follows that \texttt{my_ratl^.num} and \texttt{my_ratl^.num^} mean the same thing. One way to see this is that \texttt{my_ratl^.num} has sort \texttt{Obj[int]}, because \texttt{my_ratl^} is a tuple of objects. See Section 11.10 \[Structure and Class Types\], page 299 for more examples, and for details of how the trait functions \texttt{eval} and the sort \texttt{Val[Ratl]} are defined).

You may define a similar shorthand for the abstract values of a type you specify the abstract values of explicitly (see Chapter 7 \[Class Specifications\], page 167), by specifying in LSL what the trait function \texttt{eval} means for the abstract values. Once this is done, the meaning of a state function gives the appropriate shorthand.
6.2.2 Allocated and Assigned

The domain of a state is the set of objects that are allocated in that state. Informally, being allocated means, as you would expect, that the object is in the domain of that state. The idea that an object loc is allocated in a state st can be written in a specification as the term \texttt{allocated}(obj, st). (Chalin calls such objects “active” [Chalin95].)

An object can exist without being allocated. One example of such an object would be an object on the run-time stack of C++ that was allocated by a procedure that has returned. Another example is an object that was allocated with operator \texttt{new}, but which has since been deleted. Formally, such objects are not in the domain of the state modeled by the trait \texttt{State} (see Section 2.8.2 [Formal Model of States], page 30).

In Larch/C++, the formal model of objects distinguishes between typed and untyped objects. (See Section 2.8.1 [Formal Model of Objects], page 24.) Thus by widening an untyped object, \texttt{obj}, to some random type \texttt{Loc}[T], one obtains a typed object, \texttt{widen}(obj). But an untyped object, even one that is allocated in a state, only underlies a finite number of typed objects, and so not all such widenings should be considered to be allocated. Hence for a typed object, the trait function \texttt{allocated} takes the type of the object and the types recorded in the state into account. See Section 6.2.3 [The Modifies Clause], page 110 for one way that this distinction helps in the semantics.

Once allocated, an object may be assigned a value; objects that have no well-defined value (Chalin’s term again), are unassigned. The standard example is an uninitialized variable. The term \texttt{assigned}(obj, st) is true if the object \texttt{obj} is allocated in \texttt{st} and has a well-defined value in \texttt{st}.

The details are given in the following trait. See Section 2.8.1 [Formal Model of Objects], page 24 for the traits assumed by this trait, and for more details on the general model of objects and values.
asserts
\forall loc: \text{Loc}[T], \text{st}: \text{State}
\begin{align*}
\text{allocated}(loc, \text{st}) \\
= \text{allocated}(\text{widen}(loc), \text{st}) \\
\land \ \text{type_of}(loc) \in \text{types_of}(\text{widen}(loc), \text{st}); \\
\text{assigned}(loc, \text{st}) \\
= \text{allocated}(loc, \text{st}) \\
\land \ \text{narrow}(\text{eval}(\text{widen}(loc), \text{st})) \neq \text{unassigned};
\end{align*}

implies
\forall loc: \text{Loc}[T], \text{st}: \text{State}
\begin{align*}
\text{allocated}(loc, \text{st}) \implies \text{widen}(loc) \in \text{domain}(\text{st}); \\
\text{assigned}(loc, \text{st}) \implies \text{allocated}(loc, \text{st});
\end{align*}

converts
\begin{align*}
\text{assigned}, \text{allocated}: \text{Loc}[T], \text{State} \to \text{Bool}
\end{align*}

Because not all objects are allocated in a state or assigned a value (see Section 6.2.2 [Allocated and Assigned], page 107), and because the logic of LSL gives an arbitrary value to an expression such as \text{eval}(i,\text{pre}) when \text{i} is not assigned, one should generally avoid using a state function on an unassigned object. This can be done by requiring objects to be assigned in the precondition.

In general, you only need to specify that an object is allocated when using pointer variables. (See Section 11.8 [Pointer Types], page 287 for the trait functions used with pointers.) Larch/C++ has constraints on variables and reference parameters used in specifications that eliminate the need to state explicitly whether variables are allocated, except for pointers.

Larch/C++ does provide a syntactic sugar that implicitly requires reference parameters and global variables used in functions to be allocated. Implicitly, the term \text{allocated}(x,\text{pre}) is conjoined to the precondition of each function specification, for each such reference parameter and global variable, \text{x}. Note that such objects are not required to be assigned, only allocated.

For example, the following is the desugared form of the specification of \text{add_one} given above (see Section 6.2.1 [State Functions], page 103). The desugaring actually produces a redundant requirement in this example (because \text{assigned}(x,\text{pre}) implies \text{allocated}(x,\text{pre})), but this illustrates the sugaring process.
extern void add_one(int& x) throw();
//@ behavior {
//@ requires allocated(x, pre) // added by the syntactic sugar
//@ /
//@ ensures x' = x^ + 1; // x' : post-value of x
//@ }

Similarly, the implicit parameter (this) in a member function specification is required to be a well-defined, non-null pointer variable that points to an assigned object (self). (Recall that the pre-state of a constructor is not considered a visible state.) See Chapter 7 [Class Specifications], page 167 for more about the this pointer and self.) Furthermore, if there are class members visible, then because of the last conjunct for the implicit argument, these class members are also required to be assigned.

These implicit preconditions are summarized in the following table.

<table>
<thead>
<tr>
<th>what</th>
<th>declaration</th>
<th>implicit precondition conjuncts</th>
</tr>
</thead>
<tbody>
<tr>
<td>formal parameter</td>
<td>T &amp; x</td>
<td>allocated(x, pre)</td>
</tr>
<tr>
<td>formal parameter</td>
<td>T * x</td>
<td>allocated(x, pre) // none!</td>
</tr>
<tr>
<td>global variable</td>
<td>extern T x</td>
<td>allocated(x, pre)</td>
</tr>
<tr>
<td>implicit argument</td>
<td>extern T x</td>
<td>assigned(this, pre) \ assigned(self, pre) \ assigned(self^, pre)</td>
</tr>
<tr>
<td>class member</td>
<td>T dm</td>
<td>assigned(dm, pre) // follows from above</td>
</tr>
</tbody>
</table>

Recall that objects that are allocated or assigned stay allocated or assigned unless they are mentioned in a trashes-clause. Hence it is not usually necessary to state that parameters or implicit arguments remain allocated or assigned in a postcondition. See Section 6.3.2 [The Trashes Clause], page 126 for more details on this point.

When a new object is created by a function, there is no implicit conjunct in a post-condition that says that it must be allocated or assigned. Such a conjunct must be written explicitly if desired. Often, however, you will write such an assertion using fresh, to assert both that the storage returned is allocated, and that it was not allocated in the pre-state (see Section 6.3.1 [Fresh], page 123). The following is an example.
6.2.3 The Modifies Clause

To specify a function that mutates objects, the specification must include a modifies-clause. The absence of a modifies-clause means that no objects can have their abstract values mutated by an execution of the function (that satisfies the pre-condition). The presence of a modifies-clause asserts that only the set of objects described, may have their abstract values newly-defined or changed by the function. We say that an object is described by a modifies-clause if it is either explicitly specified by it or if some described object depends on it (see Chapter 10 of [Leino95]). (See Section 7.6 [The Depends Clause], page 213 for a description of dependencies.) This is a strong indirect assertion that no other allocated objects, other than those described, are allowed to change their abstract values. An object that is included in the set described by the modifies-clause does not have to be changed by the function; the point is that it is allowed to be changed.

\[
\text{modifies-clause-seq ::= modifies-clause \ [ \text{modifies-clause} \ ] \ldots}
\]

\[
\text{modifies-clause ::= modifies \ [ \text{redundantly} \ ] \ text-ref-list ;}
\]

\[
\text{\quad \mid \ constructs \ [ \text{redundantly} \ ] \ text-ref-list ;}
\]

\[
\text{\quad \text{store-ref-list ::= text-ref \ [ , text-ref \ ] \ldots \ | \text{nothing} \ | \text{everything}}}
\]

\[
\text{\quad \text{store-ref ::= text} \ | \text{reach ( text)}}
\]

See Section 6.9.4 [Redundancy in Frames], page 143 for the meaning of redundantly used in a modifies-clause. When several non-redundant modifies-clauses are listed in a modifies-clause-seq, this is the same as listing each of their store-ref-lists in a single modifies-clause. If more than one non-redundant modifies-clause is given, then none of the store-ref-lists may be of the form nothing or everything. Because it is possible to translate multiple non-redundant modifies-clauses into a single modifies-clause, we will assume from now on that there is only one non-redundant modifies-clause.

As an example, (assuming that there are no dependencies visible) the function swap specified below may change the abstract values of both \(x\) and \(y\), but nothing else; i.e., it cannot change the abstract values of any other global variables.
Unlike earlier versions of Larch/C++ (and LCL), mutation of an object does not include deallocation or "trashing" it. (The current semantics is based on the work of Chalin [Chalin95].) The modifies-clause only concerns objects that:

- are allocated in both the pre- and post-states, and
- are assigned in the post-state.

It says that out of all such objects, only the objects described by the modifies-clause may have their abstract values changed.

The modifies clause thus does not concern objects that are: freshly allocated by the function (i.e., that are not allocated in the pre-state and allocated in the post-state), deallocated by the function (allocated in the pre-state and not allocated in the post-state), or not assigned in the post-state. The reason the modifies-clause does not concern objects $x$ that are not assigned in the post-state (i.e., such that $\neg \text{assigned}(x, \text{post})$) is because the abstract values of such objects are not well-defined. For such an unassigned object, there is no good way to define the concept of mutation. This is also the reason that the modifies clause is not concerned with objects that were not allocated in the pre-state. (Technically, this is because state functions will map such objects to arbitrary values, and there is no guarantee that the pre-state did not map such an object to the same arbitrary value as the post-state.\footnote{Thanks to Chalin for several discussions about this point.})

Allocation of objects can be specified by using the keyword fresh (see Section 6.3.1 [Fresh], page 123). Trashing objects (either by making them unassigned or by deallocating them) is only permitted if a function is specified using a trashes-clause (see Section 6.3.2 [The Trashes Clause], page 126). This means that all objects not described by the trashes-clause-seq (and for an omitted trashes-clause-seq this is literally all objects) that are allocated in the pre-state remain allocated in the post-state. Furthermore, all objects not described by the trashes-clause-seq that were assigned in the pre-state remain assigned in the post-state.

\footnote{Thanks to Chalin for several discussions about this point.}
Chapter 6: Function Specifications

An omitted modifies-clause-seq is equivalent to a modifies-clause-seq of the form modifies nothing; meaning no (already assigned) object can be mutated by the specified function. The other extreme is modifies everything; meaning the function can change all objects to which it has access in the pre-state. Thus, modifies everything means that the frame axiom is trivially satisfied.

A modifies clause may be written informally. See Section 6.1.4 [Informal Descriptions], page 91 for the syntax; in this case the clause should informally describe a set of (typed) objects. However, you should avoid using informality in the modifies-clause if possible, as it is easy, by naming formal parameters, or by using everything or reach, to specify something formal.

The clause modifies reach(x); means that the function may change the abstract values of the set of objects that are reachable from x, including x itself (see Section 6.2.3.5 [Reach], page 120).

A simple example of using a term in a modifies-clause is given by the following. In this example, the store-ref-list \*p could also be written as p[p.idx]. See Section 11.8 [Pointer Types], page 287 for details on these alternatives to writing \*p and for the trait function allocated used in the precondition.

extern void set_to_one(int *p) throw();
//@ behavior {
//@ requires allocated(p, pre);
//@ modifies *p;
//@ ensures assigned(p, post) \&\& (*p)’ = 1;
//@ ensures redundantly assigned(*p, post);
//@ }

The reason the post-condition asserts that assigned(p, post) is that p is not required to point at an assigned object in the pre-state. As stated in the redundant ensures-clause (see Section 6.9.3 [Redundant Ensures Clauses or Claims], page 141) it follows that assigned(*p, post) holds (See Section 11.8 [Pointer Types], page 287 for details on the meaning of assigned). This makes it clear when *p is assigned. (Technically, the conjunct assigned(p, post) or assigned(*p, post) is needed in the post-condition because otherwise *p will not have a well-defined value in the post-state.)

One should be careful, however, with pointers and arrays, to ensure that one gives the correct set of objects. The trait function contained_objects, is useful in this respect. For pointers into arrays (such as array formal parameters) and for arrays, one can use trait functions such as
objectsInRange found in the traits for pointers (see Section 11.8 [Pointer Types], page 287 for details in the trait Pointer) and arrays (see Section 11.7 [Array Types], page 279 for details in the trait Array). For string manipulations, one can use a more specialized trait function, such as objectsToNull (see Section 11.9 [Character Strings], page 296 for details of the trait cpp_char_string). An example of the use of objectsToNull is given below. (See Section 6.9.3 [Redundant Ensures Clauses or Claims], page 141 for the meaning of the redundant ensures-clause.)

```c
extern void poorly_encrypt(unsigned char *s) throw();
//@ behavior {
//@ uses cpp_unsignedChar_string;
//@ requires nullTerminated(s, pre);
//@ modifies objectsToNull(s, pre);
//@ ensures \A i: int (legalStringIndex(s,pre,i) 
//@ => (*(s + i))' = (*(s + i))^ + 1);
//@ ensures redundantly (legalStringIndex(s,pre,0) /
//@ (*s)' = UCHAR_MAX)
//@ => ((*s)' = 0);
//@ }
```

Further discussion of topics related to modifies-clause is found below.

### 6.2.3.1 Constructs

In C++, a constructor's job is actually to initialize an object; C++ itself allocates the storage for the object (see Section 7.2.1 [Constructors], page 188). However, the reader of the specification of a constructor should not be forced to think about how the user's code interacts with C++ to allocate and initialize an object. So Larch/C++ provides the keyword constructs as a synonym for modifies; it is used to convey to the reader that a set of objects (usually self) is not only modified, but, by the graces of C++, allocated and initialized.

### 6.2.3.2 Syntactic Sugars in the Modifies Clause

The contained_objects trait function is generally useful in a store-ref-list for extracting objects from abstract values (as opposed to objects). It is also the basis for a syntactic sugar in Larch/C++.

If the sort of a term listed in the store-ref-list is not Set[TypeTaggedObject] or of the form Obj[T], then the trait function contained_objects is applied, to the value of the term and the
pre-state to obtain a set of type-tagged objects. The set so obtained must be non-empty. This is particularly helpful as a syntactic sugar for C++ arrays and pointers, and for structure and class types for which the default trait is used.

For global array variables (or fields of structures, etc. that are arrays), the `contained_objects` trait function returns a set of all the type-tagged objects that are the elements of the array. (See Section 11.7 [Array Types], page 279 for details.) The same functionality is provided by the trait function `contained_objects` for pointers (and for arrays passed as parameters, see Section 11.8 [Pointer Types], page 287). Thus, in a store-ref-list Larch/C++ allows an array name or pointer value as a store-ref. When an array name or pointer value is mentioned in the modifies clause, it is considered shorthand for all the elements of the array (pointed to). For example, in the following, the modifies-clause is shorthand for modifies `contained_objects(ai, pre)`.

```c
//@ behavior {
//@ requires 0 <= siz /
//@ \A i:int ((0 <= i /
//@ i <= (siz-1))
//@ => (ai[i])^ + 1 <= INT_MAX);
//@ modifies ai;
//@ ensures \A i:int ((0 <= i /
//@ i <= (siz-1))
//@ => (ai[i])' = (ai[i])^ + 1);
//@ }
```

Note that this shorthand names a wider range of objects than strictly necessary; it would be more accurate to write the following modifies-clause.

```c
modifies objectsInRange(ai, 0, siz-1);
```

A similar shorthand applies to struct and class names for which the default trait (see Section 11.10 [Structure and Class Types], page 299) is used. For example, by default a struct global variable of type T is modeled in Larch/C++ as an object of sort `ConstObj[T]`, and since the trait `ConstObj` (see Section 2.8.1 [Formal Model of Objects], page 24) defines the `contained_objects` trait function. Thus if `s` is the name of a global variable that is a struct of type T, then modifies `s`; is shorthand for modifies `contained_objects(s, pre)`; which says that all the fields of s can be modified, as would be desired. However, for structs within structs, one would have to use `reach`, or some other term to state that the subfields could also be modified.

This syntactic sugar applies to each sort with `contained_objects` defined, which should be most sorts of values in Larch/C++. An advantage of this approach is that it extends to user-defined types.
(Note that it is also consistent with the definition of contained_objects for mutable objects. See Section 2.8.1 [Formal Model of Objects], page 24 for how contained_objects is defined for mutable object sorts.) A disadvantage is that few sort errors in the modifies-clause will be caught, because contained_objects should be defined for most sorts. To have some error checking in the modifies-clause, Larch/C++ considers it an error when the meaning of any store-ref in a modifies-clause is the empty set of type-tagged objects.

6.2.3.3 Modifies and Const

In C++, declaring an object to be const means that the (top-level) bits in the object’s representation cannot change (see section 7.1.6 of [Ellis-Stroustrup90]). Because such an object’s bit representations cannot be changed (at least using a path starting at the object declared const), there can be no change in its abstract value. That is, const-ness implies immutability. Thus it is often an error to list as a store-ref a const object, i.e., a term whose sort is of the form ConstObj[T]. (A notable exception to this rule is that the traits constructed by default for structs and classes model their instances as const objects, see Section 6.2.3 [The Modifies Clause], page 110 for such usage.) For example, the following is an error, because *x has no contained objects.

```c++
// @(#)$Id: make_zero_or_one.lh,v 1.7 1997/06/03 20:30:14 leavens Exp $
extern void make_zero_or_one(const int* x) throw();
//@ behavior {
//@ requires assigned(x, pre);
//@ modifies *x; // error!
//@ ensures (*x)' = (*x)^ \mod 2;
//@ }
```

Similarly in a const member function, self should not appear in the modifies clause. This is because in a const member function of a class T, the sort of self is ConstObj[T], which will usually not have any contained objects. Recall that the notation self is short for *(this\any) and in the specification of a const member function, the sort of this is ConstObj[Ptr[ConstObj[T]]] (see Section 6.1.10.1 [This and Self], page 99). See Section r.9.3.1 of [Stroustrup91] for more details about const member functions.

One might think that the omission of const for a formal parameter (that is an object) would provide sufficient information to make the modifies-clause superfluous. But this is not so. The main problem is that const-ness in C++ is only a one-level guarantee. For example, if one of an object’s data members is a constant pointer to a mutable object (that is, the pointer variable is const), then the C++ guarantee of const-ness does not ensure that the abstract value of the object does not change, because the code could modify the object’s abstract value without changing the
pointer variable. So although it is often the case that \texttt{const}-ness implies no change in abstract value, that does not always hold true.

Conversely, a change to the bits of an object (its representation in C++ terms) does not necessarily mean that the abstract value of the object changes. For example, suppose rational numbers are represented by a pair of integers, written by $i/j$. Then a rational object $x$ whose abstract value is $1/2$ might be represented by the integers $2$ and $4$ ($2/4$) and by the integers $1$ and $2$ ($1/2$). In this case, changing the bits from $2$ and $4$ to $1$ and $2$ does not change the abstract value. (However, in this case, one would need to specify that the rational number object could be modified, and that its value depends on the two integer objects, but that its value is unchanged. This is needed to allow the implementation described above, see [Leino95b] and section 10.2.2 of [Leino95]. See Section 7.6 [The Depends Clause], page 213 for details.)

So in Larch/C++ one can specify both \texttt{const} and \texttt{modifies}, the one for C++ interfaces and the other for abstract behavior. (Thanks to J. Horning for a personal communication on this point.)

### 6.2.3.4 Formal Details of the Modifies Clause

A \textit{store-ref} can be any \textit{term} whose sort is \texttt{Set[TypeTaggedObject]} or of the form \texttt{Obj[T]} (or, as a syntactic sugar, a \textit{term} with a sort for which the trait function \texttt{contained_objects} is defined). See Section 2.8.1 [Formal Model of Objects], page 24 for sorts of the form \texttt{Obj[T]} and \texttt{ConstObj[T]}. See Section 6.2.3.5 [Reach], page 120 for the sort requirements of arguments to \texttt{reach}.

The sort \texttt{TypeTaggedObject} is defined in the following trait. The sort \texttt{TYPE} in that trait represents the Larch/C++ sort of the object, which would have the form \texttt{Obj[T]} or \texttt{ConstObj[T]} for some C++ type T. (See Section 2.8.1 [Formal Model of Objects], page 24 for the sorts \texttt{Obj[T]} and \texttt{ConstObj[T]}. See Section 2.8.2 [Formal Model of States], page 30 for the sort \texttt{Object}.)

```
% @(#)$Id: TypeTaggedObject.lsl,v 1.6 1995/11/09 23:02:43 leavens Exp $
TypeTaggedObject(TYPE): trait
  TypeTaggedObject tuple of obj: Object, type_tag: TYPE
```

The following summarizes the semantics of a function specification with a \textit{modifies-clause}. First a set of type-tagged objects is obtained from the \textit{modifies-clause}. Then the closure of this set is used to expand the set by including dependent objects [Leino95]. Then this set is used to construct a predicate, MP, which is conjoined to the written postcondition.
The set of type-tagged objects, UTTOs(store-ref-list), is obtained from the store-ref-list in a function’s modifies-clause as follows. If the store-ref-list is nothing, then UTTOs(store-ref-list) = {}. If the store-ref-list is everything, then UTTOs(store-ref-list) is the set of all type-tagged objects that are allocated in the pre-state. Otherwise, let SR be a store-ref in the store-ref-list of the modifies-clause. Let TTO(SR) be a Set[TypeTaggedObject] defined as follows.

- TTO(SR) = SR, if the sort of SR is Set[TypeTaggedObject] (this includes the case when SR has the form reach(term): see Section 6.2.3.5 [Reach], page 120 for the details in this case), and
- TTO(SR) = contained_objects(SR, pre), otherwise.

It is an error for a set TTO(SR) to be empty. Then, in the case where store-ref-list is not nothing or everything, UTTOs(store-ref-list) is the union of the sets TTO(SR) for each store-ref SR in the store-ref-list.

Let Closure(Env, UTTOs(store-ref-list)) be the closure of this set so that all objects in the environment Env on which the objects in UTTOs(store-ref-list) depend (recursively) are added (see Section 7.6 [The Depends Clause], page 213 and Chapter 11 of [Leino95]).

Let ModifiedObjects(pre, post) be the set such that for each typed object sort, Loc[T], and for each typed object loc of sort Loc[T], widen(loc) is in ModifiedObjects(pre, post) if and only if isModified(loc, pre, post) holds in the theory of TypedObj(Loc, T). This is summarized somewhat informally by the following.

\[
\text{ModifiedObjects}(\text{pre, post}) = \{ \text{widen}(\text{loc}) \mid \text{isModified}(\text{loc, pre, post}), \text{loc is a typed object} \}
\]

The predicate isModified(loc, pre, post) is only true if the type of loc is a type recorded in the state pre. This prevents arbitrary type perspectives from affecting whether an object is modified. Note that the notion of modification is essentially typed, because it depends on the notion of equality of abstract values, which is defined by the trait that specifies those abstract values. It would be wrong, and tantamount to comparing bits, to have defined this notion on untyped values.

The predicate MP is then the following. (Except for \texttt{\subseteq} from the trait Set, which is defined in the LSL Handbook of [Guttag-Horning93], the other trait functions are described following the predicate.)
ModifiedObjects(pre, post)
\subseteq ignoringTypeTags(Closure(Env,
UTTOs(store-ref-list)))

In the above predicate, the type-tags in the set Closure(Env, UTTOs(store-ref-list)) are not used. However, the reason for having these type-tags is not for the modifies clause, but for the semantics of reach (see Section 6.2.3.5 [Reach], page 120) and unchanged (see Section 6.2.3.6 [Unchanged], page 121). One does not want to compare type-tagged objects for the modifies clause, as that would prohibit cross-type aliasing and many uses of subtyping.

The meaning of the trait function modified for each sort of the form Loc[T] is given by the trait ModifiesSemantics(Loc, T) below. This trait would be instantiated for each such sort, so that it applies to the sort of loc in the formula above.

\% This is based on Chalin's help and his work [Chalin95].
ModifiesSemantics(Loc, T): trait
  assumes State, AllocatedAssigned(Loc, T), TypedObjEval(Loc, T)
  introduces
    isModified: Loc[T], State, State -> Bool
  asserts
    \forall loc: Loc[T], pre,post: State
    isModified(loc, pre, post) ==
    (assigned(loc, pre) \text{ } \land \text{ } assigned(loc, post)
    \land \text{ } eval(loc,pre) \neq eval(loc,post))
    \lor \text{ } (allocated(loc, pre) \land \neg assigned(loc, pre)
    \lor \text{ } assigned(loc,post));
  implies
    \forall loc: Loc[T], pre,post: State
    isModified(loc, pre, post)
    \Rightarrow (allocated(loc, pre) \land assigned(loc, post));
  converts isModified
    exempting \forall loc: Loc[T], st: State
    isModified(loc,bottom,st), isModified(loc, st,bottom)

See Section 2.8.2 [Formal Model of States], page 30 for the assumed trait State. See Section 6.2.2 [Allocated and Assigned], page 107 for the assumed trait AllocatedAssigned. See Section 2.8.1 [Formal Model of Objects], page 24 for the trait TypedObj which includes ModifiesSemantics.

The trait function ignoringTypeTags is defined by the following trait.
Ignoring redundancy, and the `trashes-clauses` and `calls-clauses`, the meaning of a function specification with a `modifies-clause` is the following. A C++ function satisfies its specification if and only if for each type-correct function call, if the precondition predicate is satisfied in the pre-state, then the function call terminates, the function mutates at most the set of objects described in the `modifies-clause`, and the postcondition is satisfied by the pre-state and the post-state. (It should be understood that the desugared forms are used for the precondition and postcondition.) Ignoring redundancy, termination, and the `trashes-clause-seq` and `calls-clause-seq`, one can write the predicate that must be satisfied by the pre and post-states as follows, where MP is the predicate that describes the modifies clause (as defined above).

\[
\text{desugar}(\text{pre-cond}) \Rightarrow (\text{desugar}(\text{post-cond}) \land \text{MP})
\]

As an example, consider the following specification.

```cpp
//@ behavior {
//@ modifies i;
//@ ensures assigned(i, post) \land i’ = 1;
//@ }
```

The predicate that characterizes the relation specified for the function `set_ref_to_one` in the above specification is as follows. (The variable `residue_i` introduced by the Closure operator records whatever dependencies that may exist on `i` but which are not in scope; see section 11.3 of [Leino95].)
\[\text{allocated}(i, \text{pre}) \Rightarrow (\text{assigned}(i, \text{post}) \land (\text{eval}(i, \text{post}) = 1) \land \text{ModifiedObjects}(\text{pre}, \text{post}) \land \subseteq \text{ignoringTypeTags}(\{\text{asTypeTaggedObject}(i)\} \cup \{\text{asTypeTaggedObject}(\text{residue}_i)\});\]

Because the trait function \text{ignoringTypeTags} takes off the type tag, the above can be written more simply as follows.

\[\text{allocated}(i, \text{pre}) \Rightarrow (\text{assigned}(i, \text{post}) \land (\text{eval}(i, \text{post}) = 1) \land \text{ModifiedObjects}(\text{pre}, \text{post}) \land \subseteq (\{\text{widen}(i)\} \cup \{\text{widen}(\text{residue}_i)\});\]

### 6.2.3.5 Reach

Informally, an \text{icpp-primary} of the form \text{reach}(x) denotes the set of all objects reachable from \(x\). One way of thinking of this is that it is the set of objects to which a pointer could be returned by a C++ function that takes \(x\) as an argument. This includes \(x\) itself.

For example, consider the following global variable declaration.

```c
struct ratl { int num; int den; };
ratl r;
```

In the above example, \text{reach}(r) is a set containing three type-tagged objects: the object \(r\) itself, and the \text{num} and \text{den} fields of \(r\).

To formalize these intuitions, it is necessary for the Larch/C++ user to explicitly say what objects are (directly) contained in each sort of abstract value. This is done by defining the trait function \text{contained_objects}. For example, consider how this is done for a built-in C++ type. The trait for \text{struct ratl} contains a definition of a trait function \text{contained_objects}, which says that in an abstract value \(r^\ast\), the directly contained objects are \(r^\ast\text{.num}\) and \(r^\ast\text{.den}\). (To be technical, these are made into type-tagged objects, see Section 7.5 [Contained Objects], page 207 for details. See Section 11.10 [Structure and Class Types], page 299 for details of structs.)

The sort of the argument to \text{reach} must be a sort of the form \text{Obj}[T] or \text{ConstObj}[T] for some \(T\). The sort of the result is the sort \text{Set[TypeTaggedObject]}. (See Section 6.2.3 [The Modifies Clause], page 110 for the trait \text{TypeTaggedObject}.)}
Formally, the set of objects, \( \text{reach}(x) \), is the smallest set such that the following holds.

- \( \text{asTypeTaggedObject}(x) \) is in the set, and
- if a type-tagged object, such as \( \text{asTypeTaggedObject}(o) \), is in the set, then the set \( \text{contained}_\text{objects}(o^\text{pre}) \) is a subset of the set. (The type tag is needed here, because the exact version of \( \text{contained}_\text{objects} \) depends on the sort of \( o \), which is recorded in a type-tagged object.)

See Section 7.5 [Contained Objects], page 207 for the trait function \( \text{asTypeTaggedObject} \).

### 6.2.3.6 Unchanged

For some functions, there may be cases in which objects that are permitted to be modified are not modified. An \( \text{lcpp-primary} \) starting with \code{unchanged} asserts that its argument objects are not mutated (see Section 6.2 [Mutation], page 101). For example, \code{unchanged(a)} means that, if \( a \) were assigned in the pre-state, then it has the same abstract value in the post-state as in the pre-state (and if it were allocated but not assigned in the pre-state, then it did not become assigned in the post-state). The \( \text{lcpp-primary} \) \code{unchanged(everything)} asserts that no object is mutated. Such \( \text{lcpp-primary} \) forms can only be used in a postcondition.

For example, \code{unchanged(t1,\text{reach(t2)})} asserts that \( t1 \) and all objects reachable from \( t2 \) are not mutated (and not deallocated).

As another example, given a specification of \code{BankAccount} (which itself includes a specification of \code{Money}), the following specification says that if in a call \code{withdraw(x,m)} the value of \( m \) is greater than the value of the \code{credit} field in the value of \( x \), then the call has no effect.

```c
//@ behavior {
//@ requires assigned(a, pre) /
//\ amt >= 0;
//@ modifies a^.credit;
//@ ensures if a^.credit^ >= amt
//@ then a'.credit' = a^.credit - amt
//@ else unchanged(a^.credit);
//@ }
```
Formally, the meaning of \texttt{unchanged(store-ref-list)} is given as follows. A set of type-tagged objects, UTTOs\texttt{(store-ref-list)}, is obtained from the \texttt{store-ref-list} exactly as described above for the \textit{modifies-clause} (see Section 6.2.3 [The Modifies Clause], page 110). Then \texttt{unchanged(store-ref-list)} denotes the conjunction, for each type-tagged object \texttt{tto} in UTTOs\texttt{(store-ref-list)}, of terms of the following form (where \texttt{tto.type_tag} represents the LSL sort \texttt{Loc[T]}).

\[ \neg \text{isModified(narrow(tto.obj):Loc[T], pre, post)} \]

In the above predicate, the trait function \texttt{narrow} converts an untyped object (the result of \texttt{tto.obj}) to a typed object of the type given by the type tag recorded in \texttt{tto}. The sort suffix (:\texttt{Loc[T]}) is included to say what overloading of \texttt{narrow} is desired, namely the one of from the trait \texttt{TypedObj(Loc, T)}, where \texttt{Loc}, and \texttt{T} are from the type tag recorded in \texttt{tto}. This is one reason why type-tagged objects are needed, as modification is a concept defined by the notion of equality for each kind of abstract value.

For example, the postcondition of \texttt{withdraw} as specified above means the following.

\begin{verbatim}
if a^.credit^ >= amt
then a'.credit' = a^.credit - amt
else \neg isModified(narrow(
    asTypeTaggedObject(eval(a,pre).credit).obj):Obj[Money],
    pre, post)
\end{verbatim}

This can be simplified to the following, using the fact that widening and narrowing a typed object results in the same typed object.

\begin{verbatim}
if a^.credit^ >= amt
then a'.credit' = a^.credit - amt
else \neg isModified(eval(a,pre).credit, pre, post)
\end{verbatim}

(The advantage of the more complex formal semantics is that it works in more general situations.)

An \texttt{lcpp-primary} starting with \texttt{unchanged} has sort \texttt{Bool}. See Section 6.2.3 [The Modifies Clause], page 110 for the rules for the sorts allowed for a \texttt{store-ref}.

See Section 6.10 [Case Analysis], page 143 for another example of \texttt{unchanged}. 
6.3 Allocation and Deallocation

In Larch/C++, the traits for objects, and the built-in C++ types have trait functions that allow you to specify whether an object is allocated or not. The trait function for this purpose is named `allocated`, and it has many overloading for different sorts. Generally, it takes an object and a state (such as `pre` or `post`) as an argument. See Section 6.2.2 [Allocated and Assigned], page 107 for the trait functions that are part of the built-in types.

In addition, Larch/C++ provides syntactic support for asserting that an object is newly allocated (`fresh`), and for specifying what objects are considered “trashed” or potentially deallocated by a function (the `trashes-clause` and `trashed`). Unless the `trashes-clause` is used, a function is not allowed to deallocate objects. These ideas are discussed in the subsections that follow.

6.3.1 Fresh

A `lepp-primary` beginning with `fresh`, for example `fresh(x)`, can only be used in a postcondition. (See Section 6.1.10 [Larch/C++ Special Primaries], page 99 for the exact syntax.) Informally, `fresh(x)` says that the object denoted by the term `x` was not allocated in the pre-state, and is allocated in the post-state. Writing `fresh(t1,t2)` says that both `t1` and `t2` are newly allocated, and that, in addition, `t1` and `t2` are distinct objects.

A common use of `fresh` is to state that the result of a function call is newly allocated. For example, consider the following, which says that the result of a call to `make_ratl` must be a pointer to newly allocated memory.

```plaintext
// @(#)$Id: make_ratl.lh,v 1.7 1997/06/03 20:30:13 leavens Exp $
typedef int *ratl;
extern ratl make_ratl(int n, int d) throw();
//@ behavior {
//@ requires d > 0;
//@ ensures assigned(result, post) \ size(locs(result)) = 2
//@ \ (result[0])' = n \ (result[1])' = d
//@ \ fresh(result[0], result[1]);
//@ }
```

The terms in the `term-list` passed to `fresh` should usually be of sort `Set[TypeTaggedObject]` or a sort of the form `Obj[T]` or `ConstObj[T]`, for some sort `T`; alternatively, as a syntactic sugar, they can be terms of a sort for which the trait function `contained_objects` is defined. The syntactic
sugar obtains a set of objects (which must not be empty) from a value by applying contained_objects to the value and the post-state.

(Since the post-state is used, the syntax of a store-ref-list is not used with fresh, because the syntactic sugar for a store-ref-list uses the pre-state.)

The semantics of fresh(term-list) is as follows. Let E be a term in the term-list. Let TTO(E) be defined as follows.

- TTO(E) = E, if the sort of E is Set[TypeTaggedObject],
- TTO(E) = {asTypeTaggedObject(E)}, if the E has a sort of the form Obj[T] or ConstObj[T], and
- TTO(E) = contained_objects(E, post), otherwise.

Then ignoringTypeTags(TTO(E)) is a set of untyped objects obtained from TTO(E) by taking off each type tag from each object in TTO(E). Let AssertedFresh(term-list) be the union of the sets TTO(E), for each term E in term-list. Somewhat informally, this is summarized as follows.

\[ \text{AssertedFresh(term-list)} = \{ \text{tto} \mid \text{tto is in TTO(E)}, \text{E is a term in term-list} \} \]

Then fresh(term-list), means that for each E in term-list, the sets ignoringTypeTags(TTO(E)) are mutually disjoint, and that the following is true, for each typed object sort Loc[T].

\[ \forall \text{loc:Loc[T]} \]  
\[ (\text{loc} \in \text{AssertedFresh(term-list)} \Rightarrow \text{isFresh}(\text{loc, pre, post})) \]

For example, the term fresh(result[0], result[1]) used in the above example has the following meaning, because the only sort in question is Obj[int].

\[ (\text{ignoringTypeTags} \{\text{asTypeTaggedObject}(\text{result[0]})\}) \]  
\[ \\setminus (\text{ignoringTypeTags} \{\text{asTypeTaggedObject}(\text{result[1]})\}) = \text{\{\}} \]  
\[ (\forall \text{\forall loc:Obj[int]} \]  
\[ (\text{loc} \in (\text{\{asTypeTaggedObject(result[0])\}} \setminus (\text{\{asTypeTaggedObject(result[1])\}}) \Rightarrow \text{isFresh}(\text{loc, pre, post})) \]
The above can be simplified, using facts about type-tagged objects, to the following.

\[
\{\text{widen}(\text{result}[0])\} \setminus \{\text{widen}(\text{result}[1])\} = \{
\quad \land \quad \text{isFresh}(\text{result}[0], \text{pre}, \text{post}) \land \text{isFresh}(\text{result}[1], \text{pre}, \text{post})
\]

The trait \text{FreshSemantics} below gives the definitions of the trait function \text{isFresh} used in the semantics. (See Section 6.2.3 [The Modifies Clause], page 110 for the trait \text{TypeTaggedObject} that defines the trait function \text{ignoringTypeTags}.)

\[

\text{FreshSemantics}(\text{Loc}, T): \text{trait}
\begin{align*}
\text{assumes} & \text{AllocatedAssigned}(\text{Loc}, T) \\
\text{introduces} & \text{isFresh}: \text{Loc}[T], \text{State}, \text{State} \rightarrow \text{Bool} \\
\text{asserts} & \forall \text{loc}: \text{Loc}[T], \text{pre}, \text{post}: \text{State} \\
& \text{isFresh}(\text{loc}, \text{pre}, \text{post}) \\
& = \neg \text{allocated}(\text{loc}, \text{pre}) \land \text{allocated}(\text{loc}, \text{post}) \\
\text{implies} & \text{converts \text{isFresh}} \\
& \text{exempting} \forall \text{loc}: \text{Loc}[T], st: \text{State} \\
& \text{isFresh}(\text{loc}, \text{bottom}, st), \text{isFresh}(\text{loc}, st, \text{bottom})
\end{align*}

The sort of a \text{lcpp-primary} of the form \text{fresh}(t) is \text{Bool}.

In the specification of \text{make_ratl} given above, and in similar cases, it is inconvenient to have to explicitly list each element of an array pointed at as fresh. But since the trait function \text{contained_objects} is defined for pointers (see Section 11.8 [Pointer Types], page 287), the syntactic sugar described above allows one to write in the specification of \text{make_ratl fresh} result as a shorthand for \text{fresh} result[0], result[1]). In general, when \text{a} is a pointer or an array name, then \text{fresh} a means the \text{fresh} contained_objects a, post). This sugar works for multi-dimensional arrays as well; that is, when \text{a} is the name of a multi-dimensional array \text{fresh} a means that each element is fresh. (See Section 11.7 [Array Types], page 279 for details on the trait function \text{contained_objects} for arrays).

For \text{struct}s, one has to pass to \text{fresh} both the \text{struct} object itself, and the contained objects. This can be done, as in the first conjunct of the following specification’s postcondition. (See Section 11.10 [Structure and Class Types], page 299 for the abstract values of \text{struct}s.)
struct sratl { int num; int den; }
extern sratl& make_sratl(int n, int d) throw();
//@ behavior {
//@ requires d > 0;
//@ ensures fresh(result, result'.num, result'.den)
//@ /
//@ /
//@ /
//@ }

In the above specification, \texttt{fresh(result, result'.num, result'.den)} could also have been written as one of the two following terms.

\begin{itemize}
  \item fresh(result, result')
  \item fresh(result, contained\_objects(result', post))
\end{itemize}

If \(u\) is a union object, then usually one would write \texttt{fresh(u, contained\_objects(u', post))} to assert that \(u\) and its fields are fresh. See Section 11.11 [Union Types], page 302 for more details on the abstract values of unions.

### 6.3.2 The Trashes Clause

When an object is trashed, it can no longer be used. In the Larch/C++ model of C++, a function call can make this can happen in the following ways.

- The object was assigned in the pre-state, but is not assigned in the post-state (i.e., it is made unassigned).
- The object was allocated in the pre-state, but is not allocated in the post-state.

Functions that trash objects must include a \texttt{trashes-clause-seq}. If a trashes clause is not included, then the function may not trash any objects. If such a clause is included, then only the objects described may be trashed. An object is described by the \texttt{trashes-clause} if it is listed explicitly, or if it depends on a described object. This means that objects that are not described in the \texttt{trashes-clause-seq} cannot be made unassigned or deallocated. The objects described in the \texttt{trashes-clause-seq} do not have to be trashed, the point is that they are allowed to be trashed.
Because objects that are not described by the trashes-clause-seq are not allowed to be trashed, the postcondition of a function specification does not have to mention that all other objects stay allocated or assigned. Hence, the trashes-clause-seq, along with the modifies-clause-seq, gives a complete frame axiom [Borgida-etal95] for the function specification.

The idea of the trashes-clause-seq and its utility in making Larch-style specifications referentially transparent and less verbose is taken from Chalin [Chalin95].

The trashes-clause-seq has a syntax that is very similar to the syntax of the modifies-clause-seq.

\[
\text{trashes-clause-seq ::= trashes-clause [ trashes-clause ] \ldots} \\
\text{trashes-clause ::= trashes [ redundantly ] store-ref-list ;}
\]

See Section 6.9.4 [Redundancy in Frames], page 143 for the meaning of redundantly used in a trashes-clause. When several non-redundant trashes-clauses are listed in a trashes-clause-seq, this is the same as listing each of their store-ref-lists in a single trashes-clause. If more than one non-redundant trashes-clause is given, then none of the store-ref-lists may be of the form nothing or everything. Because it is possible to translate multiple non-redundant trashes-clauses into a single trashes-clause, we will assume from now on that there is only one non-redundant trashes-clause.

For example, the following specifies a function that deallocate an integer object. The requires-clause states that the pointer passed must be a non-null pointer, and that it points to allocated storage. (Equivalently, this could have been stated as \text{allocated}(p, pre), see Section 11.8 [Pointer Types], page 287 for the details of these trait functions.) The trashes-clause says that the function is allowed to, but does not have to, deallocate or make unassigned the object \( *p \). The postcondition states that \( *p \) must be deallocated. The redundant postcondition follows from the meaning of \( \sim \text{allocated}(p, post) \) in the pointer traits. (See Section 6.9.3 [Redundant Ensures Clauses or Claims], page 141 for the meaning of a redundant ensures-clause.)

\[
.extern void dealloc_int_obj(int *p) throw(); \\
//@ behavior { \\
//@ requires isValid(p) \&\& \text{allocated}(\*p, pre); \\
//@ trashes \*p; \\
//@ ensures \sim \text{allocated}(\*p, post); \\
//@ ensures redundantly \sim \text{allocated}(p, post); \\
//@ }
\]
Other topics related to the *trashes-clause* are discussed below.

### 6.3.2.1 Trashed

An *lcpp-primary* such as `trashed(t1,t2)` can be used in specifying a function that must deallocate or "finalize" an object. It can also be used in specifying C++ destructors for type of objects whose abstract values contains objects that are visible to clients. (See Section 7.2.2 [Destructors], page 190 for discussion and examples. Note, however, that in Larch/C++, it is a mistake to write `trashed(self)` in a postcondition.)

An *lcpp-primary* of the form `trashed(term-list)` can only be used in the postcondition. Furthermore, all objects described by the *term-list* must be described by the *store-ref-list* in the function's *trashes-clause* (see Section 6.3.2 [The Trashes Clause], page 126).

The *lcpp-primary* `trashed(t1,t2)` states that the objects *t1* and *t2* are to be either uninitialized or deallocated. Formally it is translated into the following term.

\[
(is\text{Trashed}(t1, pre, post) \land is\text{Trashed}(t2, pre, post))
\]

\(^{2}\) See Section 6.3.2.2 [Formal Details of the Trashes Clause], page 130 for the trait function `isTrashed`.

The sort of `trashed(term-list)` is `Bool`.

A simple example is the following specification. It must be implemented by a function that makes *ir* either unassigned or deallocated in the post-state.

```c
/** \@(#)$Id: done_with.lh,v 1.5 1997/06/03 20:30:02 leavens Exp $$

extern void done_with(int & ir) throw();
//@ behavior {
//@ trashes ir;
//@ ensures trashed(ir);
//@ }
```

\(^{2}\) Thanks to Rustan Leino, Mark Vandevoorde, Dave Detlefs, Yang Meng Tan, and Patrice Chalin for a series of personal communications about the semantics of `trashed`. 

Since \texttt{trashed(x)} does not necessarily mean that \(x\) is deallocated, one should use negation and the trait function \texttt{allocated} explicitly if one wishes to specify that an object must be deallocated, as opposed made unassigned. (See Section 6.3.2 [The Trashes Clause], page 126 for an example.) Similarly, if one wishes to specify that objects must be made unassigned, then that can be stated using negation and the trait function \texttt{unassigned}. See Section 6.2.2 [Allocated and Assigned], page 107 for details on these trait functions.

In the following example, when the reference count \texttt{ref\_count^} is 1, the object \texttt{*cp} is trashed, and otherwise it is not trashed.

```c
// @(#)$Id: dec_ref.lh,v 1.7 1997/09/16 02:56:30 leavens Exp $
extern void dec_ref(char *cp, int & ref\_count) throw();
//@ behavior {
//@ requires allocated(cp, pre) \/ assigned(ref\_count, pre)
//@ \/ ref\_count^ >= 1;
//@ modifies ref\_count;
//@ trashes *cp;
//@ ensures ref\_count’ = ref\_count^ - 1
//@ \/ (if ref\_count’ = 0 then trashed(*cp)
//@ else \sim trashed(*cp));
//@ }
```

However, there are cases when one wants to leave it up to the implementation whether to deallocate or not. An example would be when one might want to let the implementation use a garbage collector. In such cases, one would just mention the objects in question in the \texttt{trashes-clause} but not specify that they must be trashed by using \texttt{trashed} in the postcondition.

An example is the function \texttt{chaos}, which terminates, but can have any effect [Nelson89] [Hesselink92]. (Such a function is not useful for much, but the specification demonstrates the expressiveness of Larch/C++. The reason \texttt{chaos} can have any effect is that everything (every object) is threatened by the function, and everything can be both modified or trashed (but is not required to be modified or trashed).
extern void chaos();
//@ behavior {
//@ extern everything;
//@ modifies everything;
//@ trashes everything;
//@ ensures true;
//@ }

The terms in the term-list passed to trashed should usually be of sort \texttt{Set[TypeTaggedObject]} or have the form \texttt{Obj[T]} or \texttt{ConstObj[T]}; as with \texttt{fresh}, as a syntactic sugar, terms of a sort for which the trait function \texttt{contained_objects} is defined may also be used as arguments to \texttt{trashed}. A set of objects is extracted from these terms in exactly the same way as for \texttt{fresh} (see Section 6.3.1 [Fresh], page 123), and for each such object \texttt{o} it is asserted that \texttt{isTrashed(o,pre,post)}. See Section 6.3.1 [Fresh], page 123 for the details of the syntactic sugars that apply.

6.3.2.2 Formal Details of the Trashes Clause

In a \texttt{trashes-clause} the same sort restrictions and syntactic sugars apply as for the \texttt{modifies-clause}. Suffice it to say that each \texttt{store-ref} should be a \texttt{term} whose sort is \texttt{Set[TypeTaggedObject]} or of the form \texttt{Obj[T]} (or, as a syntactic sugar, a \texttt{term} with a sort for which the trait function \texttt{contained_objects} is defined). (See Section 6.2.3.4 [Formal Details of the Modifies Clause], page 116 for details.)

The following summarizes the semantics of a function specification with a \texttt{trashes-clause}. First a set of type-tagged objects is obtained from of the \texttt{trashes-clause}. This set is closed to take dependencies into account. Then this set is used to construct a predicate, \texttt{TP}, which is conjoined to the written postcondition.

The set of type-tagged objects, UTTOs(\texttt{store-ref-list}), is obtained from the \texttt{store-ref-list} in a function’s \texttt{trashes-clause} exactly as with the \texttt{modifies-clause}. Recall that this means that UTTOs(\texttt{store-ref-list}) be the union of the sets TTO(SR) of type tagged-objects that are the denotations of each \texttt{store-ref} SR in the \texttt{store-ref-list}. (See Section 6.2.3.4 [Formal Details of the Modifies Clause], page 116 for details.)
Recall also that the set $\text{Closure}(\text{Env}, \text{UTTOs}(\text{store-ref-list}))$ is the closure of this set so that all objects in the environment $\text{Env}$ on which the objects in $\text{UTTOs}(\text{store-ref-list})$ depend (recursively) are added (see Section 7.6 [The Depends Clause], page 213 and Chapter 11 of [Leino95]).

Let $\text{TrashedObjects}(\text{pre}, \text{post})$ be the set such that for each typed object sort, $\text{Loc}[T]$, and for each typed object $\text{loc}$ of sort $\text{Loc}[T]$, $\text{widen}(\text{loc})$ is in $\text{TrashedObjects}(\text{pre}, \text{post})$ if and only if $\text{isTrashed}(\text{loc}, \text{pre}, \text{post})$ holds in the theory of $\text{TypedObj}(\text{Loc}, T)$. This is summarized somewhat informally by the following.

$$\text{TrashedObjects}(\text{pre}, \text{post}) = \{ \text{widen}(\text{loc}) | \text{isTrashed}(\text{loc}, \text{pre}, \text{post}), \text{loc} \text{ is a typed object} \}$$

Recall that $\text{isTrashed}(\text{loc}, \text{pre}, \text{post})$ is only true if the type of $\text{loc}$ is a type recorded in the state $\text{pre}$. This prevents arbitrary type perspectives from affecting whether an object is trashed. Note that the notion of being trashed is typed, because it depends on the notion of when an object is unassigned. (However, this is just the way the notions fall out in the Larch/C++ traits; there is nothing essential about having this notion be typed.)

The predicate $\text{TP}$ is then the following. (Except for $\subseteq\text{subseteq}$ from the trait $\text{Set}$, which is defined in the LSL Handbook of [Guttag-Horning93], the other trait functions are described following the predicate.)

$$\text{TrashedObjects}(\text{pre}, \text{post}) \subseteq \text{ignoringTypeTags}(\text{Closure}(\text{Env}, \text{UTTOs}(\text{store-ref-list})))$$

In the above predicate, the type-tags in the set $\text{Closure}(\text{Env}, \text{UTTOs}(\text{store-ref-list}))$ are not used. However, the reason for having these type-tags is not for the trashes clause, but for the semantics of $\text{reach}$ (see Section 6.2.3.5 [Reach], page 120) and $\text{unchanged}$ (see Section 6.2.3.6 [Unchanged], page 121). One does not want to compare type-tagged objects for the trashes clause, as that would prohibit cross-type aliasing and many uses of subtyping.

The meaning of the trait function $\text{isTrashed}$ for each sort of the form $\text{Loc}[T]$ is given by the trait $\text{TrashesSemantics}(\text{Loc}, T)$ below. This trait would be instantiated for each such sort, so that it applies to the sort of $\text{loc}$ in the formula above.
introduces
isTrashed: Loc[T], State, State -> Bool

asserts
\[\forall loc: \text{Loc}[T], \text{pre, post: State} \]
\[\text{isTrashed(loc, pre, post)}\]
\[\Leftarrow (\text{assigned(loc, pre)} \land \neg \text{assigned(loc, post)})\]
\[\lor (\text{allocated(loc, pre)} \land \neg \text{allocated(loc, post)})\];

implies
\[\forall loc: \text{Loc}[T], \text{pre, post: State} \]
\[\text{isTrashed(loc, pre, post)} \Rightarrow (\text{allocated(loc, pre)}\]
\[\lor (\neg \text{allocated(loc, post)} \lor \neg \text{assigned(loc, post)})\];
\[\neg \text{isTrashed(loc, pre, post)}\]
\[\Rightarrow \neg \text{assigned(loc, post)};\]
\[\neg \text{isTrashed(loc, pre, post)} \lor \text{assigned(loc, pre)}\]
\[\Rightarrow \text{allocated(loc, post)};\]
\[\neg \text{isTrashed(loc, pre, post)} \lor \text{allocated(loc, pre)}\]
\[\Rightarrow \text{allocated(loc, post)};\]

converting isTrashed
exempting \[\forall loc: \text{Loc}[T], \text{st: State} \]
\[\text{isTrashed(loc, bottom, st)}, \text{isTrashed(loc, st, bottom)}\]

See Section 2.8.2 [Formal Model of States], page 30 for the assumed trait State. See Section 6.2.2 [Allocated and Assigned], page 107 for the assumed trait AllocatedAssigned. See Section 2.8.1 [Formal Model of Objects], page 24 for the trait TypedObj which includes ModifiesSemantics.

### 6.4 The Calls Clause

\[
calls-clause-seq ::= \text{calls-clause} [ \text{calls-clause} ] \ldots
\]
\[
calls-clause ::= \text{calls} [ \text{redundantly} ] \text{function-names} ;
\]
\[
function-names ::= \text{everything} | \text{nothing}
\]
\[
\text{function-name} [ , \text{function-name} ] \ldots
\]
\[
function-name ::= \text{term}
\]

A calls-clause says what functions may be directly called by a correct implementation of the function being specified. This is helpful for documenting the calling pattern among virtual functions, which is needed to write subclasses [Kiczales-Lamping92] [Steyaert-etal96]. The term that represents a function-name can use the values of variables in either the pre- or post-state to access functions. The following is an example of the use of the calls-clause.
class hclass {
public:
    static int sfun(int i);
    virtual int h();
};

class CallsExample {
public:
    hclass dmember;
    virtual void mymethod();

    virtual void ex(hclass vparam, hclass & rparam, hclass *p);
    //@ behavior {
    //@ modifies *p;
    //@ calls vparam.h, rparam\any.h, (*p)^.h, dmember\any.h,
    //@ hclass::sfun, mymethod;
    //@ }
    //@ }
};

This calls-clause says that the member function ex of the class CallsExample may call method h found in: the value parameter vparam, the value of the reference parameter rparam in either the pre- or post-state, the pre-state value of the pointer variable *p, and either the pre- or post-state value of the data member dmember. It may also call the static function hclass::sfun, and mymethod.

[[[Detailed semantics to be written.]]]

6.5 The Accesses Clause

accesses-clause-seq ::= accesses-clause [ accesses-clause ] ...
accesses-clause ::= accesses [ redundantly ] store-ref-list ;

An accesses-clause says what objects may be read by a correct implementation of the function being specified. By default an omitted accesses-clause is equivalent to accesses everything;, which imposes no constraints on implementations.

[[[Detailed semantics to be written.]]]
6.6 Default Arguments

Default values of formal arguments can be given in a function specification as part of the specification’s interface. The syntax is the same as that of C++. See Section 5.4.6 [Function Declarations], page 72 for the syntax.

For example, consider the following function specification.

```c++
extern float interest(float x, float rate = 0.05) throw();
//@ behavior {
//@ requires 0.0 <= rate /
//@ ensures result = x * rate;
//@ }
```

The function `interest` takes two float values denoted by `rate` and `x` respectively, and computes interest based on `rate`. The default value of `rate` is specified to be `0.05`; that is, if no value is supplied for `rate` on function invocation, `0.05` is used by default.

6.7 Global Variables

In a Larch/C++ function specification, one can declare the global variables used in the specification of a function in the `declaration-seq` section of the specification’s `fun-spec-body`. While this is not necessary if the variables were previously declared in a surrounding scope, declaring them makes clear to the reader of the specification what all the objects are that the function needs as “implicit arguments.” (They are called “implicit arguments” because the function depends, in some way on each of these arguments. In the jargon of program verification, such variables are called “threatened.”)

To mesh with C++ syntax, and to avoid confusing the reader of the specification, the declarations of such threatened variables should start with the keyword `extern`.

In the specification below, the global variable `current_token` of type `Token` is referenced in the function `next_token`; to warn the reader, this variable is explicitly declared in the `declaration-seq` clause.
A declaration of the form `extern everything;` means that the function threatens all objects. It might be used if one wants to warn the reader that the function may access a large number of global variables. However, all the global variables explicitly mentioned in the specification must have been previously declared.

A C++ function that uses global variables can be modeled mathematically as a function with implicit (reference) arguments for each global variable the C++ function uses. Since the data members of an instance are not global variables, they are not listed as such.

### 6.8 Let Clauses

A `let-clause` can be used in a function specification to avoid repeating the same `term` several times within a `spec-case`. That is, a `let-clause` allows abbreviation within the rest of a `spec-case`. If a specification has more than one `spec-case` the scope of the abbreviations does not extend to subsequent `spec-cases`. However, the scope of the abbreviations introduced by a `let-clause` can be extended to several `spec-cases` by enclosing them in a pair of curly brackets (`{ }`). See Chapter 6 [Function Specifications], page 82, for the details of the syntax. See Section 6.10 [Case Analysis], page 143, for the meaning of a specification with multiple or nested `spec-cases`.

The syntax of the `let-clause` itself is as follows. See Section 6.1.3.2 [Quantifiers], page 88 for the syntax of `varId` and `sort-name`.
As an example, the following specification uses a let-clause to avoid the use of ^ in the post-condition [Jonkers91]. Informally, it says that a call to transfer moves the prescribed amount of money from the source account to the sink account.

A specification using a let-clause is syntactic sugar for a specification written without it. The meaning of such a specification is given by textually replacing the free occurrences of the defined variables with their definitions. Bindings are allowed to depend on previous bindings, so technically this replacement process starts by substituting the term which is bound to the last name. The last name’s definition is substituted for that name, and then this desugaring recurses by processing the next to last name, and so on. In the example above, one would replace oldsink by presink.credit first, then continue backwards working on oldsrc, then presink, and finally presrc.

For example, the specification of transfer given above is equivalent to the following.
Chapter 6: Function Specifications

6.9 Redundancy in Function Specifications

Following the lead of LSL and Tan’s work on LCL [Tan94], Larch/C++ includes several features that allow one to write checkable redundancy into specifications. Tan’s work featured claims, which included the redundant postconditions found in Larch/C++. To this Larch/C++ adds several new kinds of redundancy: redundant preconditions, redundant framing, and examples. Examples were first described in [Leavens96]. The others are described in [Leavens-Baker99].

All redundant clauses do not affect the meaning of a specification. Instead they introduce things that one can check (e.g., with the Larch Prover).

Besides allowing for sanity checking of specifications, their main use is to point out things to the reader of the specification that are not needed otherwise. Stating them in the non-redundant parts of the specification, however, might cause people who are implementing or reasoning about the specification to do extra work. Also, when one has more than one way to specify a function, it may be best to retain both ways, putting one in a redundant clauses.

6.9.1 Examples in Function Specifications

An example-seq can be used to give the reader a concrete example of the behavior of a function. Such examples do not change the meaning of a specification, and could, of course, be written as comments in a specification. However, by making examples part of a Larch/C++ specification, one
introduces checkable redundancy. That is, it one can check that the relationship between the pre- and post-states described in the example satisfies the specification.

The syntax is as follows.

```
example-seq ::= example [ example ] ...
example ::= example [ liberally ] predicate ;
```

For example, in the following, the example listed shows the effect of transferring 100 dollars from the source account to the sink account.

```
#include "BankAccount.lh"

extern void transfer(BankAccount& source, BankAccount& sink, Money amt)
    throw();
//@ behavior {
//@ requires source ~ sink \&\& assigned(sink, pre) \&\& assigned(source, pre)
//@    \&\& source^.credit^ >= amt \&\& amt >= 0;
//@ modifies source^.credit, sink^.credit;
//@ ensures sink'.credit' = sink^.credit^ + amt
//@    \&\& source'.credit' = source^.credit^ - amt;
//@ example amt = money(100/1)
//@    \&\& source^.credit^ = money(500/1) \&\& sink^.credit^ = money(200/1)
//@    \&\& source'.credit' = money(400/1) \&\& sink'.credit' = money(300/1);
//@ }
```

A pair of example states would satisfy a specification when the pre-state violates the precondition; and in such an example, the poststate need not satisfy the postcondition. However, psychologically, such an example would be confusing, and so such examples are to be avoided. An example could also be confusing by violating the specified frame; and so that should be avoided as well.

You can check that an example does not violate the precondition and frame of the example, by checking the consistency of the following formula. That is, what should be checked to validate the consistency of an example with respect to the spec-case is the following, where PreCondition is the spec-case’s (desugared) precondition, MP is the predicate that codes its modifies-clause (see Section 6.2.3 [The Modifies Clause], page 110), TP is the predicate that codes its trashes-clause (see Section 6.3.2 [The Trashes Clause], page 126), and Example is the predicate from the example.
One way to prove the consistency of such a formula is to prove that this formula does not imply false. One can also construct a pair of states that satisfies it.

When a specification does not completely determine the result of a function (i.e., when it is incomplete), one should give several examples. Otherwise readers who are not careful may assume that the single example given illustrates all the possibilities. For example, the following gives two examples for the integer square root specification.

```c
// @(#)$Id: isqrt2.lh,v 1.6 1997/09/16 03:03:30 leavens Exp $
extern int isqrt(int x) throw();
//@ behavior {
//@ requires x >= 0;
//@ ensures (result-1)*(result-1) < x
//@ /
//@ example x = 28 /
//@ example x = 28 /
//@ }
```

The semantics of an example that does not use liberally is that, for all pairs of pre and post states, if the pair of states satisfies the conjunction of the predicate in the example, the precondition for the spec-case to which the example is attached, and its frame, then the pair should also satisfy the postcondition of that spec-case. That is, what should be checked to debug an example is the following, where Example, PreCondition, MP and TP are as above, and PostCondition is the postcondition of the spec-case.

\[(\text{Example} \land \text{PreCondition} \land \text{MP} \land \text{TP}) \Rightarrow \text{PostCondition}\]

In cases where there are applicable invariants (see Section 7.3 [Class Invariants], page 196) or history constraints (see Section 7.4 [History Constraints], page 200) these may be instantiated for whatever objects are assigned in the pre-state and conjoined to the hypothesis of the above formula. Thus, for example, in a class specification, one might prove the following, where Invariant(pre) is the invariant with the to the pre-state (pre) substituted for the state any used to express the invariant.

\[(\text{Example} \land \text{PreCondition} \land \text{Invariant(pre)} \land \text{MP} \land \text{TP}) \Rightarrow \text{PostCondition}\]
The meaning for an example that uses liberally is the same, except that termination is not implied. (See Section 6.12 [Liberal Specifications], page 151 for more on this topic and an example.)

Such a proof should be carried out for each example in the example-seq.

The reason for including the predicates describing the frame in the hypothesis, is to allow more succinct and less error-prone examples. (Without the frame axioms, most examples would have to state that each object was assigned in the post-state.)

6.9.2 Redundant Requires Clauses

The redundantly keyword can be used in a requires-clause to state what should be a redundant predicate that follows from the precondition. (see Chapter 6 [Function Specifications], page 82, for the syntax). Redundant requires clauses should follow any non-redundant requires-clause in a spec-case.

The main reason to use a redundant requires-clause is to highlight for the reader some property, an assumption, that is important, but which is implied by the precondition (and any invariants in force). As an example, consider the following specification of a function that decrements the integer value of the object pointed to by p. In this function, the assumption highlighted is that the argument p cannot be a null pointer.

```plaintext
//@ behavior {
//@ requires assigned(p, pre) /
//@ requires redundantly notNull(p);
//@ modifies *p;
//@ ensures returns /
//@   (*p)' = (*p)^ - 1;
//@ }
```

The semantics of a redundant requires-clause is that, if the precondition holds, then the assumption in the redundant requires-clause should also hold. That is, what has to be proved is the following, where PreCondition is the specified precondition, and Assumption is the assumption from the redundant requires-clause.

\[ \text{PreCondition} \Rightarrow \text{Assumption} \]

As before, if there are relevant invariants, these can be instantiated for the relevant objects and used in the proof.
6.9.3 Redundant Ensures Clauses or Claims

Another kind of redundancy in a function specification is a redundant ensures clause or claim [Tan94]. These have the same syntax as normal ensures-clauses, but use the keyword redundantly. (See Chapter 6 [Function Specifications], page 82 for the syntax.)

A redundant ensures-clause is the dual of an example; instead of stating some property that implies the specification, it states some property that follows from a function specification.

In Larch/C++, a sequence of redundant ensures-clauses can be placed at the end of a spec-case, following the required non-redundant ensures-clause. Each redundant post-cond in such a redundant ensures-clause should be implied by the rest of the spec-case.

Redundant ensures-clauses do not affect the meaning of a function specification. They are, however, useful in highlighting properties of a function specification, or in helping debug the specification.

Each redundant ensures-clause applies to the spec-case of which it is a part. By nesting multiple spec-cases in a pair of curly brackets (\{ \}), however, one can use redundant ensures-clauses to talk about the entire function specification.

A redundant ensures-clause that uses the keyword liberally asserts a property of executions of the function that terminate normally, but does not require normal termination. A redundant ensures-clause that does not use the keyword liberally asserts the existence of a post-state with the desired properties. Hence ensures redundantly true; asserts that every execution of the function in which the precondition is satisfied always terminates.

As a simple example, consider the following function specification. The claimed postcondition (hereafter, simply “the claim”) does not change the meaning of the specification from that given earlier (see Chapter 6 [Function Specifications], page 82) but it does highlight a property of the specification that is deducible using logic and facts about the integers.
The semantics of a redundant ensures-clause is that, if the precondition and postcondition are true, and the frame axioms given by the modifies-clause and trashes-clause are satisfied, then the claim (the redundant post-condition) should follow (see Section 5.2 of [Tan94]). That is, what has to be proved to verify a claim is the following [Tan94], where PreCondition is the (desugared) precondition of the function specification (which combines all the spec-cases), MP is the predicate that codes the modifies-clause (see Section 6.2.3 [The Modifies Clause], page 110), TP is the predicate that codes the trashes-clause (see Section 6.3.2 [The Trashes Clause], page 126), PostCondition is the (desugared) postcondition of the function specification, and Claim is the predicate from the redundant ensures-clause.

\[(\text{PreCondition} \wedge \text{MP} \wedge \text{TP} \wedge \text{PostCondition}) \Rightarrow \text{Claim}\]

In cases where there are applicable invariants (see Section 7.3 [Class Invariants], page 196) or history constraints (see Section 7.4 [History Constraints], page 200) these can also be instantiated for the relevant objects and used to help prove the above formula. Thus, for example, in a class member function specification, where there are no class instances other than self involved, it will suffice to prove the following, where Invariant\((pre)\) is the invariant with the to the pre-state \((pre)\) substituted for the state any used to express the invariant, Invariant\((post)\) is the invariant applied to the post-state, HistoryConstraint is the history constraint, and the other symbols are as above. (The history constraint is true if omitted.)

\[(\text{PreCondition} \wedge \text{Invariant}(pre) \wedge \text{MP} \wedge \text{TP} \wedge \text{PostCondition} \wedge \text{Invariant}(post) \wedge \text{HistoryConstraint}) \Rightarrow \text{Claim}\]

Such a proof should be carried out for each redundant ensures-clause in the ensures-clause-seq.
6.9.4 Redundancy in Frames

The keyword `redundantly` can also be used to state redundant frames. A redundant `modifies-clause` states a subset of the objects that are allowed to be modified by the non-redundant `modifies-clause(s)`. Similarly, a redundant `trashes-clause` states a subset of the objects that are allowed to be trashed by the non-redundant `trashes-clause(s)`. A good use of redundant modifies and trashes clauses is to point out objects that are allowed to be modified or trashed, but which are not mentioned explicitly by the non-redundant clauses. These are objects that the explicitly mentioned objects depend on (see Section 7.6 [The Depends Clause], page 213).

`[[[Needs example]]]

6.10 Case Analysis

Sometimes functions are best specified in several different cases. To allow this, a `spec-case-seq` in a function specification may have multiple `spec-cases` separated by the keyword `also`.

Recall that there are two forms of a `spec-case` (see Chapter 6 [Function Specifications], page 82). The simplest has the following syntax, and contains all the elements discussed so far in this chapter.

```
[ let-clause ] req-frame-ens [ example-seq ]
```

There is also a form of a `spec-case` that allows for internal nesting of a `spec-case-seq`. The syntax is as follows.

```
[ let-clause ] [ requires-clause-seq ]
{
    spec-case-seq
}
[ ensures-clause-seq ]
[ example-seq ]
```

This second form is useful if a `let-clause`, `requires-clause`, `ensures-clause` or `example-seq` should be applied to several `spec-cases`. 

See [Tan94] for more examples. (Larch/C++ currently only supports what Tan calls “procedure claims”.)
The meaning of a function specification with more than one spec-case in its spec-case-seq is that the function has to satisfy each spec-case in the list. The same idea gives the meaning of a nested spec-case-seq. Both kinds of specification can be regarded as syntactic sugar for a specification with only one spec-case, as will be described below.

As an example of the use of multiple cases, and then of the translation into a specification without multiple cases, consider the following.

```c
// @(#)$Id: widen1.lh,v 1.10 1997/06/03 20:30:26 leavens Exp $ extern void widen(int i) throw();//@ behavior {
//@ extern int low_bound, high_bound;
//@ // "normal" case
//@ requires assigned(low_bound, pre) /
//@ assigned(high_bound, pre)
//@ /
//@ INT_MIN + i < low_bound^ /
//@ low_bound^ < high_bound^ 
//@ /
//@ high_bound^ < INT_MAX - i;
//@ modifies low_bound, high_bound;
//@ ensures (high_bound' - low_bound') = (high_bound^ - low_bound^) + i
//@ /
//@ low_bound' <= low_bound^ /
//@ high_bound^ <= high_bound';
//@ also // other cases
//@ requires assigned(low_bound, pre) /
//@ assigned(high_bound, pre)
//@ /
//@ INT_MIN + i >= low_bound^ /
//@ low_bound^ < high_bound^ 
//@ /
//@ high_bound^ < INT_MAX - i;
//@ modifies high_bound;
//@ ensures high_bound' = high_bound^ + i;
//@ also
//@ requires assigned(low_bound, pre) /
//@ assigned(high_bound, pre)
//@ /
//@ INT_MIN + i < low_bound^ /
//@ low_bound^ < high_bound^ 
//@ /
//@ high_bound^ >= INT_MAX - i;
//@ modifies low_bound;
//@ ensures low_bound' = low_bound^ - i;
//@ }
```

In the “normal” case the function `widen` is expected to make the difference between `low_bound` and `high_bound` greater by the value of the formal parameter `i`. It can do this by either changing `low_bound` or `high_bound` or both. The other cases say that when `low_bound` is too small, `high_bound` must be increased, and vice versa.

An equivalent desugared version of the above specification is the following. This specification shows the general way to translate a spec-case-seq into a single spec-case. The general idea is
to: form the single precondition by disjoining the preconditions of the cases, list in the single modifies-clause everything listed in each of the modifies-clauses of the cases, and form the single postcondition from the conjunction of implications, with each implication being of one case’s precondition implying the postcondition of that case conjoined with an assertion saying that everything is unchanged which is not listed in the modifies-clause for that case.

```plaintext
//@ behavior { // a desugared version of widen1
//@ extern int low_bound, high_bound;
//@
//@ requires (assigned(low_bound, pre) \land assigned(high_bound, pre)
//@ \land \text{INT\_MIN} + i < low_bound' \land low_bound' < high_bound'
//@ \land high_bound' < \text{INT\_MAX} - i)
//@ \lor (assigned(low_bound, pre) \land assigned(high_bound, pre)
//@ \land \text{INT\_MIN} + i >= low_bound' \land low_bound' < high_bound'
//@ \land high_bound' < \text{INT\_MAX} - i)
//@ \lor (assigned(low_bound, pre) \land assigned(high_bound, pre)
//@ \land \text{INT\_MIN} + i < low_bound' \land low_bound' < high_bound'
//@ \land high_bound' >= \text{INT\_MAX} - i);
//@ modifies low_bound, high_bound;
//@ ensures ((assigned(low_bound, pre) \land assigned(high_bound, pre)
//@ \land \text{INT\_MIN} + i < low_bound' \land low_bound' < high_bound'
//@ \land high_bound' < \text{INT\_MAX} - i)
//@ \Rightarrow ((high_bound' - low_bound')
//@ = (high_bound' - low_bound') + i
//@ \land low_bound' <= low_bound'
//@ \land high_bound' <= high_bound'))
//@ \lor ((assigned(low_bound, pre) \land assigned(high_bound, pre)
//@ \land \text{INT\_MIN} + i >= low_bound' \land low_bound' < high_bound'
//@ \land high_bound' < \text{INT\_MAX} - i)
//@ \Rightarrow (high_bound' = high_bound' + i \land unchanged(low_bound)))
//@ \lor ((assigned(low_bound, pre) \land assigned(high_bound, pre)
//@ \land \text{INT\_MIN} + i < low_bound' \land low_bound' < high_bound'
//@ \land high_bound' >= \text{INT\_MAX} - i)
//@ \Rightarrow (low_bound' = low_bound' - i \land unchanged(high_bound)));
//@ }
```

If the spec-cases of a spec-case-seq have let-clauses, then these are first desugared before the above translation is applied.

The first specification of widen given above is less compact than it could be, because the first two conjuncts of each spec-case’s requires-clause are the same. Also, since the variables low_bound and high_bound occur so frequently in the specification, it is convenient to abbreviate them using a
**let-clause.** The following specification of `widen` uses these ideas to give a more compact specification of `widen`.

```c
// @(#)$Id: widen3.lh,v 1.6 1998/08/27 22:56:52 leavens Exp $ extern void widen(int i) throw(); //@ behavior {
//@ extern int low_bound, high_bound;
//@
//@ let lb: Obj[int] be low_bound, hb: Obj[int] be high_bound;
//@ requires assigned(lb, pre) \ assigned(hb, pre);
//@ {
//@ requires INT_MIN + i < lb^ \ lb^ < hb^ \ hb^ < INT_MAX - i;
//@ modifies lb, hb;
//@ ensures (hb' - lb') = (hb^ - lb^) + i \ lb' <= lb^ \ hb^ <= hb';
//@
//@ also // other cases
//@
//@ requires INT_MIN + i >= lb^ \ lb^ < hb^ \ hb^ < INT_MAX - i;
//@ modifies hb;
//@ ensures hb' = hb^ + i;
//@
//@ also
//@
//@ requires INT_MIN + i < lb^ \ lb^ < hb^ \ hb^ >= INT_MAX - i;
//@ modifies lb;
//@ ensures lb' = lb^ - i;
//@ }
//@ ensures redundantly inRange(lb') \ inRange(hb');
//@ example 8 < INT_MAX \ i = 2 \ lb^ = 3 \ hb^ = 6
//@ \ lb' = 1 \ hb' = 6;
//@ example 8 < INT_MAX \ i = 2 \ lb^ = 3 \ hb^ = 6
//@ \ lb' = 3 \ hb' = 8;
//@ example 8 < INT_MAX \ i = 2 \ lb^ = 3 \ hb^ = 6
//@ \ lb' = 2 \ hb' = 7;
//@ }
```

As illustrated in the above example, the potential scope of a definition made in a **let-clause** is its entire **spec-case**, this includes nested **spec-cases**.

Also illustrated by the above example is that a **requires-clause** applies to nested **spec-cases** by conjunction; that is, such a **requires-clause** is conjoined to the **requires-clause** of each nested **spec-case**.

If there are multiple, non-redundant **requires-clauses** or **ensures-clauses** in a **requires-clause-seq**, or **ensures-clause-seq** in a **spec-case**, these are first conjoined to make each **spec-case** have a single non-redundant such clause.
The redundant ensures clause and the examples given in the last specification of `widen` above apply to the entire specification case. That is, they apply to the meaning of a `spec-case` with a nested `spec-case-seq`.

Such a specification can be translated into a specification with just one `spec-case` by first desugaring the nested `spec-case-seq`, then conjoining the `requires-clause` to the desugared `requires-clause`. If desired, one could also then desugar the `let-clause`.

For example, the last specification of `widen` above is equivalent to the following,

```c
//@ behavior {
//@ extern int low_bound, high_bound;
//@ requires (assigned(low_bound, pre) \ assigned(high_bound, pre))
//@ \ ( (INT_MIN + i < low_bound^ \ low_bound^ < high_bound^ 
//@ \ high_bound^ < INT_MAX - i) 
//@ \ (INT_MIN + i >= low_bound^ \ low_bound^ < high_bound^ 
//@ \ high_bound^ < INT_MAX - i) 
//@ \ (INT_MIN + i < low_bound^ \ low_bound^ < high_bound^ 
//@ \ high_bound^ >= INT_MAX - i) 
//@ \ modifies low_bound, high_bound;
//@ ensures ((INT_MIN + i < low_bound^ \ low_bound^ < high_bound^ 
//@ \ high_bound^ < INT_MAX - i) 
//@ => ((high_bound' - low_bound') 
//@ = (high_bound^ - low_bound^) + i 
//@ \ low_bound' <= low_bound^ \ high_bound^ <= high_bound'))
//@ \ ((INT_MIN + i >= low_bound^ \ low_bound^ < high_bound^ 
//@ \ high_bound^ < INT_MAX - i) 
//@ => (high_bound' = high_bound^ + i /\ unchanged(low_bound)))
//@ \ ((INT_MIN + i < low_bound^ \ low_bound^ < high_bound^ 
//@ \ high_bound^ >= INT_MAX - i) 
//@ => (low_bound' = low_bound^ - i /\ unchanged(high_bound')));
//@ ensures redundantly inRange(low_bound') /\ inRange(high_bound');
//@ example 8 < INT_MAX /\ i = 2 /\ low_bound^ = 3 /\ high_bound^ = 6
//@ \ low_bound' = 1 /\ high_bound' = 6; 
//@ example 8 < INT_MAX /\ i = 2 /\ low_bound^ = 3 /\ high_bound^ = 6
//@ \ low_bound' = 3 /\ high_bound' = 8;
//@ example 8 < INT_MAX /\ i = 2 /\ low_bound^ = 3 /\ high_bound^ = 6
//@ \ low_bound' = 2 /\ high_bound' = 7;
//@ }
```
Redundant clauses need not appear in the translation of a case analysis, as they are not needed to give meaning to the specification. Still, it may be informative (or fun) to include them. Examples are easily taken care of by combining all the examples into one large example-seq in the translation. Redundant ensures-clauses are all listed after the non-redundant ensures-clause, but with their post-conds changed into implications. The hypothesis of such an implication is the precondition of its original spec-case, and the antecedent is the original predicate of the claims-clause.

Redundant requires-clauses are more tricky. They can be translated by conjoining all the pre-conds for each spec-case together, and then disjoining the resulting conjunctions to form a single pre-cond for a single redundant requires-clause.

See Section 6.11 [Exceptions], page 148 for more examples of case analysis.

This feature of Larch/C++ is inspired by various versions of refinement for function specifications. The simple case of a spec-case-seq without nesting is nearly identical to the mechanism found in Section 4.1.4 of [Wing83], and is similar to the capsule mechanism of [Wills92b].

### 6.11 Exceptions

As in C++, the interface of a function can declare what exceptions the function can throw (see Section r.15 in [Stroustrup91]). Note that the different exceptions that can be thrown are distinguished by their types. A common practice would be to define a class for each kind of exception; this class's data members could contain information pertinent to the exception. The following syntax is from C++.

\[
\text{exception-decl} ::= \text{throw ( [ type-list ] )}
\]

\[
\text{type-list} ::= \text{type-id [ , type-id ]} \ldots
\]

At run-time, a C++ function can either return or throw an exception, but not both. These two cases can be distinguished by using a lcpp-primary of the form \text{returns} or \text{throws} (\text{type-id}). These can only be used in a postcondition of an ensures-clause, or in an example.

In Larch/C++, both returning and throwing an exception is considered to be termination. Non-termination, of course, means going into an infinite loop. Nontermination also occurs when a function does not return to its caller, but jumps or exits or aborts in some fashion. See Section 6.12.1 [Terminates], page 152 for more discussion on this point. Because both returning and throwing an exception is considered to be termination, they are both dealt with in the ensures-clause of a function specification.
The `lcpp-primary returns` is true when the function terminates and does not throw an exception. If the function terminates and `{returns}` is true, then some exception is raised.

A `lcpp-primary` of the form `throws(T)` is true when the function terminates and throws the exception `T`.

Formally, Larch/C++ models a function may throw an exception as a relation between pre-states pairs of tagged results and post-states. One can test the tags of such a tagged result using the primaries `result` and `throws(T)`; one can extract the associated values with the primaries `result` and `thrown(T)`. The primary `result` has a well-defined value when `returns` is true, and `thrown(T)` has a well-defined value when `throws(T)` is true.

See Section 6.1.10 [Larch/C++ Special Primaries], page 99 for the syntax of a `lcpp-primary` beginning with `thrown`. The sort of an `lcpp-primary` of the form `thrown(T)` is the sort of a formal parameter of type `T` (see Section 6.1.8.1 [Sorts for Formal Parameters], page 94). The sort an `lcpp-primary` of the form `returns` or `throws(T)` is `bool`.

The following is a simple example of a function specification with an exception specified. See Section 6.10 [Case Analysis], page 143 for the meaning of having a global precondition and two subsidiary `spec-cases`.

```c
//@ behavior {
//@ requires assigned(i, pre);
//@ { // @ requires i^-2 <= INT_MAX;
//@ modifies i;
//@ ensures returns i' = i^-2;
//@ ensures redundantly result = theVoid;
//@ also
//@ requires i^-2 > INT_MAX;
//@ ensures throws(Overflow*);
//@ ensures redundantly (*thrown(Overflow))' = theException;
//@ }
//@ }
```
In a specification of a function that can raise exceptions, it is important to use the \texttt{returns} and \texttt{throws} keywords to distinguish cases of the result. This is because the logic behind LSL will assign a value to \texttt{result}, even if \texttt{returns} is false! The above specification is a rather extreme example of this, as the set of abstract values for the type \texttt{void} contains only one element, \texttt{theVoid} (see Section 11.5 [void], page 278). Hence, writing \texttt{result = theVoid} in the postcondition is not equivalent to writing \texttt{returns there}; instead \texttt{result = theVoid} is equivalent to \texttt{true} [Jones95e]! Similarly, one cannot use \texttt{thrown(Overflow)} instead of \texttt{throws(Overflow)}.

The above specification includes the following class specification, which is typical of how you would specify a class for an exception that contains no information. (See Chapter 7 [Class Specifications], page 167 for the syntax and more details about class specifications.)

```lsl
// @(#)$Id: Overflow.lh,v 1.7 1997/01/27 21:25:20 leavens Exp $
//@ uses NoInformationException(Overflow), NoContainedObjects(Overflow);
class Overflow { }
```

The trait used by the class \texttt{Overflow} to describe its abstract values is built-in to Larch/C++ for specifying exceptions with no information. The abstract value of such an exception is thus named \texttt{theException}. It is the following trait. (The included trait is given below.)

```lsl
% @(#) $Id: NoInformationException.lsl,v 1.2 1995/11/13 18:17:58 leavens Exp $
NoInformationException(T) : trait
   includes NoInformation(T, theException for it)
```

The above trait in turn includes the following trait, which specifies a one-point set.

```lsl
% @(#) $Id: NoInformation.lsl,v 1.2 1995/11/13 18:18:58 leavens Exp $
NoInformation(T) : trait
   includes NoContainedObjects(T)
   introduces
      it: -> T
   asserts T generated by it
   implies
      \forall x, y: T
      x == it;
      x == y;
```
One can also specify a function that can either throw an exception or do something else, without saying which. The following is an example. Note that this specification can be implemented by a function that always throws an exception.

```c
#include "Overflow.lh"

extern void inc3(int& i) throw(Overflow*);
//@ behavior {
//@ requires assigned(i, pre);
//@ modifies i;
//@ ensures (returns \( i' = i^" + 3 \))
//@ \( \text{\slash throws(Overflow*) \slash unchanged(i)} \);
//@ example i^" = 4 \( \text{\slash i'} = 7 \) \text{\slash returns};
//@ example i^" = 4 \( \text{\slash i'} = 4 \) \text{\slash throws(Overflow*)};
//@ }
```

6.12 Liberal Specifications

Until this point, we have ignored the possibility of one writing an `ensures-clause` (see Chapter 6 [Function Specifications], page 82) of the following form.

```
ensures liberally post-cond ;
```

Such an `ensures-clause` gives a partial-correctness specification, as opposed to the usual total-correctness specification [Dijkstra76]. In a partial-correctness specification, if the precondition holds and if the function terminates (see Section 6.12.1 [Terminates], page 152), then the `post-cond` must hold; however, normal termination is not guaranteed, even when the precondition holds.

In Larch/C++, a `spec-case` that uses the keyword `liberally` will be called a partial-correctness `spec-case`. A total-correctness `spec-case` is one that does not use the keyword `liberally`. It guarantees normal termination for all pre-states for which its precondition holds.

As an example of a liberal or partial-correctness specification, consider the following specification of a factorial function. The function specified need not terminate normally when called with any arguments (as noted in the third example), but if it does terminate, then the result must be the factorial of the argument.
The predicate false can be used in a liberal example to say that no post-state exists. This is used in the third example of the fact_liberal function specified above.

A valid implementation of the specification of fact_liberal would be an infinite loop; another implementation would loop only on negative arguments. An implementation that halts the program (on some arguments) would also be acceptable.

The above specification uses the trait FactorialTrait, which itself uses the trait int (see Section 11.1.5 [int Trait], page 272). Note that factorials of negative numbers are defined to be 1.

6.12.1 Terminates

A function call terminates if it returns to its caller or throws an exception. A function call does not terminate if it goes into an infinite loop, halts execution of the program (gracefully or not), or jumps in such a way that it does not return or throw an exception that could be caught by its caller.
Taking termination into account, we can model a C++ function by a nontotal relation from proper pre-states to post-states. A state is proper if it is not bottom (see Section 2.8.2 [Formal Model of States], page 30); bottom represents infinite looping and other kinds of abnormal termination.

A pre-state may or may not be related to a post-state by the relation specified in a Larch/C++ specification, and if it is related to a post-state, that post-state could be bottom. (A pre-state that is not related to a post-state value is one for which execution is “refused” [Nelson89] [Hesselink92]. Such relations cannot be implemented in C++, but are useful for purposes of program refinement.)

### 6.12.2 Liberal Specification and Case Analysis

Liberal specification can be combined with case analysis. For example, suppose one wanted to specify that \texttt{fact\_liberal2} must terminate normally when its argument is sensible. This could be specified as follows.

```plaintext
// @(#)$Id: fact\_liberal2\_lh,v 1.6 1997/06/03 20:30:04 leavens Exp $
extern int fact\_liberal2(int n) throw();
//@ behavior {
//@ uses FactorialTrait;
//@
//@ requires 0 \leq n /\ factorial(n) \leq INT\_MAX;
//@ ensures result = factorial(n);
//@ also
//@ requires ~(0 \leq n /\ factorial(n) \leq INT\_MAX);
//@ ensures liberally result = factorial(n);
//@ }
```

When discussing case analysis above, we presented a way to think of case analysis as syntactic sugar for a specification with a single spec-case (see Section 6.10 [Case Analysis], page 143). However, because partial-correctness and total-correctness specifications have different meanings, one cannot desugar a spec-case-seq with a mix of such spec-cases into a single spec-case. One can only desugar the specification into a spec-case-seq with two spec-cases: one for the total-correctness specification, and one for the partial-correctness specification. This is done by applying the syntactic sugaring given above for combining spec-cases separately to each kind of spec-case: total and partial. Hence, the above specification cannot be further desugared.

As an aside, by using standard techniques [Dijkstra76] [Nelson89] [Hesselink92], one can always rewrite such a specification so that it has one spec-case with an ensures-clause of the form \texttt{ensures}
Chapter 6: Function Specifications

154

true and a second spec-case with an ensures-clause of the form ensures liberally P, for some
post-cond P. For example, the following is equivalent to the above specification of fact_liberal2.

// @(#)$Id: fact_liberal3.lh,v 1.5 1997/06/03 20:30:05 leavens Exp $
extern int fact_liberal2(int n) throw();
//@ behavior {
//@
uses FactorialTrait;
//@
//@
requires 0 <= n /\ factorial(n) < INT_MAX;
//@
ensures true;
//@ also
//@
requires true;
//@
ensures liberally result = factorial(n);
//@ }

The meaning of a specification with multiple spec-cases is as always — the function must satisfy
all of them. In the above specification, this means that the function must terminate normally when
the argument satisfies the requires-clause of the first spec-case, and that when it terminates, it
must satisfy the ensures-clause of the second spec-case.

6.12.3 Examples of Liberal Specification
By using both total and partial-correctness spec-cases, one can specify interesting behaviors
that would not be possible with just total-correctness specification. This can be done because one
can precisely specify both when the function terminates and what must be true of the states in
which it is allowed to terminate.

As an example of how to specify a function that does not return to its caller, consider the
following specification of the C++ library function abort (see Section 3.4 of [Ellis-Stroustrup90]).


extern void abort() throw();
//@ behavior {
//@ requires false;
//@ ensures true;
//@ also
//@ requires true;
//@ ensures liberally false;
//@ }

The first spec-case in the above specification says: there is no way to call abort so that it is guaranteed to terminate normally. The second spec-case says that furthermore: every execution of abort either fails to terminate normally or it terminates in a state in which false holds; that means, of course, that it can never terminate normally, because there is no state in which false holds. However there is a potential execution, because the state bottom does not represent normal termination. Hence, as a relation, abort relates all states to bottom.

One can also specify functions that could not possibly be implemented, but which may be useful in program refinement. For example, the following is the specification of the function miracle (see [Nelson89] and Section 1.3 of [Hesselink92]).

extern void miracle();
//@ behavior {
//@ extern everything;
//@
//@ requires true;
//@ modifies everything;
//@ trashes everything;
//@ ensures true;
//@ also
//@ requires true;
//@ modifies everything;
//@ trashes everything;
//@ ensures liberally false;
//@ }

According to the total-correctness case of this specification, every execution of a call to miracle would have to terminate normally. According to the partial-correctness case, every such execution
would have to terminate in a state in which false holds. Since there are no such executions, such a function would have to refuse to execute, and thus could not be implemented in C++.

### 6.12.4 Meaning of Function Specifications

Every function specification can be desugared into one with only one total-correctness `spec-case` and one partial-correctness `spec-case`. A function with only one total correctness `spec-case` can be rewritten into this form in the standard way [Dijkstra76] [Nelson89] [Hesselink92] (see Section 6.12.2 [Liberal Specification and Case Analysis], page 153 for an example). A specification with one or more partial correctness `spec-cases` can be rewritten into one with a single partial correctness `spec-case` by using the desugaring for case analysis (see Section 6.10 [Case Analysis], page 143); then a total correctness `spec-case` of the following form can be added, since this adds no information to the specification.

```plaintext
requires false;
modifies everything;
trashes everything;
ensures true;
```

(The frame allows everything to be modified or trashed, but that would ordinarily be restricted by the other `spec-case`.)

So to give the meaning of a function specification, it suffices to consider a function specification with one total-correctness `spec-case` and one partial-correctness `spec-case` (one that uses liberally). A C++ function satisfies its specification if and only if for each type-correct function call: (i) if the precondition of the total-correctness case is satisfied in the pre-state, then the function call terminates, the function mutates at most the set of objects described in the total-correctness `modifies-clause`, trashes at most the set of objects described in the total-correctness `trashes-clause`, and the total-correctness postcondition is satisfied by the pre-state and the post-state, and furthermore (ii) if the precondition of the partial-correctness case is satisfied in the pre-state, and if the function call terminates, then function mutates at most the set of objects described in the partial-correctness `modifies-clause`, trashes at most the set of objects described in the partial-correctness `trashes-clause`, and the partial-correctness postcondition is satisfied by the pre-state and the post-state.

For specifications that do not use liberally, the above definition of satisfaction is the same as that as described previously (see Section 6.2.3 [The Modifies Clause], page 110).
6.13 Specifying Higher-Order Functions

A higher-order function in C++ is a function that either takes pointers to functions as arguments, or returns them. A simple example is a sort function that takes a pointer to a comparison function as one of its arguments.

There Larch/C++ syntax that is used especially to specify higher-order functions is given below. See Section 9.2 [The Uses Clause], page 255 for the syntax of trait-list.

\[
\begin{align*}
\text{higher-order-comparison} & ::= \text{lsl-op-term} \, \text{satisfies} \, \text{fun-interface} \, \text{fun-spec-body} \\
\text{expects-clause} & ::= \text{expects} \, \text{expected-trait-list} \\
\text{expected-trait-list} & ::= \text{trait} \, [ \, \text{simple-id} \, ] \, [ \, , \, \text{trait} \, [ \, \text{simple-id} \, ] \, ] \, \ldots \\
\text{using-trait-list} & ::= \text{using} \, \text{trait-or-deref-list} \\
\text{trait-or-deref-list} & ::= \text{trait} \\
& \mid \ast \, \text{simple-id} \\
& \mid \text{trait-or-deref-list} \, , \, \text{trait} \\
& \mid \text{trait-or-deref-list} \, , \ast \, \text{simple-id}
\end{align*}
\]

A higher-order-comparison is an equality-term (see Section 6.1.3 [Equality Terms and Quantifiers], page 87). It is used to check whether a C++ function, the lsl-op-term in the higher-order-comparison, satisfies a function specification. A higher-order-comparison has sort Bool.

An expects-clause is used to specify a theory that is needed to state the rest of the specification formal parameter must satisfy. This theory is typically used in the function-spec-body of a higher-order-comparison. To permit verification, code that calls a higher-order function must pass an actual parameter trait that contains the theory of each expected trait in the expects-clause. This is done using the form of a postfix-expression that Larch/C++ adds to C++ (see Section 5.4.6 [Function Declarations], page 72). The syntax is that, following the normal arguments, one can use the keyword using and then list either LSL traits or trait dereferences. These trait dereferences have the form \( \ast \, \text{simple-id} \), where the simple-id is the name of a trait formal parameter. The named traits, and the values of the corresponding actual parameters for the dereferenced formals are passed, positionally. This is similar to the specification of generic functions and classes used in OBJ2 [Goguen84] and Resolve [Ernst-etal91] [Edwards-etal94]. See Section 9.2 [The Uses Clause], page 255 for the exact syntax of trait.

To start with a standard kind of example, consider the function ArrayMap. This function takes an array, the array’s length, and a pointer to a function, and modifies the array by applying the function to each element of the array, a[i], and placing the result of the call back in a[i]. It might be implemented as follows.
To specify the `ArrayMap` function, one has to make some decisions, such as the following. Is this supposed to work when `a` is null or uninitialized? Is it supposed to work when the pointer `fun` is null or uninitialized? Does `*fun` have to work for all integers? (If it does, then we are limiting the applicability of `ArrayMap`.) Do we care what happens if some `a[i]` does not satisfy the precondition of `*fun`? Do we care about overflow in the computation of a call to `*fun`? Can `*fun` have side-effects?

In the following specification, the first conjunct in the precondition requires `a` to be non-null and the relevant part to be initialized, the second conjunct requires `fun` to be non-null and initialized, and the fourth conjunct requires that each of the relevant elements of `a` satisfy the precondition of `*fun`. The third conjunct of `ArrayMap`'s precondition is a `higher-order-comparison`. It is true just when `(*fun)` satisfies the `function-spec-body` in that conjunct. This is true if whenever `x` satisfies `inDomain(x)`, then calling the function with argument `x` is guaranteed to terminate normally with a result that satisfies `isResultFor(x, result)` and is a legal `int` (due to `inRange(result)`).
typedef int (*int_fun)(int) throw();

typedef int func(int) throw();

extern void ArrayMap(int a[], int len, int_fun fun
    /*@ expects NoSideEffectsFun(int, int) @*/ ) throw();
//@ behavior {
//@ requires assignedUpTo(a, len, pre) \ assigned(fun, pre)
//@    \ (fun)^ satisfies
//@    int ignored(int x) throw()
//@    behavior {
//@        requires inDomain(x);
//@        ensures isResultFor(x, result)
//@        \ inRange(result);
//@    }
//@    \ \forall i: int (between(0, i, len-1) => inDomain(a^[i]));
//@    modifies objectsUpTo(a, len);
//@    ensures \forall i: int (between(0, i, len-1)
//@        => isResultFor(a^[i], a'[i]));
//@ } 

The above example expects a trait parameter that implies the theory of NoSideEffectsFun(int, int), which is shown below. This trait specifies minimum requirements on the theory of the trait functions inDomain and isResultFor that are used in the function-spec-body that (*fun)^ must satisfy. Note, however, that the specification of the trait function isResultFor is not complete; this is typical for expected traits. A call to ArrayMap would provide an actual trait in a using-trait-list with a theory that contained the theory of this trait. For example, one might write the following to call ArrayMap

    ArrayMap(myArray, myArrayLen, myFun /*@ using myFunTrait @*/)

where myFunTrait is an actual parameter trait that specifies the trait functions inDomain and isResultFor required by NoSideEffectsFun(int, int). This actual parameter trait would typically provide a more precise specification of isResultFor.

[[[Should have a real example for the above.]]]
As another example, consider a higher-order function \texttt{ApplyTwice}, which takes a pointer to a function and an integer and returns the result of applying the function twice, as in the following code.

\begin{verbatim}
// this is C++ code, not a specification
typedef int (*int_fun)(int);
int ApplyTwice(int_fun f, int i) {
    return f(f(i));
}
\end{verbatim}

In the following specification, we have decided that the client can only count on the result provided the following conditions are met (as specified in the precondition). The pointer \( f \) must be non-null and initialized. The argument \( i \) must meet the precondition of \(*f\), and must result in a value that again meets the precondition of \( f \), and that \( f \) must be deterministic. Note that much of this information is not available in the code above, which only specifies how the computation is implemented, not what contract it fulfills.
typedef int (*int_fun)(int) throw();

extern int ApplyTwice(int_fun f, int i
    /*@ expects NoSideEffectsDetFun(int, int) @*/) throw();
//@ behavior {
//@ requires assigned(f, pre)
//@ /
//@ 
//@ int ignored(int j) throw()
//@ behavior {
//@ requires inDomain(j);
//@ ensures result = resultFor(j) /
//@ inRange(result);
//@ }
//@ /
//@ requires redundantly inRange(result);
//@ example \forall j: int (inDomain(j)
//@ => (between(0, j, 30) /
//@ resultFor(j) = j+1))
//@ /
//@ \forall i = 5 /
//@ result = 7;
//@ }

In the above specification, the example given supposes some additional properties of the trait functions inDomain and resultFor. Of course, the actual trait parameter might have different properties, but this indicates how knowing the actual parameter trait allows one to extract additional information from the postcondition.

The trait expected for the above example is built on the trait NoSideEffectsFun given above.
A common kind of higher-order function is a kind of iterator. Consider the following code for a C++ function, `ArrayForEach`. It applies `f` to each element of an array. In `ArrayForEach`, `f` is expected to have side-effects.

```cpp
// this is C++ code, not a specification
typedef void (*elem_fun)(int) throw();
void ArrayForEach(int a[], int len, elem_fun f) {
    for (int i = 0; i < len; i++) {
        (*f)(a[i]);
    }
}
```

In the following specification, of `ArrayForEach`, we try to allow as much freedom as possible to `*f`. That is, we allow `*f` to access global variables and to modify them (this allows one to use `ArrayForEach` to do output, or to sum the elements into a global), and even to trash them (this allows one to use `ArrayForEach` to decide what other memory to deallocate). To support arbitrary preconditions on the globals, and arbitrary postconditions, we pass the pre-state to the trait function `inDomain`, and we pass the pre- and post-states to `isStateFor`. However, we are careful not to allow `*f` to deallocate the part of `a` being used, and to not change its own location! In order to make it possible for `*f` not to change itself, we require that `*f` is not aliased to any of the `a[i]`. The postcondition says that the post-state is a possible state obtained from iterating the state change of `*f`, and also includes some other conclusions that follow from the inability of `*f` to change its own location, and to deallocate the part of `a` being used.

```plaintext

typedef void (*elem_fun)(int) throw();
 extern void ArrayForEach(int a[], int len, elem_fun f 
    /*! expects SideEffectsFun(int) */) throw();
//@ behavior {
//@   uses ArrayIterateProc(Obj, int);
//@   requires assignedUpTo(a, len, pre) \ assigned(f, pre)
//@   \ (*f)^" satisfies
//@   \ int ignored(int x) throw()
//@   behavior {
//@     extern everything;
//@     requires inDomain(x,pre);
//@     modifies everything;
//@     trashes everything;
//@     ensures isStateFor(x, pre, post)
//@     \ assignedUpTo(a, len, post)
//@     \ \forall i: int
```
The expected trait in the above specification is the following.

The specification of ArrayForEach also uses the following trait, which is not provided as a parameter.
% Iteration of state changes for elements of an array
ArrayIterateProc(Loc, Elem): trait
  assumes SideEffectsFun(Elem)
  includes Array(Loc, Elem)
  introduces
    isIteratedStateFor: Arr[Loc[Elem]], int, int, State, State -> Bool
  asserts \forall a: Arr[Loc[Elem]], i,ub: int, pre,next,post: State
    (i >= ub)
    => isIteratedStateFor(a, i, ub, pre, pre);
    ((i < ub)
      /\ isStateFor(eval(a, pre)[i], pre, next)
      /\ isIteratedStateFor(a, i+1, ub, next, post))
    => isIteratedStateFor(a, i, ub, pre, post);

[[[More needs to be said about the formal semantics of the higher-order constructs. Should also give an example of a higher-order-comparison used in a postcondition.]]]

6.14 Behavior Programs

As in the refinement calculus [Back88] [Back-vonWright89a] [Morgan94] [Morris87], besides stating a specification using pre- and postconditions, one can also state a specification as an abstract program. The syntax for this in Larch/C++ is to use the keywords behavior program followed by a compound statement. Typically, such a compound statement would use specification-only variables (see Section 9.1 [Ghost Variables], page 254) and specification statements, so as to keep the specification as abstract as possible.

Specification statements are added to the usual C++ statements for this purpose. They include assert statements and various clauses from the grammar for fun-spec-body, as well as a fun-spec-body itself.
The meaning of a *fun-spec-body* as a statement is that, the code that replaces the *fun-spec-body* must satisfy the given specification. The meaning of a *requires-clause* as a statement is that the precondition given must be true at that point in the program. This is essentially the same meaning as the *assert* statement, except that in an *assert* one can refer to both the pre-state and current state’s values of variables. The current state’s value of a variable is referred to using the notation for post-state values (‘’). Similarly, the meaning of an *ensures-clause* is that the given postcondition must describe a true relationship between the pre-state values of variables and the current state’s values (again written using the usual post-state notation).

Typically a *requires-clause* would only be used as a stand-alone statement at the beginning of a *compound-statement*, and an *ensures-clause* would only be used at the end of a *compound-statement*. Intermediate assertions would be written using *assert*.

The meaning of the *accesses-clause*, *modifies-clause*, *trashes-clause*, *calls-clause* is that the given constraint must be satisfied in the current state (with respect to the pre-state). For example, a *modifies-clause* says that since the beginning of the function’s execution, only the given objects have been modified. Note that omitting one of these clauses in a behavior program is akin to not having any restriction at all; this is different than the usual notion of omitting a *modifies-clause* or *trashes-clause*.

As a very simple example of a behavior program, consider the specification of *inc4* below. This specification says that a correct implementation of *inc4* must behave as a refinement of the abstract program that is given. Specifically, it may assume that the reference parameter *i* is assigned in the pre-state, and then it can increment the value of *i* by 4, provided that this will not cause an overflow, otherwise it must throw *Overflow*. 

The reader may wish to compare the above specification of `inc4` with that of `inc2` above (see Section 6.11 [Exceptions], page 148).

One good use for a behavior program is to specify higher-order functions. For example, consider the `ArrayMap` example above (see Section 6.13 [Specifying Higher-Order Functions], page 157). One may specify this using a behavior program, yielding a specification that is significantly shorter than that given using a higher-order comparison.

The possible disadvantages of using a behavior program include that it may be more difficult to reason about, and a potential for implementation bias [Jones86b].
7 Class Specifications

A main feature of Larch/C++, the feature that supports object-oriented programming, is the ability to specify C++ classes. The syntax is as follows. The syntax of a class-specifier is much the same as C++. In addition, the member-seq (see Section 7.2 [Class Member Specifications], page 186) besides containing declarations and specifications of data and function members, may contain one or more of the following: uses-clauses (see Section 9.2 [The Uses Clause], page 255), invariant-clauses (see Section 7.3 [Class Invariants], page 196), and constraint-clauses (see Section 7.4 [History Constraints], page 200). The member-seq may also contain various features to support the specification of behavioral subtypes and specification inheritance (see Section 7.9 [Inheritance of Specifications and Subtyping], page 215).

\[
\text{class-specifier ::= class-head \{ [ member-seq ] \)}
\]

\[
\text{class-head ::= class-key [ identifier ] [ base-spec ]}
\]

\[
\text{\quad | class-key nested-name-specifier identifier [ base-spec ]}
\]

\[
\text{\quad | class-key [ nested-name-specifier ] template-class-instance [ base-spec ]}
\]

\[
\text{\quad | abstract class [ nested-name-specifier ] [ identifier ] [ base-spec ]}
\]

A class-head is part of the C++ interface of the specified class.

The syntax of a class-specifier is also used in Larch/C++ for the declaration of signature types; these are used to help specify the requirements on template parameters (see Chapter 8 [Template Specifications], page 239). It is also used for defining struct and union types (see Section 5.2.3 [Type Specifiers], page 59). Such types are declared by using a class-key of struct or union.

In Larch/C++, structure, class, union, and signature types are treated in the same manner. (Of course, as in C++, the visibility of a structure starts out as public, whereas for a class it starts as private.) For such types, Larch/C++ can automatically construct a model of their abstract values for you, by putting together the declared non-static class members (which would usually be specification-only declarations for classes). This is the usual and common way to specify such types. However, you may also assume complete responsibility for providing the abstract model by using a trait that specifies a sort of the same name as the type you are defining (and the trait function contained_objects). (See Section 9.2 [The Uses Clause], page 255 for how to use a trait. Examples are also given later in this chapter.) If you use such a trait, that overrides the automatically-supplied abstract values. See Section 11.10 [Structure and Class Types], page 299 for details on the automatically-supplied traits for struct types. See Section 11.11 [Union Types], page 302 for details on the automatically-supplied traits for union types.
In the rest of this chapter, we will use the word “class” instead of constantly saying “class or structure or union.” However, aside from the difference in default visibility noted earlier, everything said about classes applies equally to structures and unions.

Returning to the specification of classes, in the member-seq, the uses-clause allows one to describe traits that are used within a class specification. As in our examples, however, when one uses a trait to aid a class specification, most often the uses-clause would appear outside the class, so that clients will also have that trait available when they include the class’s specification. See Section 9.2 [The Uses Clause], page 255 for details.

A C++ class satisfies a class specification if it has the same interface (see Section 2.2 [Interfaces], page 15), if it has the declared data members, and if each member function satisfies its specification.

The main focus of the sections below is on how to use class specifications, and how the various features interact with each other.

7.1 Examples of Class Specifications

In this section, we show how to specify some simple classes in Larch/C++.

7.1.1 A First Class Design (Person)

The first major hurdle you face as a user of Larch/C++ is how to specify the abstract values of a class.

Consider the specification of a class Person. The first question to ask is: what kind of information should be kept about a Person? To keep things simple, suppose you want to keep track of a person’s name and age.

As in programming, the usual thing is to introduce some data members to record this decision. However, since we don’t want to require that these particular data members appear in every implementation, we make them “specification” or “ghost” variables. That is we prefix their declarations with the Larch/C++ keyword spec, and put them in annotation comments. This means that these declarations need not appear in an implementation. (See Section 9.1 [Ghost Variables], page 254 for more details on spec-decls.) For example, we would write the following as the declaration of a data member that tracks a person’s age.
// spec int age;

To track a person’s name, we want some kind of string. If we had a C++ template class defined already for strings, such as String<char>, then we could declare the name as follows.

// spec String<char> age;

However, suppose we don’t already have such a class defined, what can we do? Well, one could go off and specify such a class and all its details in Larch/C++, but that would be overkill, since we don’t need to use the class itself, as name is just a specification variable, not something we are (necessarily) manipulating in a C++ program. To avoid all that effort, we just use another spec-decl to declare the template class String, as in the following file.

// @(#)$Id: AbstractString.lh,v 1.1 1997/07/28 21:02:51 leavens Exp $  
//@ spec template<class C> class String;  
//@ uses AbstractStringTrait; // defines the sort String[char]  
//@ uses NoContainedObjects(String<char>);

The used trait AbstractStringTrait (see Section 4.13.5 [Abstract String Constants], page 51) associates a sort with the name String<char>. (Larch/C++ uses this trait by default in any case, but this makes it clearer.) Since we are associating the abstract values explicitly with the type String<char>, it is necessary to state whether the abstract values have any contained objects, hence the use of the trait NoContainedObjects(String<char>) (see Section 7.5 [Contained Objects], page 207).

Instead of using such abstract strings, you might think to use a C++ pointer to an array of null terminated characters. But this would be a mistake for such a simple specification, as it would make the abstract values much more complicated than they need to be. All that is needed are the characters in the name, details such as that they are contained in an array etc. are irrelevant. See Section 5.4.4 [Structure and Class Declarations], page 69 for the comparable, but more complicated, abstract values of the type Entry.)

Note that the abstract values of C++ integers are specified in the Larch/C++ trait int (see Section 11.1.5 [int Trait], page 272 for details). The association between the type int and the abstract values in that trait is made, again, by having a sort with the same name int, defined in the trait int.
At this point it is a good idea to think about invariant properties you want to hold for the values in the specification variables. For example, you might want ages to always be positive. There are two ways to handle this. One is to change the type of the specification variable `age` to some type (like `unsigned int`) that has only positive integer values. However, if you want to use the type `int` in the interface to communicate with clients, that may cause difficulty. The second way is to use a Larch/C++ invariant-clause when you get to the point of writing the interface specification. An invariant allows you to state that you will only be using a subset of the potential abstract values.

Once you have decided how to model the information in instances of the type `Person`, you can proceed to designing and specifying the C++ interface.

For the design, typically you will want some or all of the following kinds of member functions.

- **Constructors**, which have the same name as the name of the class. For the class `Person`, we will use one constructor named `Person`. You may have one or more constructors with different types of arguments. The job of a constructor is to initialize an object once C++ has allocated space for it.

- **A destructor**, which has the name of the class prefixed by `~`. For the class `Person`, the destructor would be named `~Person`. The job of a destructor is normally to deallocate any dynamically allocated contained objects. Since there are no contained objects in a `Person` instance, the destructor should do nothing. (This is fine, because it is the C++ operator `delete` that actually deallocates storage for a `Person` instance, not the destructor.)

- **Mutators**, which are member functions that change the abstract value of specification variables (or the default argument, `*this`). For the class `Person`, we might want something to change a person’s name, and to add a year to their age (on their birthday). We call these `Change_Name` and `Make_Year_Older`.

- **Observers**, which are member functions that do not change the abstract value of the default argument, but instead return some information about it. For the class `Person`, we might want a member function to return the person’s name and their age. We call these `Name` and `Years_Old`.

It is also possible to have a combination mutator and observer, but it is simpler to keep them separate.

Once you have designed the member functions, you can specify their interfaces and behavior in more detail. Things to decide include the following:

- What the parameter types are for constructors, mutators, and observers, and what the result type is for each mutator and observer.
• Whether some of the member functions are to be virtual, so that they will work with sub-
typing. Unless you have some efficiency reasons not to make them virtual, it seems best
to make all but the constructor virtual, as this gives greater flexibility. (The constructors, of
course, cannot be virtual.) Other C++ interface details, such as the use of const, can also
be decided here.

• What, informally, you want each member function to do. These informal statements can be
written as comments in the Larch/C++ formal specification.

Then you should be ready to write a formal specification, such as the following (which is discussed
further below).

```cpp
//@ uses cpp_string;

class Person {
public:
//@ spec int age; // age interpreted as number of years old
//@ spec String<char> name;

//@ invariant len(name\any) > 0 \ age\any >= 0;
//@ constraint age" <= age'; // age can only increase

Person(const char *moniker, int years) throw();
//@ behavior {
//@ requires nullTerminated(moniker,pre) \ lengthToNull(moniker,pre) > 0
//@ /
//@ modifies name, age;
//@ ensures name' = uptoNull(moniker,pre) \ age' = years;
//@ }

virtual ~Person() throw();
//@ behavior {
//@ ensures true;
//@ }

virtual void Change_Name(const char *moniker) throw();
//@ behavior {
//@ requires nullTerminated(moniker,pre) \ lengthToNull(moniker,pre) > 0;
//@ modifies name;
//@ ensures name' = uptoNull(moniker,pre);
//@ ensures redundantly len(name') > 0;
//@ }
```
virtual char * Name() const throw();
//@ behavior {
//@   ensures nullTerminated(result,post)
//@     \ fresh(objectsToNull(result,post))
//@     \ uptoNull(result,post) = name\any;
//@ }

virtual void Make_Year_Older() throw();
//@ behavior {
//@   requires age^ < INT_MAX;
//@   modifies age;
//@   ensures age' = age^ + 1;
//@   example age^ = 29 /\ age' = 30;
//@ }

virtual int Years_Old() const throw();
//@ behavior {
//@   ensures result = age\any;
//@ }
};

In the specification of the class Person above, the included file, ‘AbstractString.lh’, is the one shown above that specifies the type String<char>. The trait used, cpp_string, defines trait functions, such as nullTerminated and lengthToNull that are useful in dealing with C++ character strings. (See Section 11.9 [Character Strings], page 296 for details.)

The invariant specified for the class Person says that a person always has a name that has one or more characters in it, and that the person’s age is at least 0 (years). See Section 7.3 [Class Invariants], page 196 for more details on invariants.

The constraint specified for the class Person says that a person’s age may only increase over time. See Section 7.4 [History Constraints], page 200 for more details on such history constraints.

The constructor takes a null-terminated C++ string and an integer and uses them to initialize the specification variables name and age.

Since age and name are data members, they have sorts, Obj[int] and Obj[String[char]], respectively. Hence, to refer to their abstract values in the post-state, one must write name’ and age’. (See Section 2.8 [Objects and Values], page 21 for an introductory discussion about objects and values. See Section 6.2.1 [State Functions], page 103 for details on state functions such as ’.) Notice that the constructor’s precondition is needed to have a sensible C++ character string, and to establish the invariant in the post-state.
The destructor’s specification is standard; it says that the destructor does nothing.

See Section 6.9.3 [Redundant Ensures Clauses or Claims], page 141 for details on using a redundant 
ensures-clause such as the one in change_name.

See Section 6.9.1 [Examples in Function Specifications], page 137 for details on using an example such as the one in make_year_older.

It may be of some interest to see the LSL trait that Larch/C++ automatically (in theory) constructs for this specification. The following trait, Person_Trait assumes and then includes some traits relating to the types mentioned in the declaration of Person; then in the last set of includes, it generates the theory of values of Person objects, and then the theory of const Person objects.

\%
% This trait would be automatically constructed by Larch/C++

Person_Trait: trait

assumes AbstractStringTrait, int, cpp_member_function
includes MutableObj(int), MutableObj(String[char]),
ConstObj(int), ConstObj(String[char]),
ConstObj(cpp_member_function)
includes Person_Pre_Trait(Person, Obj, Val[Person]),
Person_Pre_Trait(Const[Person], ConstObj, Val[Person])

The trait Person_Pre_Trait that defines the theory of Person and Const[Person] with the various renamings is as follows. Note that the tuples defined for these sorts include all the (non-static) members of the class Person, including the member functions. See Section 11.10 [Structure and Class Types], page 299 for details and a similar example.

\%
% This trait would be automatically constructed by Larch/C++

Person_Pre_Trait(Person, Loc, Val): trait

assumes AbstractStringTrait, int, cpp_member_function,
TypedObj(Loc, String[char]), TypedObj(Loc, int),
ConstObj(cpp_member_function),
contained_objects(Loc[String[char]]), contained_objects(Loc[int])

includes NoContainedObjects(Val), contained_objects(Person)
Person tuple of age: Loc[int], name: Loc[String[char]],
destructor: ConstObj[cpp_member_function],
Change_Name: ConstObj[cpp_member_function],
Name: ConstObj[cpp_member_function],
Make_Year_Older: ConstObj[cpp_member_function],
Years_Old: ConstObj[cpp_member_function]

Val tuple of age: int, name: String[char],
destructor: cpp_member_function,
Change_Name: cpp_member_function,
Name: cpp_member_function,
Make_Year_Older: cpp_member_function,
Years_Old: cpp_member_function

introduces
eval: Person, State -> Val
allocated, assigned: Person, State -> Bool

asserts
\forall per: Person, oi: Loc[int], os: Loc[String[char]],
odestroy, och_nm, onm, omyo, oyo: ConstObj[cpp_member_function],
st: State

contained_objects([oi,os,odestroy,och_nm,onm,omyo,oyo], st) ==
\{asTypeTaggedObject(oi)} \U \{asTypeTaggedObject(os)}
\U \{asTypeTaggedObject(odestroy)} \U \{asTypeTaggedObject(och_nm)}
\U \{asTypeTaggedObject(onm)} \U \{asTypeTaggedObject(omyo)}
\U \{asTypeTaggedObject(oyo)};

eval(per,st) ==
\{eval(per.age, st), eval(per.name, st),
eval(per.destructor, st), eval(per.Change_Name, st),
eval(per.Name, st), eval(per.Make_Year_Older, st),
eval(per.Years_Old, st)};

allocated(per,st) ==
\{allocated(per.age, st) \U allocated(per.name, st) \U allocated(per.destructor, st) \U allocated(per.Change_Name, st) \U allocated(per.Name, st) \U allocated(per.Make_Year_Older, st) \U allocated(per.Years_Old, st)};

assigned(per,st) ==
\{assigned(per.age, st) \U assigned(per.name, st) \U assigned(per.destructor, st) \U assigned(per.Change_Name, st) \U assigned(per.Name, st) \U assigned(per.Make_Year_Older, st) \U assigned(per.Years_Old, st)};
implies
  converts
  contained_objects: Person, State -> Set[TypeTaggedObject],
  eval: Person, State -> Val,
  allocated: Person, State -> Bool, assigned: Person, State -> Bool

7.1.2 A Design with a Nontrivial Trait (Money)

For another example, consider the specification of a class Money. Although one could also use specification variables to specify Money, doing so would be inconvenient for clients of the specification. The inconvenience arises because Money does not have easily remembered “parts” and because it is likely to only have one “part.” This is often the case for types, like Money whose object cannot be mutated, and hence are immutable. For such types, it is better to specify their abstract values explicitly, using a LSL trait, instead of using specification variables. This allows clients of the specification to use the abstract values directly, instead of having to remember the name of a data member. It also makes Money act more like a C++ built-in type.

It is usually a good idea to look in Guttag and Horning’s LSL Handbook (Appendix A of [Guttag-Horning93]) to see if something like what you want has already been specified in LSL. Since there is no trait in Guttag and Horning’s LSL handbook for money, we will also specify our own trait.

To define the trait specifying the abstract values of Money, we begin by asking a question. What information is contained in money? In the USA, one might immediately think of a number of dollars and cents, making one think that Money should be modeled as a tuple of dollars and cents. However, if one uses such a trait, one quickly finds that adding and subtracting amounts of money are difficult. After thinking about this, it may occur to you that a better model of (US) money is a number of cents.

What trait functions need to be specified? One can follow the same general idea for specifying a Larch/C++ class, except that as there is no mutation or destruction in the mathematical world of LSL, one only needs constructors and observers. In the following trait there are two constructors: pennies, which takes a number of pennies and returns an amount of money, and money, which takes a rational and converts it into money by rounding down to the nearest cent. For observers, we will use dollars and cents, which extract the number of dollars and cents from a given amount of money. We call the trait that models this basic idea of money MoneyBasics, and put it in the file ‘MoneyBasics.lsl’.

% @(#)Id: MoneyBasics.lsl,v 1.1 1997/07/30 04:54:58 leavens Exp $
MoneyBasics: trait
   includes long, Floor(long for int)
   introduces
      money: Q -> Money
      pennies: long -> Money
      dollars, cents: Money -> long

asserts
   Money generated by pennies
   Money partitioned by dollars, cents
   \forall q: Q, p: long
      money(q) == pennies(floor(q*(100/1)));
      dollars(pennies(p)) == div(p, 100);
      cents(pennies(p)) == mod(p, 100);

implies
   Money generated by money
   \forall m: Money
      dollars(money(1:Q)) == 1;
      cents(money(1:Q)) == 0;
      pennies((100*dollars(m)) + cents(m)) == m;

converts
   money: Q -> Money,
   dollars: Money -> long,
   cents: Money -> long

In the asserts section of the trait, one often writes that the sort is generated by a subset of the primitive constructors that are sufficient to make all the values of the type. In this case, Money is generated by the trait function pennies. One also often writes that the sort is partitioned by a subset of the observers that are sufficient to distinguish any two values that should be distinguished. In this case, Money is partitioned by the trait function dollars and cents.

What equations to write? One generally should write an equation telling what trait function in the partitioned by list (or lacking such a list, each observer) returns for each primitive constructor (see [Guttag-Horning78] and page 54 of [Guttag-Horning93]).

In writing the equations it helps to think of functional programming. (Sometimes, although not illustrated by this example, it helps to think of induction with the constructors that do not take arguments of the sort being specified as the base case, and other constructors as inductive cases.)

You might also want to write some implications in the trait. These are written following the keyword implies, and can either highlight properties that follow from the axioms, or can state “obvious” properties that are used to debug the specification (as in chapter 7 of [Guttag-Horning93]). In the trait MoneyBasics the first implication says that Money is generated by the constructor
money. When one has a second way to specify a property, such as a way to generate Money, it is often useful to place that in an implication. The second and third implications, i.e., the first two implied equations, are intended to be obvious properties that are true of one dollar. The last implied equation says how to put dollars and cents back together. The converts section says that the trait functions money, dollars, and cents are well-defined relative to the other trait functions (i.e., pennies).

In the trait MoneyTrait below, we add to the basic theory some trait functions to allow money to be added and subtracted, compared, and checked for being in a certain range. For interest calculations, we also specify how to multiply a rational number and an amount of money.

When writing equations for the non-primitive constructors, one can specify them by stating what observers do when these functions are applied to combinations of primitive observers. But often it is convenient to define them as abbreviations that rewrite into the more primitive trait functions, which is what is done in the trait for +, −, and ∗.

% @(#)$Id: MoneyTrait.lsl,v 1.9 1997/07/30 04:54:59 leavens Exp $

MoneyTrait: trait
  includes MoneyBasics, DerivedOrders(Money)
  introduces
    __+__, __-__: Money, Money -> Money
    __*__: Q, Money -> Money
    __>__: Money, Money -> Bool
    MONEY_MAX, MONEY_MIN: -> Money
  inRange: Money -> Bool

  asserts
  \forall q: Q, p,p1,p2: long, m: Money
  pennies(p1) + pennies(p2) == pennies(p1 + p2);
pennies(p1) - pennies(p2) == pennies(p1 - p2);
q * pennies(p1) == pennies(floor(q * (p1/100)));
pennies(p1) > pennies(p2) == p1 > p2;
pennies(10000) < MONEY_MAX;
MONEY_MIN < pennies(-10000);
inRange(m) == MONEY_MIN <= m \&\& m <= MONEY_MAX;
Chapter 7: Class Specifications

implies
  TotalOrder(Money)
\forall q,q1,q2: Q, p,p1,p2: long, m,m1,m2: Money
  pennies(100*dollars(m)) + pennies(cents(m)) == m;
  m1 > m2 == dollars(m1) > dollars(m2)
  \(\lor\ (dollars(m1) = dollars(m2)
    \land\ cents(m1) > cents(m2));

converts
  __+__: Money, Money -> Money,
  __-__: Money, Money -> Money,
  __*__: Q, Money -> Money,
  __>__: Money, Money -> Bool

You may find, as we did, that you have to go back and add trait functions to your trait when
they are needed to specify some of the member functions of the class. For example, we needed to
add trait functions to specify an ordering on money values. Later we also realized that the LSL
Handbook in [Guttag-Horning93] does not have a trait function floor, which is needed. This could
have been put in the trait, but it seems to be a generally useful concept, so we made another trait
for it. That trait is given below.

Floor(int): trait
  includes Rational(int for Int)
  introduces
    floor: Q -> int
  asserts
    \forall q: Q
    (q-1) < (floor(q)/1);
    (floor(q)/1) <= q;
  implies
    \forall n: int, a,b: P
    floor(n/1) == n;
    floor(floor(n/a)/b) == floor(n/(a*b));

Even later, we realized that, in order to allow for finiteness of the representation of money, we
had to add the constants MONEY_MIN and MONEY_MAX to the trait, and the trait function inRange. It
would be a mistake (contradictory) to try to enforce these limits in the trait by adding an assertion
of the following form to the trait. The correct place to do that is in the invariant of the class. Thus
the following would be wrong in the trait.

\forall m: Money
  inRange(m);  \% error! contradictory in the trait!
Having written the trait for money, we can now design and specify the class Money. Since this is an immutable type, there are no mutator operations. For constructors, C++ has decided what they can be named. So we distinguish the constructors on the basis of their argument type. We use a constructor with a double precision floating point argument, and one with a number of pennies as an argument. For observers, we specify functions Dollars and Cents. We then specify addition, subtraction, interest multiplication, and various comparison functions. Thinking about addition and subtraction makes one think about overflow. To keep things fairly simple in this design, liberal specification is used to avoid exceptions (see Section 6.12 [Liberal Specifications], page 151). The next section has an example with exceptions (see Section 7.1.3 [A Class with Exceptions (Stack)], page 183). The design of Money resembles its used trait because the type is immutable.

```cpp

#ifndef Money_lh
#define Money_lh

//@ uses MoneyTrait;
//@ uses NoContainedObjects(Money);

class Money {
public:
//@ invariant inRange(self\any);
//@ constraint self' = self^; // Money is immutable

        Money(double amt) throw();
//@ behavior {
//@ requires inRange(money(rational(amt)));
//@ constructs self;
//@ ensures self' = money(rational(amt));
//@ }

        virtual ~Money() throw();
//@ behavior {
//@ ensures true;
//@ }

        Money(long int cts = 0L) throw();
//@ behavior {
//@ requires inRange(pennies(cts));
//@ constructs self;
//@ ensures self' = pennies(cts);
//@ }

        virtual long int Dollars() const throw();
//@ behavior {
//@ ensures result = dollars(self\any);
//@ }
```
virtual long int Cents() const throw();
//@ behavior {
//@   ensures result = cents(self\any);
//@ }

virtual Money & operator + (const Money & m2) const throw();
//@ behavior {
//@   requires assigned(m2, pre) \ inRange(self\any + m2\any);
//@   ensures returns;
//@ also
//@   requires assigned(m2, pre);
//@   ensures liberally fresh(result) \ assigned(result, post)
//@   \ result' = self\any + m2\any;
//@ example liberally self\any = money(300/1) \ m2\any = money(400/1)
//@   \ result' = money(700/1)
//@   \ fresh(result) \ assigned(result, post);
//@ }

virtual Money & operator - (const Money & m2) const throw();
//@ behavior {
//@   requires assigned(m2, pre) \ inRange(self\any - m2\any);
//@   ensures returns;
//@ also
//@   requires assigned(m2, pre);
//@   ensures liberally fresh(result) \ assigned(result, post)
//@   \ result' = self\any - m2\any;
//@ }

virtual Money & operator * (double factor) const throw();
//@ behavior {
//@   requires inRange(rational(factor) \ self\any);
//@   ensures returns;
//@ also
//@   ensures liberally fresh(result) \ assigned(result, post)
//@   \ result' = rational(factor) \ self\any;
//@ }

virtual bool operator == (const Money & m2) const throw();
//@ behavior {
//@   requires assigned(m2, pre);
//@   ensures result = (self\any = m2\any);
//@ also
//@   ensures redundantly result = (dollars(self\any) = dollars(m2\any)
//@   \ cents(self\any) = cents(m2\any));
//@ }

virtual bool operator > (const Money & m2) const throw();
//@ behavior {
//@ requires assigned(m2, pre);
//@ ensures result = (self\any > m2\any);
//@ }

virtual bool operator >= (const Money & m2) const throw();
//@ behavior {
//@ requires assigned(m2, pre);
//@ ensures result = (self\any >= m2\any);
//@ }

virtual bool operator < (const Money & m2) const throw();
//@ behavior {
//@ requires assigned(m2, pre);
//@ ensures result = (self\any < m2\any);
//@ }

virtual bool operator <= (const Money & m2) const throw();
//@ behavior {
//@ requires assigned(m2, pre);
//@ ensures result = (self\any <= m2\any);
//@ }

#endif

The first uses-clause in Money says that the trait MoneyTrait is to be used. The identity of
the name Money in this trait and the name of the class Money tells Larch/C++ that it should not
automatically construct a trait for Money’s abstract values, but that the abstract values are given
explicitly.

The use of the trait NoContainedObjects(Money) tells Larch/C++ that the abstract values of
Money objects do not contain subobjects. A definition of the contained_objects trait function
is needed for each sort of abstract value (see Section 7.5 [Contained Objects], page 207). When
the abstract values are defined explicitly by the user, then the contained_objects trait function
must also be defined. The most convenient way to do this when the abstract values contain no
subobjects, is as shown here.

The money abstract values specified in MoneyTrait may be of any size, but the implementation
will have to impose some limit. This is the reason for the invariant.

In the invariant, the Larch/C++ keyword self denotes an arbitrary, assigned, object of type
Money. Its sort, as an object, is Obj[Money]. This is the sort of self throughout this class
specification. The sort of \texttt{self\_any} is thus \texttt{Money}, which is the sort of values specified in the LSL trait \texttt{MoneyTrait}.

Money objects are immutable. For example addition of money produces a new money object. The history constraint says that, once a money object is created, its abstract value does not change.

In a history constraint, as in the invariant, \texttt{self} denotes an arbitrary, assigned, object of type \texttt{Money}.

In the rest of the specification, the keyword \texttt{self} refers to the object \texttt{*(this\_any)*}; i.e., it refers to the implicit receiver of a member function call or to the implicit object being constructed by a constructor.

The second constructor for \texttt{Money} specifies a default for its parameter. For example, the abstract value of \texttt{Money(3L)} is \texttt{pennies(3)}, and the abstract value of \texttt{Money()} is \texttt{pennies(0)}.

The specification of the member function \texttt{+} uses two cases, separated by the keyword \texttt{also}. The first case says that the function has to terminate when the sum is in range. The second says that if it terminates, it must do the right thing. (See Section 6.10 [Case Analysis], page 143 for details on having multiple \texttt{spec-case}s in a function specification. See Section 6.12 [Liberal Specifications], page 151 for details on the use of \texttt{liberally}.)

The specification of the member function \texttt{+} uses the trait function of the same name. This is not a difficulty for Larch/C++, as the meaning is clear because the C++ member function cannot be called in a predicate (see Section 4.6 [Identifiers], page 39 for worlds in which identifiers live). If you think the people reading your specification will find this kind of thing confusing, you can make things clearer for them by using different names for each. Another way to clear up such a potential confusion is to give an example, as is done in the specification of \texttt{+} (see Section 6.9.1 [Examples in Function Specifications], page 137).

Because \texttt{Dollars} and the other functions do not modify \texttt{self}, they refer to the abstract value of \texttt{self} using \texttt{self\_any}. This is a way of saying “either \texttt{\_pre} or \texttt{\_post}, it doesn’t matter” (see Section 6.2.1 [State Functions], page 103 for details).

A C++ idiom is to take instances as reference parameters to avoid the run-time overhead of passing them by value. This is done in the overloads of the C++ operators \texttt{+, -}, and so on. Also these overloaded operators return a reference to a freshly allocated \texttt{Money} object (see Section 6.3.1 [Fresh], page 123).

In the specification of \texttt{operator ==}, it should be noted that the C++ type \texttt{bool} is modeled by the sort \texttt{Bool}. (See Section 11.4 [bool], page 276 for details.)
7.1.3 A Class with Exceptions (Stack)

It is traditional to give a specification of some kind of stack as an example specification. It is almost too easy to specify unbounded stacks, by using the LSL Handbook trait Stack (page 170 of [Guttag-Horning93]). A specification of unbounded stacks is included in the Larch/C++ release. In this section, we specify bounded stacks of integers in the following. (See Chapter 8 [Template Specifications], page 239 for how to specify stacks of arbitrary types.) The abstract values of bounded integer stacks include both a stack of integers and a bound. To model this we again use specification variables. The specification variable max_size holds the maximum size of a BoundedIntStack. The specification variable stk holds a value specified by the trait mentioned above. We use a specification-only class declaration to declare the type of this variable, IntStack.

The interface specification of the class BoundedIntStack follows. Note that the history constraint for the class specifies that, although the bound is specified when an object is created, it cannot be changed thereafter.

The interface specified for BoundedIntStack uses exceptions (see Section 6.11 [Exceptions], page 148) and case analysis (see Section 6.10 [Case Analysis], page 143). The case analysis is used, for the functions that throw exceptions, to give two sets of pre- and postconditions: one for the “normal” case, and one for the case when the exception is thrown. This specification also illustrates how to specify mixed mutators and observers; for example, push returns the default argument (the object self). Note that result = self means that they are the same object.

```cpp
#include "StackError.lh"
//@ uses Stack(int for E, int for Int, IntStack for C);
//@ spec class IntStack;
class BoundedIntStack {
   public:
      //@ spec IntStack stk;
      //@ spec int max_size;

      //@ invariant max_size\any >= 0 /
      //@ size(stk\any) <= max_size\any;
      //@ constraint max_size" = max_size';

      BoundedIntStack(int limit = 50) throw();
      //@ behavior {
      //@ requires limit >= 0;
      //@ modifies max_size, stk;
      //@ ensures max_size' = limit /
      //@ stk' = empty;
      //@ }
```
virtual ~BoundedIntStack() throw();
//@ behavior {
//@   ensures true;
//@ }

virtual BoundedIntStack& push(int e) throw(StackError*);
//@ behavior {
//@   requires size(stk^) < max_size^;
//@   modifies stk;
//@   ensures returns /
//@     result = self
//@     /
//@     stk' = push(stk^, e);
//@   example max_size^ >= 2
//@     /
//@     stk" = push(\empty, 8) /
//@     e = 3
//@     /
//@     stk' = push(push(\empty, 8), 3) /
//@     result = self
//@     /
//@     returns;
//@ also
//@   requires size(stk^) >= max_size^;
//@   ensures throws(StackError*);
//@ example max_size^ = 2
//@   /
//@   stk" = push(push(\empty, 8), 3) /
//@   e = 7
//@   /
//@   stk' = push(push(\empty, 8), 3)
//@   /
//@   throws(StackError*);
//@ }

virtual BoundedIntStack& pop() throw(StackError*);
//@ behavior {
//@   requires ~isEmpty(stk^);
//@   modifies stk;
//@   ensures returns /
//@     result = self
//@     /
//@     stk' = pop(stk^);
//@   example stk^ = push(push(\empty, 8), 3)
//@     /
//@     returns /
//@     result = self
//@     /
//@     stk' = push(\empty, 8);
//@ also
//@   requires isEmpty(stk^);
//@   ensures throws(StackError*);
//@ }

virtual int top() const throw(StackError*);
//@ behavior {
//@   requires ~isEmpty(stk\any);
//@   ensures returns /
//@     result = top(stk\any);
//@   example stk\any = push(push(\empty, 8), 3) /
//@     result = 3;
//@ also
//@   requires isEmpty(stk\any);
//@   ensures throws(StackError*);
//@ }
virtual bool isEmpty() const throw();
//@ behavior {
//@   ensures result = (isEmpty(stk\any));
//@   ensures redundantly result = (size(stk\any) = 0);
//@ }

virtual bool isFull() const throw();
//@ behavior {
//@   ensures result = (size(stk\any) = max_size\any);
//@ }

virtual int numElems() const throw();
//@ behavior {
//@   ensures result = size(stk\any);
//@ }

};

The interface specification of BoundedIntStack above also makes rather liberal use of examples (see Section 6.9.1 [Examples in Function Specifications], page 137). While these tend to make the specification somewhat lengthy, they are invaluable in communicating with readers who do not take the trouble to look at the trait in detail.

The C++ interface specified for numElems says that it is a virtual function, which returns an int. It also is specified to be a const member function. Finally the specified interface tells C++ that no exceptions are thrown, by writing throw(). (If one omits this, then as in C++, the specification does not limit what exceptions can be thrown.)

The exceptions that are thrown are all from the included specification of the class StackError. This is specified as follows.

// @(#)Id: StackError.lh,v 1.4 1997/01/12 22:21:28 leavens Exp $
//@ uses NoInformationException(StackError);
class StackError { };
7.2 Class Member Specifications

The member-seq in a class specification specifies both data and function members of the class. The syntax for the member-seq is as follows. (The syntax for a constant-expression is exactly as in C++.)

\[
\text{member-seq ::= [ member-seq ] member-declaration} \\
\text{ | [ member-seq ] access-specifier :} \\
\text{ | [ member-seq ] larch-cpp-clause}
\]

\[
\text{larch-cpp-clause ::= uses-clause} \\
\text{ | simulates-clause} \\
\text{ | depends-clause} \\
\text{ | represents-clause} \\
\text{ | invariant-clause} \\
\text{ | constraint-clause}
\]

\[
\text{member-declaration ::= function-definition} \\
\text{ | [ decl-specifier-seq ] [ member-declarator-list ] ; [ fun-spec-body ]} \\
\text{ | [ decl-qualifier-seq ] ctor-declarator ; [ fun-spec-body ]} \\
\text{ | [ decl-qualifier-seq ] special-function-declarator ; [ fun-spec-body ]} \\
\text{ | [ ] nested-name-specifier [ template ] unqualified-id} \\
\text{ | using-declaration} \\
\text{ | template-declaration} \\
\text{ | spec-decl} \\
\text{ | refinement-member-decl}
\]

\[
\text{member-declarator-list ::= member-declarator [ , member-declarator ] ...}
\]

\[
\text{member-declarator ::= declarator [ constant-initializer ]} \\
\text{ | [ identifier ] : constant-expression}
\]

\[
\text{constant-initializer ::= = constant-expression}
\]

\[
\text{access-specifier ::= public | protected | private}
\]

A member-seq thus consists of member-declarations, uses-clauses, simulates-clauses, depends-clauses, represents-clauses, invariant-clauses, constraint-clauses, and access-specifiers. As in C++, the access-specifiers give the accessibility of the members specified following them up to the next access-specifier. Recall that the default access-specifier, if none is explicitly given, is private. Since usually one specifies public function members, it is important to remember to give an access-specifier explicitly. Members specified with a given access must be implemented by definitions with that access. Besides the usual specification of public members, one may also specify protected and private members in Larch/C++. See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for when and how to specify protected and private members.
In a protected or private interface specification, some data members that are not visible in the public interface may need to be modified. To allow this modification to take place, one uses a depends-clause. See Section 7.6 [The Depends Clause], page 213 for details on dependencies.

Invariants and constraints may also be specified anywhere a member may be specified; this is to facilitate invariants about data members, which can only be stated after such members are declared. See Section 7.3 [Class Invariants], page 196 for details on invariants. See Section 7.4 [History Constraints], page 200 for details on constraints.

A member-declaration can take several forms. The first implements function members. The second, with the syntax

\[ \text{[ decl-specifier-seq ] [ member-declarator-list ] ; [ fun-spec-body ]} \]

is the most common in Larch/C++. It is commonly used to declare and specify function members. See Chapter 6 [Function Specifications], page 82 for the syntax and semantics of a function-spec-body that is not specific to classes. It can also be used to simply declare function members, without giving a specification, to declare data members, to grant friendship, and to adjust access. Such a declaration is implemented by a matching declaration in the code, unless it is a spec-decl. See Section 7.11 [Specifying Exposed Data Members], page 235 for cases when you might want to use declarations that are not spec-decls to specify data members in a class. See Section 7.12 [Specifying Friends], page 237 for when and how to grant friendship. However, for simple class specifications, it is usual to only use spec-decls and to specify function members. To adjust access, the decl-specifier-seq are omitted and the member-declarator-list consists of a id-expression which contains :: (see Section 5.4 [Declarators], page 65 for the syntax). This adjusts the access to names inherited from base classes (see Section 7.8 [Specifying Derived Classes], page 214), and has the same meaning as in C++. See section 11.4 of [Ellis-Stroustrup90] for details.

The third form of a member-declaration is used to declare, and optionally specify, a constructor for a class. See Section 7.2.1 [Constructors], page 188 for more discussion and examples.

The fourth form is a using-declaration. This can also be used to adjust access to inherited members. See Section r.3.1 of [Stroustrup95] for details. See Section 5.5 [CPP Namespace and Using Declarations], page 78 for the syntax of using-declaration.

See Section 9.1 [Ghost Variables], page 254 for the syntax and semantics of spec-decl.

See Section 10.6 [Nested Refinement], page 268 for the syntax and semantics of a refinement-member-decl.
In the syntax of a member-declarator, the use of a colon (:) followed by a constant-expression is for specifying a C++ bit-field (see section 9.6 of [Ellis-Stroustrup90]). It would only be used in a class specification to record a detailed design decision.

Inside the specification of a member function, data members are considered to be in scope. Thus they do not need to be made explicitly made visible with an extern declaration (see Section 6.7 [Global Variables], page 134), as was done in early versions of Larch/C++.

If you specify the abstract values of a class explicitly, by giving a LSL trait that has a sort with the same name as the class, then you may not be able to refer to data members through self. For example, if you specify a class Counter with a public data member count, but you also explicitly use a trait that describes the abstract values of Counter instead of letting Larch/C++ automatically construct the trait for you, then you can only write self^.count in a member function specification if your trait has a trait function __.count defined on the abstract values. On the other hand, you could still use an invariant to specify a relationship between your data member and parts of the abstract values you specified. See Section 7.11 [Specifying Exposed Data Members], page 235 for an example.

The subsections below describe some details of various member specifications.

### 7.2.1 Constructors

In C++, a constructor is a member function with the same name as the class it appears in. For example, a constructor for class Person is named Person. A class can specify several constructors.

In C++, the compiler takes care of allocating storage for instances of classes (see section 12.6 of [Ellis-Stroustrup90]). So the job of a constructor is simply to initialize the newly created instance. That is, the constructor really modifies the abstract value of the instance, from an undefined initial value to the value desired. Thus by the semantics of Larch/C++, one has to state that self is modified in the modifies-clause. Thus, the typical modifies-clause in a constructor looks like the following (or something equivalent, see below).

```
modifies self;
```

(Hence it is not usually sensible for the constructor to be specified as a const member function.) However, since clients of a class need not be interested in such detail, in Larch/C++ one can use constructs instead of modifies in a modifies-clause. That is, one can write the following.

```
constructs self;

This does not affect the meaning of the modifies-clause, the same implementations satisfy the specification, but it gives an impression to readers that is more connotative. An example is given in the specification of Money above (see Section 7.1.2 [A Design with a Nontrivial Trait (Money)], page 175). See Section 6.2.3 [The Modifies Clause], page 110 for the syntax of the modifies-clause.

In C++, a constructor for a class has the same name as the class itself. There can be several constructors for each class, as long as they differ in their parameter types (see chapter 13 of [Ellis-Stroustrup90]).

It is often useful to specify constructors with default parameter values, as in the specifications of Money and BoundedIntStack (see Section 7.1 [Examples of Class Specifications], page 168). See Section 6.6 [Default Arguments], page 134 for details on how to do this.

A constructor that can be called with no parameters is called a default constructor (section 12.1 [Ellis-Stroustrup90]). A class without a default constructor (such as Person in Section 7.1.1 [A First Class Design (Person)], page 168) cannot be used as an element of an array. Thus, as a general rule, one should try to specify a default constructor for each class. For the class Person, it is not obvious what the name should be by default. One possibility for the default constructor of the class Person is as follows. However, such a specification is not completely satisfactory, because it is not clear that the “default” person should have age 0. (The term uptoNull("J. Doe",any) has sort String[char], and uptoNull is used to convert the C++ string constant, of sort Arr[Obj[char]] to that sort.)

    Person();
    //@ behavior {
    //@   constructs name, age;
    //@   ensures name’ = uptoNull("J. Doe",any) \ age’ = 0;
    //@ }

Although Larch/C++ will provide an implicit specification of the default constructor if you do not specify any constructors for a non-abstract class (see Section 7.2.3 [Implicitly Generated Member Specifications], page 191), doing so is usually a bad idea, because the implicit specification does not tell the user much. As a rule, you should always specify at least one constructor for each concrete class.
As in C++ (see section 12.6 of [Ellis-Stroustrup90]), no return type can be specified for a constructor. Furthermore, a constructor cannot be specified as virtual (but see section 6.7.1 of [Stroustrup91]).

As in C++, constructors and their specifications are not inherited.

[[[Need example of constructor that has an exposed data member and allocates storage for it dynamically.]]]

A constructor in a class \( T \) is a copy constructor if the type of its first (leftmost) parameter is \( \text{const } T\& \) or \( T\& \), and if all its other parameters have defaults (see Section 6.6 [Default Arguments], page 134). As in C++, a constructor in a class \( T \) cannot have just one parameter of type \( T \).

See Section 7.2.3 [Implicitly Generated Member Specifications], page 191 for the meaning of a class specification that does not declare any copy constructors.

### 7.2.2 Destructors

In C++, a destructor is a member function whose name is written with a twiddle (\( ~ \)) preceding the name of the class it appears in. For example, the destructor for class \( \text{Person} \) is named \( ~\text{Person} \).

In C++, a destructor’s main responsibility is to deallocate objects that the constructor (or other member functions) may have dynamically allocated (see section 12.4 of [Ellis-Stroustrup90]). Since such allocations are usually invisible to clients, the deallocations are also usually of no concern to clients. Thus the typical destructor specification merely says that calling the destructor modifies nothing and terminates. This can be written with the following spec-case.

\[
\text{ensures true;}
\]

Cases where one would want to say more in a destructor include cases where the data members are exposed (see Section 7.11 [Specifying Exposed Data Members], page 235), or cases where the class is being used as a repository of storage, which will all be deleted at the same time. In such cases, one might write a spec-case like the following. This says that the object may be modified and that the directly reachable objects in the object should not be reused after the call. (See Section 6.3.2.1 [Trashed], page 128 for the meaning of \( \text{trashed} \).)

\[
\begin{align*}
\text{modifies self;} \\
\text{trashes contained_objects(self^, pre);}
\end{align*}
\]
ensures trashed(contained_objects(self^, pre));

[[[Need a real example.]]]

A destructor may not be declared \texttt{const} (see section 9.3.1 of [Ellis-Stroustrup90]).

Because the C++ \texttt{delete} operator actually trashes an instance, not the destructor, in the specification of a destructor it is only necessary to list the object \texttt{self} in the \texttt{modifies-clause} if the destructor changes the abstract value (for example, by trashing exposed data members that are parts of the abstract value). (See Section 6.2.3 [The Modifies Clause], page 110 for details on the \texttt{modifies-clause}. See Section 6.3.2 [The Trashes Clause], page 126 for details on the \texttt{trashes-clause}.)

As in C++ (see section 12.4 of [Ellis-Stroustrup90]), no return type can be specified for a destructor. A class can specify at most one destructor, which is not inherited.

In contrast to constructors, one can, and one often should, specify a destructor as \texttt{virtual}. Ellis and Stroustrup recommend making the destructor \texttt{virtual} whenever there is at least one other \texttt{virtual} function (page 278 of [Ellis-Stroustrup90]).

See Section 7.2.3 [Implicitly Generated Member Specifications], page 191 for the meaning of a class specification with no destructor declared.

### 7.2.3 Implicitly Generated Member Specifications

C++ implicitly defines several member functions for the user if they are not explicitly defined (section 12.3c of [Ellis-Stroustrup90]). Implicitly defined member functions are \texttt{public}. If the C++ member function is implicitly defined, and if the class is not abstract, then Larch/C++ implicitly provides an appropriate specification. C++ implicitly generates member functions in the following cases.

- A default constructor (i.e., one with no parameters, (section 12.1 [Ellis-Stroustrup90]), if no constructor (with any number of arguments) for the class has been declared.
- A copy constructor (section 12.1 [Ellis-Stroustrup90]), if no copy constructor has been declared.
- A destructor (section 12.4 [Ellis-Stroustrup90]), if no destructor has been declared.
- An assignment operator (sections 5.17 and 12.8 of [Ellis-Stroustrup90]), if no assignment operator has been declared.
The interfaces for these member functions are given in the following.

```cpp
// @(#)$Id: default_interfaces.lh,v 1.2 1997/01/10 23:49:22 leavens Exp $

class T {
public:
   // default interfaces for member functions when not explicitly declared.
   T() throw(); // constructor
   T(const T& arg) throw(); // copy constructor
   ~T() throw(); // destructor
   T& operator = (const T& from) throw(); // assignment operator
};
```

As a general rule, the default specifications generated by Larch/C++ are sensible and useful. However, it is probably not a good idea to leave the specification of the default constructor to Larch/C++. Furthermore, if one specifies a class with virtual member functions, then one probably wants a virtual destructor, but the destructor implicitly generated by C++ is not virtual. Thus, for a class with virtual member functions, it is best to explicitly specify a virtual destructor.

The following describe the implicit specifications generated by Larch/C++. No specifications are implicitly generated for abstract classes.

Consider the specification of a class T. Then the default constructor specification, when implicitly generated is as follows. The specification just says that only the object self can be modified, nothing about the abstract value of self is guaranteed, except that it is assigned and that its immediate subobjects (e.g., specification variables) are assigned.

```cpp
#include "default_interfaces.lh"

T::T() throw();
//@ behavior {
//@ constructs self;
//@ ensures true;
//@ ensures redundantly assigned(self, post) \ assignd(self', post);
//@ }
```

The implicitly-generated destructor specification, when needed, is the following. The specification says that calls to the destructor are guaranteed to terminate, and the destructor may not change any objects. Hence the default destructor does nothing. (Recall that it is the C++ delete
operator that actually trashes a class instance, not the destructor.) Note that the implicitly generated destructor is not virtual; thus if you have virtual function members in a class, you probably should declare a destructor explicitly.

```
#include "default_interfaces.lh"
T::~T() throw();
//@ behavior {
//@  ensures true;
//@ }
```

The implicitly-generated copy constructor specification, when needed, is the following. The specification says that the value of the abstract value of the constructed object, \texttt{self''}, is the same as the value of the argument’s abstract value, \texttt{arg\_any\_any}. This corresponds to the code that C++ generates for the default copy constructor, which copies the values from each data member of \texttt{arg} to the corresponding data member of \texttt{self}. (See Section 6.2.2 [Allocated and Assigned], page 107 for the trait function \texttt{assigned} used in the pre-condition.)

```
#include "default_interfaces.lh"
T::T(const T& arg) throw();
//@ behavior {
//@  requires assigned(arg, any) /\ assigned(arg\_any, any);
//@  constructs self;
//@  ensures self'' = arg\_any\_any;
//@  ensures redundantly assigned(self, post);
//@ }
```

The implicitly-generated assignment operator specification, when needed, is the following. The specification says that the result is the object being assigned (\texttt{self}), and that the value of the abstract value of \texttt{self} in the post-state \texttt{self''} is the same as the value of the abstract value of the argument from.
As an example, the type Person specified above (see Section 7.1.1 [A First Class Design (Person)], page 168) has both a constructor and a destructor specified, so no constructor or destructor is implicitly generated. The destructor is specified because there are virtual member functions, and thus the implicitly generated non-virtual destructor would not be quite right. In Person, there is no copy constructor or assignment operator specified, so implicitly the following public member specifications are implicitly added to the specification of Person by Larch/C++. (See Chapter 10 [Refinement], page 257 for a discussion of this kind of refinement specification.)
The above specifications are written using `self`, instead of the two specification variables `name` and `age`. Thus, one has to look at the automatically-constructed trait `Person_Trait` (see Section 7.1.1 [A First Class Design (Person)], page 168) and recall that `self''` means `eval(eval(self, post), post)` (see Section 6.2.1 [State Functions], page 103) to understand it. The idea is that `eval(self, post)` is a tuple of objects, containing objects that represent the specification variables `age` and `name`. These objects will generally differ, and so what is really desired is that the values within these objects are the same in the post-state. This explains the second evaluation, which returns a tuple of values. A more readable form of the default specification’s meaning for the pre- and postconditions can be seen by looking at the redundant `requires-clause` and `ensures-clause` of each specification.

The class `Money` specified above (see Section 7.1.2 [A Design with a Nontrivial Trait (Money)], page 175) also does not have a copy constructor or assignment operator specified, so these are also implicitly specified by Larch/C++, as with the class `Person`. The second-layer of `eval` and `assigned` functions, which work on `Money` values, are defined in the trait `PureValue(Money)`, which is included in the trait `NoContainedObjects(Money)` (see Section 7.5 [Contained Objects], page 207 for details).

If one wishes to not have give clients access to one of these implicitly generated member functions, one has to, as in C++, declare it as `protected` or `private`. For example, it might be useful to specify the copy constructor and assignment operator as `private`, if one wishes to prevent copies of objects of the type from being made.

### 7.3 Class Invariants

The `invariant-clause` specifies an invariant property that must be true of all values of the type. More precisely, the specified property must be true of the abstract value of each assigned object of
Chapter 7: Class Specifications

the type in all visible states. (This semantics is based on the work of Poetzsch-Heffter [Poetzsch-
Heffter97].) The syntax is as follows.

\[
\text{invariant-clause ::= invariant [ redundantly ] predicate ;}
\]

In stating an invariant, one uses the state function \(\text{\texttt{\textbackslash any}}\). For example, one might write \texttt{\texttt{self\textbackslash any}} to refer to the abstract value of an arbitrary object of the class being specified. More commonly, as in the invariant of the class \texttt{Person} repeated below (see Section 7.1.1 [A First Class Design
(Person)], page 168), one applies \texttt{\textbackslash any} to the specification variables of the class. This says that the abstract values of assigned \texttt{Person} objects must have a \texttt{name} field that is not empty, and with a \texttt{age} field that is nonnegative.

\[
invariant \texttt{\texttt{\texttt{\texttt{len(name\textbackslash any)}} > 0 /\ \texttt{age\textbackslash any } >= 0;}}
\]

One helpful way to think of a class invariant is as a restriction on the space of abstract values for that class. For example, the invariant for \texttt{Person} objects given above can be thought of as saying that the subset of abstract values of the sort \texttt{Person} that is of interest is the subset in which the \texttt{name} field is not empty and the \texttt{age} field is nonnegative. The way this restriction is accomplished is by asserting that the invariant property is true of all assigned objects in each visible state. (Note that \texttt{self} and data members are not required to be assigned in the pre-state for constructors.)

Both the pre- and post-states of a public member function are visible states, since they can be seen by clients. However, hidden states may exist during the execution of a member function, and in these states an invariant may be temporarily violated. This is okay, because clients of a class never see such intermediate states.

Because the pre- and post-states are visible, it may also help to manifest invariants of assigned objects as redundant pre- and postconditions for each member function. That is, one can assume the invariant as a redundant precondition, for each assigned object, with \texttt{\textbackslash any} changed to \texttt{\textbackslash pre}. If desired, the invariant can be highlighted in function specifications by the use of a redundant \textit{requires-clause} (see Section 6.9.2 [Redundant Requires Clauses], page 140). Recall, however, that an object and its fields are not assigned upon entry to a constructor, so one would not assume the invariant in the precondition of a constructor.

Similarly, one can claim invariants for assigned objects as redundant postconditions, with \texttt{\textbackslash any} changed to \texttt{\textbackslash post}. If desired, these redundant predicates can be stated in a redundant \textit{ensures-clause}. This shows that whenever a function is allowed to modify an assigned object, for example, \texttt{self}, a correct implementation must ensure that the invariant holds for each such object in the post-state.
Although invariants can manifest themselves as redundant requires-clauses and ensures-clauses in function specifications, the logical formulas used in the invariant do not have to be redundant with the rest of the specification of a class. That is, they can add information to a specification that might otherwise be absent.

Redundancy can also be used to help debug invariants. An invariant with the keyword redundantly does not affect the meaning of a specification, but is used to state some redundant property that should follow from the non-redundant invariants.

The formalization of this interpretation of invariants is that the invariant states a theory that is true of all assigned objects of the appropriate type in all visible states. An invariant involving data members of an object of a class is asserted to hold for those data members in all assigned objects of that class in all visible states [Poetzsch-Heffter97]. An invariant involving static or global variables is asserted to hold for those variables whenever they are assigned in visible states.

The semantics requires the invariant to hold in all visible states substituted for any, for each such assigned object mentioned in the invariant.\footnote{Thanks to Peter Mueller for a discussion on this point, and for bringing [Poetzsch-Heffter97] to my attention.}

What happens if the specification of a function is written in such a way that it seems that a correct implementation would violate the invariant? Then the function becomes overconstrained, and cannot be correctly implemented, since a correct implementation would have to both satisfy and violate the invariant.

As an example, consider the following function specification, which might be a member in the class Person. We highlight the relevant parts of the invariant by stating redundant pre- and postconditions.
The problem is clearly seen by comparing the postcondition and the claimed redundant postcondition. It's clear that the post-state cannot make $age'$ both equal to -3 and also nonnegative.

To formalize this idea, we first rewrite the use of data members in invariants to make them refer to these data members through self. This rewriting takes advantage of the automatically-generated trait constructed for a class by Larch/C++, which treats specification variables as objects that are part of a tuple that makes up the abstract value of self. (See Section 7.1.1 [A First Class Design (Person)], page 168 for the trait Person_Trait.) For example, we can rewrite the invariant for Person as follows.

$$\text{invariant } \text{len}(\text{self}\_\text{any.name}\_\text{any}) > 0 \land \text{self}\_\text{any.age}\_\text{any} \geq 0;$$

Once this is done, we have an invariant-clause of the form invariant invariant_pred(self, any); this can be thought of as translated into the use of two traits. The uses-clause, exemplified by the one given below, would appear outside any class declaration in which the invariant-clause appears. The first trait used is always Invariant(T), where the sort of self is Obj[T]. The second trait, written as ITranslation below, would be different each time; it would be generated for each invariant clause as described below.

$$//@ \text{uses } \text{Invariant}(T), \text{ITranslation};$$

The trait Invariant is the following, which just includes two other traits.

```lsl
% @(#)Id: Invariant.lsl,v 1.1 1997/01/27 20:33:15 leavens Exp $
Invariant(T): trait
   includes Invariant_Visible(Obj, T),
      Invariant_Visible(ConstObj, T)
```
The trait Invariant_Visible that is included by the above is the following. This trait states that every assigned object in a visible state satisfies the invariant predicate, \textit{invariant_pred}. By including this trait for both regular and constant objects, all assigned objects of the class are covered. (See Section 2.8.1 [Formal Model of Objects], page 24 for the trait \textit{TypedObj}.)

\begin{verbatim}
% @(#)$Id: Invariant_Visible.lsl,v 1.1 1997/01/27 20:33:15 leavens Exp $
Invariant_Visible(Loc,T): trait
  includes TypedObj(Loc, T)
  assumes visible, i_pred(Loc,T)
  asserts
    \forall o: Loc[T], st: State
    (assigned(o,st) /\ visible(st)) => invariant_pred(o,st);
\end{verbatim}

The notion of a visible state is not formalized here, but the signature of the trait function \textit{visible} is given by the following. Note that \textit{visible(pre)} and \textit{visible(post)} should both be true for client functions, and for public member functions of a class.

\begin{verbatim}
% @(#)$Id: visible.lsl,v 1.1 1997/01/27 20:33:15 leavens Exp $
visible: trait
  includes State
  introduces
    visible: State -> Bool
\end{verbatim}

The other trait used in this semantics, written as \textit{ITranslation} in the example above, is generated from the particular predicate in the \textit{invariant-clause}. Each such trait translates the invariant’s predicate into a trait that defines the trait function \textit{invariant_pred}. This trait function must have the signature described by the two instantiations of the following trait, which is assumed by the instantiations of \textit{Invariant_Visible}.

\begin{verbatim}
% @(#)$Id: i_pred.lsl,v 1.1 1997/01/27 20:33:15 leavens Exp $
i_pred(Loc,T): trait
  assumes TypedObj(Loc,T)
  introduces
    invariant_pred: Loc[T], State -> Bool
\end{verbatim}

The translation would use the \textit{eval} trait function from the trait \textit{TypedObj}, so that, for example, occurrences of \texttt{self\_any} would become \texttt{eval(self, any)}. Hence the signature given above.

As an example of this translation, consider again the rewritten invariant from Person given above. The generated trait for this invariant might be expressed as follows.
The above trait just includes the following trait twice, once for regular and constant objects. In the following trait, we make use of the fact that self and any are not reserved words in LSL to make the translation be as close as possible to the original predicate.

\[
\forall self: \text{Loc}[\text{Person}], \forall any: \text{State}
\]
\[
\text{invariant}\_\text{pred}(self, any) \equiv \text{len}(\text{eval}(\text{eval}(self, any).\text{name}, any)) > 0 \\
/\ \text{eval}(\text{eval}(self, any).\text{age}, any) \geq 0;
\]

In the general case, an invariant may involve static variables and global variables. The idea of the formalization for these cases is the same except that no rewriting using self is involved. We omit the formal details for the sake of brevity.

An invariant-clause may appear anywhere a member-declaration may appear in a class specification. See Chapter 7 [Class Specifications], page 167 for the details of the syntax. We suggest that normally you write it near the top of the class specification. You can write multiple invariants in a class specification; this is just a shorthand for writing the conjunction of the given invariants. See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for how one can use the flexibility of syntax.

### 7.4 History Constraints

The constraint-clause specifies a property of on the history of values which an assigned object may take [Liskov-Wing93b] [Liskov-Wing94]. For example, in the specification of Person (see Section 7.1.1 [A First Class Design (Person)], page 168), the constraint says that a person’s age can only increase. Such specifications are particularly useful for specifying abstract classes, where there are no constructors [Liskov-Wing93b] [Liskov-Wing94]. They may also be used to state properties such as that the assigned objects of a class are immutable, or that certain parts of the abstract value do not change or change monotonically. For example, in the specification of Money (see Section 7.1.2
[A Design with a Nontrivial Trait (Money)], page 175), the constraint says that an assigned Money object’s abstract value never changes, hence money objects are immutable.

The syntax is as follows. The optional constrained-set allows a constraint to be stated that only applies to the functions listed.

\[
\begin{align*}
\text{constraint-clause} & ::= \text{constraint} \ [\text{redundantly}] \ \text{predicate} \ [\text{constrained-set}] \ ; \\
\text{constrained-set} & ::= \text{for} \ \text{fun-interface-list} \\
\text{fun-interface-list} & ::= \text{fun-interface} \ [\ , \ \text{fun-interface}] \ldots \\
& | \text{nothing} | \text{everything}
\end{align*}
\]

Ignoring the constrained-set for the moment, a history constraint specifies a relationship between each pair of visible states ordered in time. (See Section 6.2.1 [State Functions], page 103 for a definition of visible state.) This relationship must be both reflexive and transitive.

For example, the first state might be the state just before calling some public member function, and the latter state might be the state just after the call returns. Hence the two states will be referred to as the “pre-state” and the “post-state”. However, in general the two states do not have to be the pre- and post-states of a call, the “pre-state” just has to occur before the “post-state” in the execution of the program [Liskov-Wing93b] [Liskov-Wing94].

In a history constraint, one can use the state-functions \texttt{^} and \texttt{\pre} to refer to an assigned object’s value in the “pre-state”, and \texttt{'} and \texttt{\post} to refer to such a value in the “post-state”. For example, recall the history constraint from the specification of the class Money (see Section 7.1.2 [A Design with a Nontrivial Trait (Money)], page 175). This history constraint, given below, says that the abstract values of assigned Money objects do not change over time.

\[
\text{constraint self}^\pre = \text{self}';
\]

Therefore this is a way of saying that Money objects are immutable.

As with invariants, it may be helpful to think of a history constraint as being manifested for assigned objects in redundant ensures-clauses of public member functions. That is, the history constraint is a property that is added to the theory of pairs of visible states for objects of the type. For example, if constraint\_predicate(self, pre, post) is the given predicate, then for each pair of visible states such that pre precedes post in an execution, the following must hold.

\[
\text{assigned}(self, pre) \Rightarrow \text{constraint\_predicate}(self, pre, post)
\]
Although the constraint can be made manifest through redundant claims in member function specifications, the constraint is not necessarily redundant with the specifications given in the class. Instead, the constraint is an additional obligation that must be met by an implementation of the class.

Redundancy can also be used to help debug history constraints. A history constraint with the keyword `redundantly` does not affect the meaning of a specification, but is used to state some redundant property that should follow from the non-redundant history constraints.

If a function is specified in a way that would force a correct implementation to violate the history constraint, then the function will not be correctly implementable. Similarly, the class as a whole must preserve the constraints to be correctly implementable.

As with invariants, the formal semantics of a history constraint can be given by translating it into the use of two traits. The `uses-clause`, exemplified by the one given below, would appear outside any class declaration in which the `constraint-clause` appears. The first trait used is always `HistoryConstraint(T)`, where the sort of `self` is `Obj[T]`. The second trait, written as `HCTranslation` below, would be different each time; it would be generated for each constraint clause as described below.

```latex
//@ uses HistoryConstraint(T), HCTranslation;
```

The trait `HistoryConstraint` is the following, which just includes two other traits.

```latex
% @(#)Id: HistoryConstraint.lsl,v 1.1 1997/01/27 20:33:15 leavens Exp $
HistoryConstraint(T): trait
    includes Constraint_Visible(Obj, T),
    Constraint_Visible(ConstObj, T)
```

The trait `Constraint_Visible` that is included by the above is the following. This trait states that if an object is assigned in two visible states, then the history constraint predicate, `constraint_pred` is satisfied by that object in the two states. By including this trait for both regular and constant objects, all assigned objects of the class are covered. (See Section 2.8.1 [Formal Model of Objects], page 24 for the trait `TypedObj`.)
Constraint_Visible(Loc,T): trait
   includes TypedObj(Loc, T)
   assumes visible, follows, c_pred(Loc,T)
   asserts
      \forall o: Loc[T], pre, post: State
         (assigned(o,pre) \& \ assigned(o,post)
           \& visible(pre) \& visible(post)
           \& follows(pre, post))
      => constraint_pred(o,pre,post);

See Section 7.3 [Class Invariants], page 196 for the trait visible.

The notion of one state following another in a computation is not formalized here, but the
signature of the trait function follows is given by the following trait. Note that follows(pre, post) should be true for any function.

follows: trait
   includes State
   introduces
      follows: State, State -> Bool

The other trait used in this semantics, written as HCTranslation in the example above, is
generated from the particular predicate in the constraint-clause. Each such trait translates the
invariant’s predicate into a trait that defines the trait function constraint_pred. This trait
function must have the signature described by the two instantiations of the following trait, which
is assumed by the instantiations of Constraint_Visible.

c_pred(Loc,T): trait
   assumes TypedObj(Loc,T)
   introduces
      constraint_pred: Loc[T], State, State -> Bool

The translation would use the eval trait function from the trait TypedObj, so that occurrences
of self^* would become eval(self, pre), and occurrences of self’ would become eval(self, post).
As an example of this translation, consider again the history constraint from the class `Money` given above. The generated trait for this invariant might be expressed as follows.

```lsl
trait MTranslation:
    includes MoneyConstraint(Obj), MoneyConstraint(ConstObj)
```

The above trait just includes the following trait twice, once for regular and constant objects. In the following trait, we make use of the fact that `self`, `pre`, and `post` are not reserved words in LSL, so that the translation is close as possible to the original predicate.

```lsl
trait MoneyConstraint(Loc):
    includes c_pred(Loc,Money), MoneyTrait
    asserts
        \forall self: Loc[Money], pre, post: State
        constraint_pred(self, pre, post) == eval(self,pre) = eval(self,post);
```

In a `constraint-clause`, the optional `constrained-set` lists a number of member functions or friend functions. The same function must not appear twice in the list, and only member and friend functions may be listed. A `constrained-list` of the form for `everything` is similar to listing all the member and friend functions (except for the basic constructors), but applies to all member and friend functions, even when new ones are added. This has an effect on public subclasses as well, as the constraint applies to all of their member functions by specification inheritance (see Section 7.9 [Inheritance of Specifications and Subtyping], page 215). An omitted `constrained-list` is shorthand for writing a `constrained-set` of the form for `everything`.

With a `constrained-set` that explicitly lists various member and friend functions, the history constraint has a more limited meaning. It says that the constraint must hold for every assigned object which is manipulated using only the listed functions. (In terms of the semantics given above, this means that `follows(pre, post)` should not hold if the transformation from `pre` to `post` involves a call of a member or friend function that is not listed.)

History constraints with `constrained-sets` can also be used to say that certain attributes of an object are not changed by certain member or friend functions.
As an example, consider the following specification of the class Person2. In this class the first history constraint applies to all member functions (except the constructors), but the rest have a constrained-set.

The second history constraint says that the member functions name and years_old do not change the abstract value of an object of class Person. Without the listed constrained-set, the constraint would contradict other parts of the specification. (Whole interfaces of functions must be used because C++ allows overloading.)

The third history constraint says that change_name does not change the age of a Person2 object. Note that it conjoining the predicate \( age' = age^- \) to the postcondition of change_name would be equivalent. (This is an example of the idea for understanding history constraints described above.)

The fourth history constraint says that make_year_older does not change the name of a Person2 object.

```cpp
#include "AbstractString.lh"
//@ uses cpp_string;
class Person2 {
    public:
        //@ spec int age; // age interpreted as number of years old
        //@ spec String<char> name;

        //@ invariant len(name\any) > 0 /\ age\any >= 0;

        //@ constraint age^- <= age'; // age can only increase
        //@ constraint name^- = name' /\ age^- = age',
        //@ for virtual char * name() const,
        //@ constraint age' = age^,
        //@ for virtual void change_name(const char *);
        //@ constraint name' = name^-,
        //@ for virtual void make_year_older();

        Person2(const char *moniker, int years) throw();
        //@ behavior {
        //@ requires nullTerminated(moniker,pre) /\ lengthToNull(moniker,pre) > 0
        //@ /
        //@ modifies name, age;
        //@ ensures name' = uptoNull(moniker,pre) /\ age' = years;
        //@ }
```
virtual ~Person2() throw();
//@ behavior {
//@   ensures true;
//@ }

virtual void Change_Name(const char *moniker) throw();
//@ behavior {
//@   requires nullTerminated(moniker,pre) \ lengthToNull(moniker,pre) > 0;
//@   modifies name;
//@   ensures name' = uptoNull(moniker,pre);
//@   ensures redundantly len(name') > 0;
//@ }

virtual char * Name() const throw();
//@ behavior {
//@   ensures nullTerminated(result,post)
//@     \ fresh(objectsToNull(result,post))
//@   \ uptoNull(result,post) = name\any;
//@ }

virtual void Make_Year_Older() throw();
//@ behavior {
//@   requires age^ < INT_MAX;
//@   modifies age;
//@   ensures age' = age^ + 1;
//@   example age^ = 29 \ age' = 30;
//@ }

virtual int Years_Old() const throw();
//@ behavior {
//@   ensures result = age\any;
//@ }
};

Note that second, third, and fourth history constraints of Person2 are redundant, in the sense that they already follow from the modifies-clauses of the relevant functions.

The first history constraint in Person2 is redundant in a sense, because one can recover it from an analysis of the specification; however it is a property of the class as a whole.

History constraints are also not redundant in the specification of abstract classes (see Section 7.10 [Abstract Classes], page 234).

One way to check that a class satisfies a history constraint is to check that the pre- and post-states of each member and friend function listed in the constrained-set satisfy it, and to ensure that the
Chapter 7: Class Specifications

class's abstraction barrier is never violated. (This is called the history rule in [Liskov-Wing94].) If
the constrained-set is for everything or omitted, then this restriction must, of course, be satisfied
by all the member and friend functions. (Note, however, that the basic constructors, for which
self is not assigned in the pre-state, trivially satisfy the constraint with respect to self.)

7.5 Contained Objects

Larch/C++ relies on the specifier of a class whose abstract values are explicitly given by a LSL
trait to tell it what objects may be contained in the abstract values of a class. This is needed to give
meaning to the lcpp-primary reach(x) (see Section 6.2.3.5 [Reach], page 120), and in various other
syntactic sugars. Therefore, when explicitly specifying the abstract values of a class with an LSL,
as opposed to using specification variables, one needs to give some thought to whether the abstract
values themselves contain objects. Once this is decided, the trait function contained_objects
should be defined to tell Larch/C++ about any contained objects in the abstract value, along with
their types.

The signature required for the trait function is given by the following trait. (See Section 6.2.3
[The Modifies Clause], page 110 for the trait TypeTaggedObject.)

% @(#)$Id: contained_objects.lsl,v 1.9 1995/11/10 06:47:25 leavens Exp $
% Assumption about the sort of contained_objects, and useful includes
contained_objects(T): trait
  includes State, TypeTag(T)
  introduces
    contained_objects: T, State -> Set[TypeTaggedObject]

Often the abstract values of a class do not contain any objects, only pure values. In this
case, defining the contained_objects trait function is easy. One simply includes the LSL trait
NoContainedObjects, with an appropriate renaming or sort parameter. For example, in specifying
the class Money (see Section 7.1.2 [A Design with a Nontrivial Trait (Money)], page 175), we used
the trait NoContainedObjects(Money). The trait NoContainedObjects is defined as follows.
NoContainedObjects(V) : trait
    includes PureValue(V), % defines eval, allocated, assigned
    contained_objects(V) % gives sort assumption
    asserts
    \forall v: V, st: State
    contained_objects(v, st) == {};

Besides defining the trait function contained_objects, this trait also defines three other trait functions, as described by the following trait. These help with various defaults in Larch/C++, specifically they make sense out of the default copy constructor and the default assignment operator (see Section 7.2.3 [Implicitly Generated Member Specifications], page 191), and they make sense out of the implicit preconditions for member functions of a class (see Section 6.2.2 [Allocated and Assigned], page 107).

PureValue(V) : trait
    assumes State
    introduces
    eval: V, State \rightarrow V
    allocated, assigned: V, State \rightarrow Bool
    asserts
    \forall v: V, st: State
    eval(v, st) == v;
    allocated(v, st);
    assigned(v, st);

To define one's own version of the trait function contained_objects, one uses the trait function asTypeTaggedObject from the following trait.
TypeTag(T): trait
    assumes State, WidenNarrow(T, Object)
    includes TypeTaggedObject, IgnoringTypeTags,
    SortNames(T, TYPE, type_of for sort_of)
introduces
    asTypeTaggedObject: T -> TypeTaggedObject
asserts
    \forall o: T
    asTypeTaggedObject(o) == [widen(o), type_of(o)];
implies
    \forall o: T
    asTypeTaggedObject(o).obj == widen(o);

See Section 6.2.3.4 [Formal Details of the Modifies Clause], page 116 for IgnoringTypeTags. The trait SortNames is given below. It specifies a metalogical operation, sort_of, that is supposed to reify sorts in LSL. This operation is not completely formalized here.

SortNames(S, SORTNAME): trait
introduces
    sort_of: S -> SORTNAME
asserts
    \forall s, s1: S
    sort_of(s) == sort_of(s1);

For an example of a class with abstract values that contain objects, we will specify a class PersonSet. This class will have abstract values that are Person objects (see Section 7.1.1 [A First Class Design (Person)], page 168 for the specification of Person). A PersonSet, as a set of objects, may contain two distinct Person objects with the same abstract value. This reflects the reality that, just because two people have the same name and age, does not mean that they are identical. Therefore, for the abstract values of PersonSet, we use the LSL Handbook trait Set (page 167 of [Guttag-Horning93]), but for the elements we use Obj[Person], not Person. Hence we write the following trait. This trait adds the trait functions contained_objects, eval, allocated, and assigned to the trait functions defined in the included traits. (See Section 2.8.1 [Formal Model of Objects], page 24 for the trait MutableObj. See Section 7.1.1 [A First Class Design (Person)], page 168 for the trait Person_Trait.) Specifying the trait function eval allows one to write

\footnote{2 Personal communication, J. Smith.}


self’’ to mean the set of values of the objects in the post-state value of self. We make the trait functions allocated and assigned always return true for PersonSet values, so that the implicit preconditions of member functions (see Section 6.2.2 [Allocated and Assigned], page 107) do not require these objects to be assigned.

Once this is decided, the interface specification of PersonSet can be written. An interface specification follows, which is discussed further below.

```c++
// @(#)$Id: PersonSet.lh,v 1.30 1997/07/31 02:43:26 leavens Exp $
#include "Person.lh"

//@ uses PersonSetTrait(PersonSet for Set[Obj[Person]]);

class PersonSet {
public:
  //@ invariant \A p: Obj[Person] ((p \in (self\any)) => assigned(p, any));
  
  PersonSet() throw();
  //@ behavior {
  //@   modifies self;
  //@   ensures self' = {};
  //@  }
  
  virtual ~PersonSet() throw();
  //@ behavior {
  //@   ensures true; // persons in self not deleted
  //@  }
```
virtual void add(Person& e) throw();
//@ behavior {
//@ requires assigned(e, pre);
//@ modifies self;
//@ ensures self' = self^ \U {e} ;
//@ }

virtual void remove(Person& e) throw();
//@ behavior {
//@ modifies self;
//@ ensures self' = delete(e, self^) ;
//@ }

virtual bool member(Person& e) const throw();
//@ behavior {
//@ ensures result = (e \in self^) ;
//@ }

virtual int size() const throw();
//@ behavior {
//@ ensures result = size(self\any) ;
//@ }

virtual void bump_years() throw();
//@ behavior {
//@ requires \A p: Obj[Person] ((p \in (self\any)) => assigned(p, pre));
//@ modifies contained_objects(self, any);
//@ ensures \A p: Obj[Person]
//@ ((p \in (self\any))
//@ => (p'.age' = p^.age^ + 1 /
//@ p'.name' = p^.name^)) ;
//@ example \E p1: Obj[Person], p2: Obj[Person]
//@ (assigned(p1, pre) /
//@ assigned(p2, pre)
//@ /
//@ self^ = {p1} \U {p2} /
//@ p1^.age^ = 19 /
//@ p1^.name^ /
//@ p2^.age^ = 27
//@ /
//@ self' = {p1} \U {p2} /
//@ p1'.age' = 20 /
//@ p2'.age' = 28
//@ /
//@ p1'.name' = p1^.name^ /
//@ p1'.name' = p1^.name^) ;
//@ }
PersonSet& copy() const throw();
//@ behavior {
//@ requires \A p: Obj[Person] ((p \in (self\any)) => allocated(p, pre));
//@ ensures fresh(result, contained_objects(result, post))
//@ /
//@ result’ = self\any\any;
//@ example \E p1: Obj[Person], p2: Obj[Person],
//@ pnew: Obj[Person], p1new: Obj[Person]
//@ (assigned(p1, pre) /
//@ assigned(p2, pre)
//@ /
//@ self\any = {p1} \U {p2}
//@ /
//@ p1\any.age = 19 /
//@ p1\any.name = A"fred"
//@ /
//@ p1\any.age = 27 /
//@ p1\any.name = A"sue"
//@ /
//@ fresh(result, p1new, p2new)
//@ /
//@ result’ = {p1new} \U {p2new}
//@ /
//@ p1new’.age = 19 /
//@ p1new’.name = A"fred"
//@ /
//@ p1new’.age = 27 /
//@ p1new’.name = A"sue";
//@ }

PersonSet& copy1() const throw();
//@ behavior { // one-level copy
//@ ensures fresh(result) /
//@ assigned(result, post)
//@ /
//@ result’ = self\any;
//@ example \E p1: Obj[Person], p2: Obj[Person]
//@ (self\any = {p1} \U {p2}
//@ /
//@ fresh(result) /
//@ result’ = {p1} \U {p2});
//@ }

Notice that in the specification of PersonSet, reference parameters are used to pass Person objects, instead of Person values. For example, the sort of e in the member function add is Obj[Person] (see Section 6.1.8.1 [Sorts for Formal Parameters], page 94).

The specification of the member function bump_years deserves some comment. Notice that the abstract value of self is not changed by bump_years; what is changed are the abstract values of all the Person objects contained in self. To specify this, the function contained_objects is used in the modifies-clause (see Section 6.2.3 [The Modifies Clause], page 110). The postcondition is stated using a quantification over person objects (person values would not do).

The specifications of the member functions copy and copy1 make an interesting contrast. The function copy makes a copy both of self and the contained Person objects; this is why the notation self\any\any is used. The function copy1 returns a newly allocated PersonSet object, but this set shares the contained objects that are its elements with self.
7.6 The Depends Clause

A *depends-clause* is used to specify that some object depends on other objects [Leino95]. The syntax is as follows.

```
depends-clause ::= depends store-ref on store-ref-list ;
```

The first *store-ref* must name a single object, which will be referred to as the *abstract object* below. The *store-ref-list* names the *concrete objects* that the abstract object depends on.

The effect of a *depends-clause* is to allow the concrete objects to be modified when the abstract object is mentioned in a *modifies-clause*, and to be trashed when the abstract object is mentioned in a *trashes-clause*. (See Section 6.2.3 [The Modifies Clause], page 110 for details on the *modifies-clause*. (See Section 6.3.2 [The Trashes Clause], page 126 for details on the *trashes-clause*.)

See Section 7.9.1 [Inheritance of Specifications with Specification Variables], page 216 for an example of the use of the *depends-clause* in specifying a derived class.

7.7 The Represents Clause

A *represents-clause* is used to how some object represents on other objects that depend on it [Leino95]. The syntax is as follows.

```
represents-clause ::= represents store-ref by predicate ;
```

The first *store-ref* must name a single object, which will be referred to as the *abstract object* below. The *predicate* specifies the relationship that holds between the abstract object and some *concrete objects* that it depends on.

The *represents-clause* describes relationships that hold in all visible states. Unlike an invariant, which gives a proof obligation, the represents clause gives something that aids a proof. When used in a class, it can be thought of as conjoined to every public member function’s pre- and post-condition.

[[[Detailed semantics to be written.]]]
See Section 7.9.1 [Inheritance of Specifications with Specification Variables], page 216 for an example of the use of the represents-clause in specifying a derived class. See Section 10.2 [Class Refinement], page 260 for other examples.

7.8 Specifying Derived Classes

One can specify that a class is to be derived from some other class using the same syntax as in C++ (chapter 10 of [Ellis-Stroustrup90]). Thus the syntax of the base-spec of a class-specifier is as follows. See Section 7.2 [Class Member Specifications], page 186 for the syntax of access-specifier. See Section 5.2.3.2 [Class and Namespace Names], page 61 for the syntax of complete-class-name.

```
base-spec ::= : base-list
base-list ::= base-specifier [ , base-specifier ] ...
base-specifier ::= [ virtual ] [ access-specifier ] complete-class-name
| access-specifier [ virtual ] complete-class-name
```

All of this is part of the interface specified for a class in Larch/C++. For a C++ class to satisfy this part of the specified interface, it must have at least the base classes listed, in the base-list (with the specified access-specifiers and virtual attributes). A C++ class satisfying a Larch/C++ specification may have more than the listed base classes. In particular, it may have more private and protected base classes, as these are not usually of concern to clients. See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for a discussion of when you might want to specify such information about an implementation.

Whether a base class is declared to be public has an impact on the C++ type system (section 4.6 of [Ellis-Stroustrup90]). It also has an impact on the meaning of the specification, through specification inheritance (see Section 7.9 [Inheritance of Specifications and Subtyping], page 215).

Recall that the default access-specifier for the base in a class declaration is private (section 11.2 of [Ellis-Stroustrup90]). Good style recommends explicitly specifying giving the access-specifier for all base classes.

7.9 Inheritance of Specifications and Subtyping

As in C++, a derived class inherits members of its base classes, and can access the public and protected members it inherits. This is also true of the interface information in a Larch/C++ class
specification. In C++, the behavior of the base classes is also inherited to some extent, because the code for virtual member functions in base classes is inherited. However, because one can override definitions of member functions in derived classes, there is no guarantee that the objects of a derived class have any behavioral relationship to objects of its base classes. This is perfectly acceptable for protected and private inheritance, but not for public inheritance.

To see why public inheritance is special, consider a class `PlusAccount` which is derived from a public base class `BankAccount`. A class derived from a public base class is called a subtype. In our example, `PlusAccount` is a subtype of `BankAccount`. We also say that `BankAccount` is a supertype of `PlusAccount`.

In C++ a pointer variable, `ba`, of type `BankAccount *` can point to either a `BankAccount` object, or to a `PlusAccount` object. (Similarly, a C++ reference to a `BankAccount` can be an alias for a `PlusAccount`.) What happens when one executes the following C++ code?

```
ba->deposit(Money(50.00))
```

Depending on the run-time type of `*ba`, C++ will execute either `BankAccount::deposit` or `PlusAccount::deposit`. This dynamic binding of a member function to a call can be thought of as a kind of overloading that is resolved at run-time. It allows C++ programs to be polymorphic, and is one of the main features of object-oriented programming.

However, the use of dynamic binding can make reasoning about programs difficult [Leavens-Weihl90] [Leavens91] [Leavens-Weihl95]. In thinking about what `ba->deposit(Money(50.00))` does, one should not have to do a case analysis for each subtype. Instead, one would like to reason about what this means using the static type of `ba`, as would be done if there were no subtypes. This is the idea of supertype abstraction, which means letting supertypes stand for all of their subtypes in reasoning. The advantage of using supertype abstraction in reasoning is that such reasoning does not have to change when new subtypes are added to the program. That is, such reasoning is modular [Leavens-Weihl90] [Leavens91] [Leavens-Weihl95]. (This is important, because one of the things typically done in object-oriented programs is to add in new subtypes of existing types.)

However, supertype abstraction is only valid as a reasoning technique if certain semantic conditions are placed on the behavior of the subtype. A subtype that satisfies such conditions is called a behavioral subtype. The exact conditions that are needed for behavioral subtyping are still a matter of research, but sufficient conditions are already clear: for each virtual member function of each supertype, the subtype’s member function (with the same name and argument types) must satisfy the specification of the supertype’s member function [America87] [Meyer88] [Liskov-Wing93]
[Liskov-Wing93b] [Leino95] [Dhara-Leavens96] [Dhara97]. (There are also various restrictions on

types and accessibility, but these are already enforced by C++.)

To support supertype abstraction, a derived class specification in Larch/C++ inherits specifications from its public base classes. This is done in such a way as to force the derived type to be a behavioral subtype of its public base classes [Dhara-Leavens96]. Larch/C++ does not use specification inheritance for protected and private base classes, which allows users to specify derived classes that are not behavioral subtypes (of non-public base classes).

The way specification inheritance works in Larch/C++ is as follows. Consider a class specification Derived. If both the Base and Derived have an implicitly-defined trait (that is, if they both use specification variables), then the abstract values of the sort Base are obtained from the abstract values of Derived by simply omitting the specification variable members not found in Base. However, if either Base or Derived are the names of sorts from explicitly-given LSL traits, then the user must tell Larch/C++ how the abstract values of sort Derived can be interpreted as abstract values of sort Base. In either case, there is a mapping defined between the abstract values of Derived in a state and the abstract values of Base in that state. Using this mapping, the specification of each virtual member function of class Base can be inherited by class Derived; that is, the specification of each virtual member function of class Base must be satisfied by the corresponding member function of class Derived.

We first describe the more usual case, where the specifications of all classes involved are written using specification variables. Then we describe the case where explicitly-given LSL traits are used for some of the classes. Following that, we describe some more details and briefly discuss related work.

### 7.9.1 Inheritance of Specifications with Specification Variables

Inheritance of specifications for classes that are written using specification variables involves simply copying the specifications of the virtual member functions from the base class to the derived class. Since the specification variables of the base class are also in the derived class, these specifications automatically have a sensible meaning.

For an example, consider again the types BankAccount and PlusAccount. We specify the abstract values of the supertype by using two specification variables: credit and owner. The credit is a specification variable of type Money, and owner is a specification variable of an abstract string type.
/@(#)$Id: BankAccount.lh,v 1.17 1997/07/28 21:02:52 leavens Exp $

#ifndef BankAccount_h
#define BankAccount_h

#include "Money.lh"
#include "AbstractString.lh"

class BankAccount {
public:
//@ spec Money credit;
//@ spec String<char> owner;

//@ invariant credit\any >= 0;
//@ constraint owner^ = owner';

BankAccount(Money amt, const char *own) throw();
//@ behavior {
//@ uses cpp_const_char_string;
//@ requires pennies(1) <= amt /
nullTerminated(own, pre);
//@ modifies credit, owner;
//@ ensures credit' = amt /
owner' = uptoNull(own, pre);
//@ }

virtual ~BankAccount() throw();
//@ behavior {
//@ ensures true;
//@ }

virtual Money balance() const throw();
//@ behavior {
//@ ensures result = credit\any;
//@ }

virtual void pay_interest(double rate) throw();
//@ behavior {
//@ requires 0.0 <= rate /
rate <= 1.0;
//@ modifies credit;
//@ ensures credit' = rational(1.0 + rate) * credit^;
//@ example rate = 0.05 /
credit^ = money(400/1)
//@ /
credit' = money(420/1);
//@ }
virtual void deposit(Money amt) throw();
//@ behavior {
//@ requires amt >= 0;
//@ modifies credit;
//@ ensures credit' = credit^ + amt;
//@ example credit^ = pennies(40000) \ amt = pennies(1)
//@ \ credit' = pennies(40001);
//@ }

virtual void withdraw(Money amt) throw();
//@ behavior {
//@ requires 0 <= amt \ amt <= credit^;
//@ modifies credit;
//@ ensures credit' = credit^ - amt;
//@ example credit^ = pennies(40001) \ amt = pennies(40000)
//@ \ credit' = pennies(1);
//@ }
}
#endif

See Section 7.1.2 [A Design with a Nontrivial Trait (Money)], page 175 for the included interface specification Money.

The type String<char> is specified in the file ‘AbstractString.h’ (see Section 7.1.1 [A First Class Design (Person)], page 168 for details.

The subtype, PlusAccount, is intended to have objects with both a savings and a checking account. These objects act like BankAccount objects with a credit that is the sum of their savings and checking accounts. By using specification variables, the owner and credit members of BankAccount are inherited by PlusAccount. Hence each PlusAccount object has both of these specification variables, in addition to the specification variables it declares itself.

As a programmer, your first instinct may be to only add one more specification variable to PlusAccount, and to reuse the inherited member, credit for either the savings or checking account balance. This would be reasonable in a program, as then it would save space, as instances would only have two Money variables, instead of three. However, remember that these are just specification variables, not real variables, and they take up no space in objects at all. Indeed, it would be inconsistent to treat the credit member differently than it was treated in BankAccount, which under this view is the sum of the savings and checking balances. So in the following specification, we use a common “trick”: the specification has three specification variables of type Money: the inherited credit, savings, and checking. We specify that credit depends on savings and checking and, by using an invariant, that its value is their sum.
The \textit{depends-clause} allows \texttt{savings} and \texttt{checking} to be modified whenever \texttt{credit} is mentioned in a \textit{modifies-clause}. Note that this does not work the other way around; for example in the extra case for \texttt{pay\_interest} below, just mentioning \texttt{checking} in the modifies clause is incorrect, as it would say that \texttt{credit} could not be changed. The \textit{represents-clause} says how \texttt{savings} and \texttt{checking} together represent \texttt{credit}.

```cpp
//@ spec Money savings;
//@ spec Money checking;
//@ depends credit on savings, checking;
//@ represents credit by credit\any = savings\any + checking\any;
//@ invariant savings\any >= 0 /\ checking\any >= 0;
//@ constraint checking^ = checking'
//@ for void deposit(Money amt);
```

```cpp
virtual ~PlusAccount() throw();
//@ behavior {
//@ ensures true;
//@ }
```

```cpp
virtual Money balance() const throw();
// balance spec inherited, but it might be reimplemented
```
virtual void pay_interest(double rate) throw();
//@ behavior {
//@ requires 0.0 <= rate \ rate <= 1.0;
//@ modifies credit, checking;
//@ ensures checking' = rational(1.0 + rate) * checking^;
//@ example checking^ = money(50000/100) \ rate = 0.05
//@ \ checking' = money(52500/100);
//@ // by inheritance, interest also added to savings
//@ }

// withdrawal takes money out of savings first, then checking
virtual void withdraw(Money amt) throw();
//@ behavior {
//@ requires 0 <= amt;
//@ {
//@ requires amt <= savings^;
//@ modifies credit, savings;
//@ ensures unchanged(checking);
//@ // by inheritance, amt is taken out of savings
//@ example savings^ = pennies(40001) \ amt = pennies(1)
//@ \ savings^ = pennies(40000)
//@ \ checking' = checking^;
//@ also
//@ requires savings^ < amt \amt <= (savings^ + checking^);
//@ modifies credit, savings, checking;
//@ ensures savings' = pennies(0)
//@ \ checking' = checking^ - (amt - savings^);
//@ example savings^ = pennies(40000) \ amt = pennies(52500)
//@ \ checking^ = pennies(60000)
//@ \ savings' = pennies(0)
//@ \ checking' = pennies(47500);
//@ }
//@ }

virtual void checking_deposit(Money amt) throw();
//@ behavior {
//@ requires amt >= 0;
//@ modifies credit, checking;
//@ ensures checking' = checking^ + amt \ unchanged(savings);
//@ }

The interface specification of the subtype \texttt{PlusAccount} inherits the specification of \texttt{balance} without respecification. So what should happen when \texttt{balance} is invoked on a \texttt{PlusAccount}? By specification inheritance, it must satisfy the specification of \texttt{balance} given for \texttt{BankAccount}. In this kind of example, one can visualize this idea by copying the specification given for \texttt{BankAccount} to the specification of \texttt{PlusAccount}. Hence \texttt{balance} must return the value of \texttt{credit}, which, by the invariant, is the total of the savings and checking accounts.

Applying specification inheritance to a virtual member function that is given its own specification is only a bit more complex. The idea is that the derived class’s operation must satisfy both specifications: the one specified for the base class and the one specified for the derived class (and both invariants and history constraints). A good way to visualize this is to consider both specifications as spec-cases of a single specification. For example, for \texttt{pay\_interest}, the specification for \texttt{PlusAccount} above gives one spec-case, and the other is given in the specification for \texttt{BankAccount}. The expanded form of this specification would look like the following.

```c++
// pay_check takes money out of checking first, takes from savings if needed
virtual void pay_check(Money amt) throw();
//@ behavior {
//@ requires 0 <= amt;
//@ {
//@ requires checking^ >= amt;
//@ modifies credit, checking;
//@ ensures checking' = checking^ - amt \ unchanged(savings);
//@ example checking^ = pennies(52501) \ amt = pennies(1)
//@ \ checking' = pennies(52500) \ unchanged(savings);
//@ also
//@ requires checking^ < amt \ amt <= (savings^ + checking^);
//@ modifies credit, checking, savings;
//@ ensures savings' = savings^ - (amt - checking^)
//@ \ checking' = 0;
//@ example savings^ = pennies(52500) \ checking^ = pennies(40000)
//@ \ amt = pennies(62500)
//@ \ savings' = pennies(40000) \ checking' = pennies(0);
//@ }
//@ }
};
#endif

The interface specification of the subtype \texttt{PlusAccount} inherits the specification of \texttt{balance} without respecification. So what should happen when \texttt{balance} is invoked on a \texttt{PlusAccount}? By specification inheritance, it must satisfy the specification of \texttt{balance} given for \texttt{BankAccount}. In this kind of example, one can visualize this idea by copying the specification given for \texttt{BankAccount} to the specification of \texttt{PlusAccount}. Hence \texttt{balance} must return the value of \texttt{credit}, which, by the invariant, is the total of the savings and checking accounts.

Applying specification inheritance to a virtual member function that is given its own specification is only a bit more complex. The idea is that the derived class’s operation must satisfy both specifications: the one specified for the base class and the one specified for the derived class (and both invariants and history constraints). A good way to visualize this is to consider both specifications as spec-cases of a single specification. For example, for \texttt{pay\_interest}, the specification for \texttt{PlusAccount} above gives one spec-case, and the other is given in the specification for \texttt{BankAccount}. The expanded form of this specification would look like the following.

```c++
// pay_check takes money out of checking first, takes from savings if needed
virtual void pay_check(Money amt) throw();
//@ behavior {
//@ requires 0 <= amt;
//@ {
//@ requires checking^ >= amt;
//@ modifies credit, checking;
//@ ensures checking' = checking^ - amt \ unchanged(savings);
//@ example checking^ = pennies(52501) \ amt = pennies(1)
//@ \ checking' = pennies(52500) \ unchanged(savings);
//@ also
//@ requires checking^ < amt \ amt <= (savings^ + checking^);
//@ modifies credit, checking, savings;
//@ ensures savings' = savings^ - (amt - checking^)
//@ \ checking' = 0;
//@ example savings^ = pennies(52500) \ checking^ = pennies(40000)
//@ \ amt = pennies(62500)
//@ \ savings' = pennies(40000) \ checking' = pennies(0);
//@ }
//@ }
};
#endif
extern void PlusAccount::pay_interest(double rate) throw();
//@ behavior {
//@ requires 0.0 <= rate /\ rate <= 1.0;
//@ modifies credit;
//@ ensures credit' = rational(1.0 + rate) * credit^;
//@ example rate = 0.05 /\ credit^ = money(400/1)
//@   /\ credit' = money(420/1);
//@ also
//@ requires 0.0 <= rate /\ rate <= 1.0;
//@ modifies credit, checking;
//@ ensures checking' = rational(1.0 + rate) * checking^;
//@ example checking^ = money(50000/100) /\ rate = 0.05
//@   /\ checking' = money(52500/100);
//@ }

Thus, in the specification of PlusAccount, it suffices to only say what happens to checking; what happens to savings can be inferred from the invariant and the specification given in BankAccount. Working this out, one sees that interest must be paid on both the savings and checking parts, as shown by the desugaring above. Thus the additional spec-case in PlusAccount keeps an implementation from transferring money between savings and checking when paying interest. (See Section 6.10 [Case Analysis], page 143 for the meaning of multiple specification cases.)

For deposit, the history constraint added in PlusAccount says that checking does not change. Putting this together with the specification given in BankAccount and the invariant, which says the credit is the sum of the savings and checking in any state, this means that the deposit goes entirely into savings.

The member function withdraw has an even more interesting specification; it inherits a case from BankAccount, and adds two more cases.

The invariant-clauses of public superclasses are also inherited and must be satisfied by the subclass. For example, PlusAccount must satisfy the invariant of BankAccount as well as its own invariant. In this example, one can visualize the inheritance of the invariant by copying it to the specification of the subtype.

Except for weak behavioral subtypes (see below), inheritance of history constraints is done in exactly the same way as for invariants. So for such specifications, the subtype must satisfy the history constraints of its supertypes and whatever history constraints are written in its specification.
7.9.2 Inheritance with Explicitly-Given Traits and Weak Subtyping

When the abstract values of a class are specified by using a LSL trait, instead of using specification variables, then the semantics of specification inheritance is a bit more complex than that described in the previous section. In this section we describe how to specify derived classes when either the derived class, or one of its public subclasses has a trait given explicitly. We also describe how to specify weak behavioral subtypes.

The key idea is that the user must specify how the abstract values of the derived class can be interpreted, in a given state, as abstract values of each of the relevant public base classes. When the derived class has its abstract values explicitly described by a trait, this must be done for each of its public base classes. When the derived class has its abstract values described by using specification variables, this must only be done for each of the public base classes that have their abstract values explicitly described by a trait.

The user tells Larch/C++ how the abstract values of a derived class can be interpreted as abstract values of a public base class by writing a `simulates-clause` in the derived class specification. The `simulates-clause` for a public base class, names a trait function that maps the sort of abstract values for the derived class to the abstract values of that base class. This trait function is called a simulation function (or coercion function [America87] [Liskov-Wing93] [Liskov-Wing93b]).

For simple cases, the signature of the simulation function is given by the signature of \( f \) in the following trait.

```%
% @(#$Id: SimulationFun.lsl,v 1.1 1995/01/08 00:46:23 leavens Exp $
SimulationFun(f, Derived, Base): trait
  introduces
  f: Derived -> Base
%
```

In the syntax that follows, the simulation function is named by a `fcnId`. By using the keyword `weakly`, the user tells Larch/C++ that a weak behavioral subtype is being defined, instead of a strong behavioral subtype. (When the simulation function is constructed automatically by Larch/C++, the by `fcnId` part of the syntax can be omitted to indicate a weak behavioral subtype.)

```
simulates-clause ::= [ weakly ] simulates supertype-list ;
supertype-list ::= supertype [ , supertype ] ... 
supertype ::= sort-name [ by fcnId ]
```
As an example, consider a type `MutableMoney` that is derived from the public base class `Money` (see Section 7.1.2 [A Design with a Nontrivial Trait (Money)], page 175). This example is adapted from Dhara’s Ph.D. dissertation [Dhara97], which used it, as we will here, to explain the idea of weak behavioral subtyping. In this example an instance of `MutableMoney` will act like a `Money` object with the same amount of pennies. However, unlike `Money`, whose objects are immutable, `MutableMoney` will also have mutators. Hence `MutableMoney` will only be a weak behavioral subtype of `Money`. This means that a `MutableMoney` object will only act like a `Money` object when manipulated by the member functions of the type `Money`, assuming that there is no way to simultaneously mutate it (e.g., by an alias). (Exact conditions for on aliasing and how to reason about programs using weak behavioral subtyping a matter of current research, see [Dhara97] for details.)

As with the class `Money`, we specify the abstract values of `MutableMoney` explicitly. This is done in the following trait, by including the trait `Money(MutableMoney for Money)`. The trait `Money` is also included, thus overloading the constructors such as `pennies`. The new operation introduced is `toMoney`, which tells how to take a `MutableMoney` value and coerce it to a `Money` value. (The trait also includes axioms to ensure that the values of the constants `MONEY_MAX` and `MONEY_MIN` are the same for `MutableMoney` as they are for `Money`.)

```plaintext
% @(#)Id: MutableMoneyTrait.lsl,v 1.3 1997/07/31 02:41:11 leavens Exp $

MutableMoneyTrait: trait
  includes MoneyTrait, MoneyTrait(MutableMoney for Money)
  introduces
    toMoney: MutableMoney -> Money
  asserts
    \forall p: long
      toMoney(MONEY_MAX) == MONEY_MAX:Money;
      toMoney(MONEY_MIN) == MONEY_MIN:Money;
      toMoney(pennies(p)) == pennies(p);
  implies
    MutableMoneyHom
```

The following is the interface specification of the derived class, `MutableMoney`.

```plaintext
// @(#)Id: MutableMoney.lh,v 1.3 1997/07/30 20:03:02 leavens Exp $

#ifndef MutableMoney_lh
#define MutableMoney_lh

//@ uses MutableMoneyTrait;
```
/@ uses NoContainedObjects(MutableMoney);

class MutableMoney : public Money {
public:
  //@ weakly simulates Money by toMoney;

  MutableMoney(double amt) throw();
  //@ behavior {
  //  //@ requires inRange(money(rational(amt)));
  //  //@ constructs self;
  //  //@ ensures self' = money(rational(amt));
  //  //@ }

  MutableMoney(long int cts = 0L) throw();
  //@ behavior {
  //  //@ requires inRange(pennies(cts));
  //  //@ constructs self;
  //  //@ ensures self' = pennies(cts);
  //  //@ }

  virtual void AddIn(const Money & m2) throw();
  //@ behavior {
  //  //@ requires assigned(m2, pre) /
  //  //@ inRange(self^ + m2\any);
  //  //@ modifies self;
  //  //@ ensures liberally self' + m2\any;
  //  //@ example liberally self^ = money(300/1) /
  //  //@ m2\any = money(400/1)
  //  //@ /
  //  //@ self' = money(700/1);
  //  //@ }

  virtual void SubtractIn(const Money & m2) throw();
  //@ behavior {
  //  //@ requires assigned(m2, pre) /
  //  //@ inRange(self^ + m2\any);
  //  //@ modifies self;
  //  //@ ensures liberally self' - m2\any;
  //  //@ example liberally self^ = money(700/1) /
  //  //@ m2\any = money(400/1)
  //  //@ /
  //  //@ self' = money(300/1);
  //  //@ }

  virtual void MultiplyIn(double factor) throw();
  //@ behavior {
  //  //@ requires inRange(rational(factor) * self^);
  //  //@ ensures liberally self' = rational(factor) * self^;
  //  //@ }
};

#endif
Notice the weakly simulates clause in the above specification. It says that \texttt{MutableMoney} is a weak behavioral subtype of \texttt{Money}. It also says what abstract value each \texttt{MutableMoney} value simulates. That is, a \texttt{MutableMoney} value \( mm \) simulates the \texttt{Money} value \( \text{toMoney}(mm) \).

The simulation function should have the property that, it commutes with all the trait functions that take the supertype as an argument. This property is asserted in the following trait, which is implied by the trait \texttt{MutableMoney} given above. Technically, this property makes the simulation function a homomorphism on the abstract values. Hence the property is called the \textit{homomorphism property}.

\begin{verbatim}
% @(#)Id: MutableMoneyHom.lsl,v 1.2 1997/07/31 20:59:42 leavens Exp $

\texttt{MutableMoneyHom: trait}
\texttt{\qquad assumes MutableMoneyTrait}
\texttt{\qquad asserts}
\texttt{\qquad \forall mm, mm1, mm2: MutableMoney, q: Q}
\texttt{\qquad dollars(mm) == dollars(toMoney(mm));}
\texttt{\qquad cents(mm) == cents(toMoney(mm));}
\texttt{\qquad toMoney(money(q)) == money(q);}
\texttt{\qquad toMoney(mm1 + mm2) == toMoney(mm1) + toMoney(mm2);}
\texttt{\qquad toMoney(mm1 - mm2) == toMoney(mm1) - toMoney(mm2);}
\texttt{\qquad toMoney(q * mm) == q * toMoney(mm);} 
\texttt{\qquad (mm1 > mm2) == toMoney(mm1) > toMoney(mm2);}
\texttt{\qquad (mm1 < mm2) == toMoney(mm1) < toMoney(mm2);}
\texttt{\qquad (mm1 >= mm2) == toMoney(mm1) >= toMoney(mm2);}
\texttt{\qquad (mm1 <= mm2) == toMoney(mm1) <= toMoney(mm2);}
\texttt{\qquad inRange(mm) == inRange(toMoney(mm));}
\end{verbatim}

One way to view the homomorphism property is that it allows Larch/C++ to define all the trait functions that take abstract values of the supertype, making these trait functions applicable to the abstract values of the subtype. The definitions of such overloaded trait functions would be like those given in the trait \texttt{MutableMoneyHom}.

Because the trait functions that take abstract values of the supertype can be thought of as defined on the abstract values of the subtype, any specification of a function taking arguments of the supertype can be interpreted in a standard way for arguments of the subtype. Indeed, any specification need only mention one type, and is automatically valid for all subtypes of that type. This property is called \textit{modular specification} in [Leavens90].

Modular specification at the interface level is achieved by the use of supertype abstraction. In practicing supertype abstraction, the reader of a specification thinks in terms of the abstract values of the types written (statically) in the specification. If subtyping is being used, the reader’s view
corresponds to using the simulation function to map the abstract values of the subtype up to the
type used in the specification.

As in the previous subsection, the ability to interpret the supertype’s specifications for abstract
values of the subtype means that one can inherit the supertype’s specifications. As before, inher-
heritance of specifications can be thought of as including the supertype’s specification as additional spec-cases, added to those given for the member function in the subtype. (This does not apply
non-virtual function members, including constructors, as these are not inherited. However, it does
apply to virtual destructors, which are inherited.) The inclusion of each spec-case from the cor-
responding trait function of a supertype in the subtype’s specification body makes the subtype’s
function satisfy that supertype’s specification. The subtype need not have anything specified for it,
in which case the meaning is determined completely by the specifications of its supertypes. If the
subtype specification has additional spec-cases, these must be satisfied in addition to those of each
supertype. That is, when there are multiple supertypes with the same virtual member function (of
the same argument types), then all of the specifications must be satisfied. (Sometimes this may
lead to a specification that cannot be implemented.)

The only difference from the previous section, where specification variables were used to define
the abstract values instead of explicit traits, is that the specifications of the supertype cannot be
literally copied to the subtype, but must use the simulation function to avoid type errors. This
difference, is mostly a technical detail, and so if you are not interested in such details, you may
want to skim the rest of this subsection lightly.

The basic technical idea is to use the simulation function on each subterm that would have the
subtype’s abstract value in the specification cases, invariants, and history constraints copied from
the supertype. Usually this boils down to, for example, changing self to toMoney(self), self’
to toMoney(self’), and self\any to toMoney(self\any).

The other important technical point is that when copying the history constraint for a weak
behavioral subtype, the history constraint is only applied to the public virtual member functions
of the weak behavioral supertype [Dhara-Leavens96]. For example, since MutableMoney is only
a weak behavioral subtype of Money, the history constraint specified for Money is not applied to
the new member functions AddIn, SubtractIn, and MultiplyIn. This allows these new member
functions to be mutators.

To show this in detail, the following is an equivalent specification of MutableMoney, which shows
the effect of specification inheritance. Notice in particular the inherited history constraint.

```plaintext
#ifndef MutableMoney2_h
#define MutableMoney2_h

#include "Money.h"

//@ uses MutableMoneyTrait;
//@ uses NoContainedObjects(MutableMoney);

class MutableMoney : public Money {
public:
    //@ weakly simulates Money by toMoney;

    // inherited invariant
    //@ invariant inRange(toMoney(self\any));

    // inherited constraint
    //@ constraint toMoney(self') = toMoney(self^)
    //@ for virtual long int Dollars() const throw(),
    //@ virtual long int Cents() const throw(),
    //@ virtual Money & operator + (const Money & m2) const throw(),
    //@ virtual Money & operator - (const Money & m2) const throw(),
    //@ virtual Money & operator * (double factor) const throw(),
    //@ virtual bool operator == (const Money & m2) const throw(),
    //@ virtual bool operator > (const Money & m2) const throw(),
    //@ virtual bool operator >= (const Money & m2) const throw(),
    //@ virtual bool operator < (const Money & m2) const throw(),
    //@ virtual bool operator <= (const Money & m2) const throw();

    MutableMoney(double amt) throw();
    //@ behavior {
    //@ requires inRange(money(rational(amt)));
    //@ constructs self;
    //@ ensures self' = money(rational(amt));
    //@ }

    MutableMoney(long int cts = 0L) throw();
    //@ behavior {
    //@ requires inRange(pennies(cts));
    //@ constructs self;
    //@ ensures self' = pennies(cts);
    //@ }

    // inherited virtual member function specifications follow

    virtual ~Money() throw();
    //@ behavior {
    //@ ensures true;
    //@ }

}
//@ spec virtual long int Dollars() const throw();
//@ behavior {
//@   ensures result = dollars(toMoney(self\any));
//@ }

//@ spec virtual long int Cents() const throw();
//@ behavior {
//@   ensures result = cents(toMoney(self\any));
//@ }

//@ spec virtual Money & operator + (const Money & m2) const throw();
//@ behavior {
//@   requires assigned(m2, pre) \ inRange(toMoney(self\any) + m2\any);
//@   ensures returns;
//@   also
//@   requires assigned(m2, pre);
//@   ensures liberally fresh(result) \ assigned(result, post)
//@     \ toMoney(result') = toMoney(self\any) + m2\any;
//@   example liberally toMoney(self\any) = money(300/1)
//@     \ m2\any = money(400/1)
//@     \ toMoney(result') = money(700/1)
//@     \ fresh(result) \ assigned(result, post);
//@ }

//@ spec virtual Money & operator - (const Money & m2) const throw();
//@ behavior {
//@   requires assigned(m2, pre) \ inRange(toMoney(self\any) - m2\any);
//@   ensures returns;
//@   also
//@   requires assigned(m2, pre);
//@   ensures liberally fresh(result) \ assigned(result, post)
//@     \ toMoney(result') = toMoney(self\any) - m2\any;
//@ }

//@ spec virtual Money & operator * (double factor) const throw();
//@ behavior {
//@   requires inRange(rational(factor) * toMoney(self\any));
//@   ensures returns;
//@   also
//@   ensures liberally fresh(result) \ assigned(result, post)
//@     \ toMoney(result') = rational(factor) * toMoney(self\any);
//@ }

//@ spec virtual bool operator == (const Money & m2) const throw();
//@ behavior {
//@ requires assigned(m2, pre);
//@ ensures result = (toMoney(self\any) = toMoney(m2\any));
//@ ensures redundantly result =
//@ (dollars(toMoney(self\any)) = dollars(m2\any)
//@ /
//@ cents(toMoney(self\any)) = cents(m2\any));
//@ }

//@ spec virtual bool operator > (const Money & m2) const throw();
//@ behavior {
//@ requires assigned(m2, pre);
//@ ensures result = (toMoney(self\any) > m2\any);
//@ }

//@ spec virtual bool operator >= (const Money & m2) const throw();
//@ behavior {
//@ requires assigned(m2, pre);
//@ ensures result = (toMoney(self\any) >= m2\any);
//@ }

//@ spec virtual bool operator < (const Money & m2) const throw();
//@ behavior {
//@ requires assigned(m2, pre);
//@ ensures result = (toMoney(self\any) < m2\any);
//@ }

//@ spec virtual bool operator <= (const Money & m2) const throw();
//@ behavior {
//@ requires assigned(m2, pre);
//@ ensures result = (toMoney(self\any) <= m2\any);
//@ }

// end of inherited specs

virtual void AddIn(const Money & m2) throw();
//@ behavior {
//@ requires assigned(m2, pre) /
//@ inRange(self\any + m2\any);
//@ modifies self;
//@ ensures liberally self’ + m2\any;
//@ example liberally self\any = money(300/1) /
//@ m2\any = money(400/1)
//@ /
//@ self’ = money(700/1);
//@ }

virtual void SubtractIn(const Money & m2) throw();
//@ behavior {
//@ requires assigned(m2, pre) \ inRange(self^ + m2\any);
//@ modifies self;
//@ ensures liberally self' = m2\any;
//@ example liberally self^ = money(700/1) \ m2\any = money(400/1)
//@ \ self' = money(300/1);
//@ }

virtual void MultiplyIn(double factor) throw();
//@ behavior {
//@ requires inRange(rational(factor) * self^);
//@ ensures liberally self' = rational(factor) * self^;
//@ }

#endif

To summarize, when a specification of a member function of a class Base is inherited in a class Derived, the sort of self changes from Obj[Base] to Obj[Derived]. Hence the sort of self^ is Derived, as is the sort of self'. To avoid the sort errors that would occur when self^ and self' (and their equivalent forms) are passed to trait functions that expect an argument of the supertype’s sort, a call to the simulation function is wrapped around such subterms. (See our papers on this subject [Dhara-Leavens96], for more details.)

Using such a rewriting of specification inheritance allows us to think of specification inheritance as a (very convenient) syntactic sugar.

7.9.3 More Details of Specification Inheritance

In this subsection we treat a few more details about specification inheritance.

[[[Need to add something about avoiding capture in the desugaring.]]]

7.9.3.1 Strong vs. Weak Behavioral Subtyping

The distinction between strong [Liskov-Wing93b] [Liskov-Wing94] and weak [Dhara-Leavens94b] behavioral subtyping only affects the inheritance of history constraints. Recall that a weak behavioral subtype is specified by using weakly in a simulates-clause that names the supertype. If this is not done, then strong behavioral subtyping is assumed.
For a strong behavioral subtype, all of the subtype’s member functions must satisfy the supertype’s history constraint; that is, unless `weakly` is used, the history constraints are inherited unchanged by the subtype and apply to all member functions in the subtype, even new ones. Because of this, instances of the subtype can be manipulated through aliases using the interface of both its supertype and its own type [Liskov-Wing93b] [Liskov-Wing94].

In a weak behavioral subtype, the history constraint is only applied to the member functions inherited from the supertype. See Section 7.9.2 [Inheritance with Explicitly-Given Traits and Weak Subtyping], page 223 above for an example.

### 7.9.3.2 Simulation Functions that Need a State

Some simulation functions need a state, to access the values of contained objects. Thus a simulation function is also allowed to have the signature of `fs` in the following trait. See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for an example of a simulation function that needs a state.

```%
% @(#)Id: SimulationFunState.lsl,v 1.3 1995/01/14 04:07:23 leavens Exp $
SimulationFunState(fs, Derived, Base): trait
  assumes State
  introduces
    fs: Derived, State -> Base
```

We can regard the definition of a simulation function of the first signature as sugar for the definition of a simulation function with the second signature, by use of the following trait.

```%
% @(#)Id: SimFunStateFromFun.lsl,v 1.2 1995/01/14 06:36:57 leavens Exp $
SimFunStateFromFun(f, fs, Derived, Base): trait
  includes SimulationFun, SimulationFunState(fs, Derived, Base)
  asserts \forall d: Derived, st: State
    fs(d, st) = f(d)
```

For example, in the specification of `MutableMoney` above, the trait function `toMoney` has the signature `MutableMoney -> Money`. Therefore, Larch/C++ implicitly uses the following trait, to overload `toMoney` with the signature `MutableMoney, State -> Money`.

```SimFunStateFromFun(toMoney, toMoney, MutableMoney, Money)```
The semantics of specification inheritance with such a simulation function is given as a syntactic sugar, by rewriting the spec-cases of the supertype using the simulation function, and adding them to the explicitly written spec-cases. The only trick is to get a sensible state passed, although this does not matter for simulation functions like toMoney, which do not depend on a state.

The rewriting goes as follows. Suppose \( g \) is a trait function defined on the supertype, Base. Suppose \( fs \) is the simulation function of signature \( \text{Derived, State} \rightarrow \text{Base} \). Then the term \( g(self^\_\text{ }) \) (and its equivalents) rewrites to \( g(fs(self^\_, \text{pre})) \). Similarly the term \( g(self') \) rewrites to \( g(fs(self', \text{post})) \). The same applies to trait functions that take more than one argument, and to trait functions; for example, \( h(self', 2) \) rewrites to \( h(fs(self', \text{post}), 2) \). When the state-fcn is any, the state used is any. For example, the terms \( h(self\_\text{any}, 2) \) rewrites to \( h(fs(self\_\text{any}, any), 2) \). Note that values of sort Obj[Derived] are passed to trait functions unchanged.

For example, when inheriting a specification from Money, a term of the form \( self' \) is rewritten to toMoney(self', post). See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for a full example using such a simulation function.

### 7.9.4 Related Work on Inheritance of Specifications

The Larch/C++ semantics for specification inheritance resembles that in Fresco [Wills92b], in that both impose the specifications of member functions on subtypes. However, Fresco does not have as general a notion as the simulation functions in Larch/C++, because all abstract values in Fresco look like tuples. In Leino’s thesis [Leino95], methods are given a single specification which must be satisfied by implementations of subtypes. All specifications of objects are given using specification variables, so, as in Fresco, all abstract values support the same abstract fields, thus there is no problem in interpreting such specifications for subtype objects. A similar kind of specification inheritance is also found in LM3 [Jones91] (and Chapter 6 of [Guttag-Horning93]) and in Eiffel [Meyer92b], but neither LM3 nor Eiffel forces the subtype to be behavioral. That is, in LM3 and in Eiffel, the programmer can choose not to inherit specifications.

The semantics of weak behavioral subtyping, and simulation functions that take state into account are new with this work (January 1995). See our other work [Dhara-Leavens96] for an extended account.
7.10 Abstract Classes

In C++, a class is an abstract class if it has any pure virtual member functions. In C++, a member
function is declared as a pure virtual function by writing = 0 instead of the usual function body
(see Section 10.3 of [Ellis-Stroustrup90]). Because pure virtual functions have no implementation,
no objects of such a class can be constructed. Hence such a class can only be used as a base class
for other classes. A class is concrete if it has no pure virtual member functions. When a class
is derived from a class with pure virtual member functions, it will only be concrete if each pure
virtual function member is overridden instead of inherited (see Section 10.2 [Ellis-Stroustrup90]).

In Larch/C++, a member function specified in a class can only be implemented by a pure virtual
function if its interface is specified as a pure virtual function. In addition, a class can be specified
to be an abstract class by prefixing its specification with the keyword abstract. This tells the
writer of a derived class that the pure virtual member functions must be implemented to obtain a
concrete class.

In Larch/C++ one may also declare a class to be abstract without specifying any pure virtual
member functions. This tells the reader that the no objects of the class should be created. In
an abstract class, there are no implicitly generated specifications (see Section 7.2.3 [Implicitly
Generated Member Specifications], page 191).

An example of an abstract class specification is the following specification of IntList. It uses
the trait List from the LSL Handbook (page 173 [Guttag-Horning93]). Instead of defining a trait,
simply to include NoContainedObjects, that trait is used directly (see Section 7.5 [Contained
Objects], page 207 for details on why this is done). All the member functions are in this specification
are pure virtual, except for length. This is because length can be programmed with the others,
and so the specification states that it must be implemented normally. Note that even for pure
virtual members a specification is given.

```cpp
//@ abstract @*/
```
virtual int head() const throw() = 0;
//@ behavior {
//@ requires ~(self\any = empty);
//@ ensures result = head(self\any);
//@ }

virtual IntList* tail() const throw() = 0;
//@ behavior {
//@ requires ~(self\any = empty);
//@ ensures isValid(result) /
// (result)' = tail(self\any);
//@ }

virtual int length() const throw();
//@ behavior {
//@ requires len(self\any) <= INT_MAX;
//@ ensures result = len(self\any);
//@ }
};

Other useful features for specifying abstract classes are invariants (see Section 7.3 [Class Invariants], page 196) and history constraints (see Section 7.4 [History Constraints], page 200).

7.11 Specifying Exposed Data Members

In C++, nothing stops you from having public data members in a class. Indeed, as typically used, a C++ struct is like a class with public data members and few, if any, member functions. However, since one of the main reasons for using classes is to hide how data is represented, one should think twice before specifying public data members in a class. Nevertheless, there may be situations in public, or as we will call them in this section, exposed data members are useful.

To take a simple example, suppose we are writing a programming language interpreter, and need a class to represent integer variables (in the language being interpreted). Objects of this class will have a name and an integer variable cell. To avoid duplicating the functionality of C++ integer variables (and to provide an example for this section), suppose we choose to expose the integer variable in the implementation. That is, suppose we decide that the C++ class implementing this type will look like the following.
class IntVar {
public:
    int var;
    // ...
};

This is very easy to specify in this form, since as the exposed data member serves the same role as a specification variable would normally. Thus, there is no problem, as long as you are willing to allow Larch/C++ to automatically construct a trait for the abstract model. (If you want to build your own trait, you can model it on the ones automatically constructed by Larch/C++. See Section 11.10 [Structure and Class Types], page 299 for details.)

The interface specification given below has only a constructor (with two default parameters) and a member function name. (Although Larch/C++ considers string literals to be arrays of characters, a pointer argument such as this can be initialized from such an array, and so the specification does not have a sort error.)

    // @(#)$Id: IntVar.lh,v 1.14 1997/07/31 03:10:16 leavens Exp $
#include "AbstractString.lh"
//@ uses cpp_const_char_string;

class IntVar {
public:
    int var;
   //@ spec String<char> its_name;
   //@ constraint its_name^ = its_name'; // the name can’t be changed

    IntVar(const char vname[] = "unnamed", int initial_value = 0) throw();
    //@ behavior {
    //@ requires nullTerminated(vname, any);
    //@ constructs var, its_name;
    //@ ensures its_name’ = throughNull(vname, any)
    //@ \ var’ = initial_value;
    //@ }

    const char * name() const throw();
    //@ behavior {
    //@ ensures nullTerminated(result, post)
    //@ \ throughNull(result, post) = its_name\any;
    //@ }
};
A final note about this example. Just because it is necessary to expose one data member, does not mean that one needs to expose them all. In the \texttt{IntVar} example, the data member \texttt{its\_name} is a specification variable, and hence cannot be used by C++ clients.

### 7.12 Specifying Friends

Larch/C++ can also specify friendship grants, which record information that may be needed in an implementation. Friendship grants, as in C++, grant access to the private interface of a class implementation to particular functions or to all the member function in some class (see section 11.4 of [Ellis-Stroustrup90]).

For Larch/C++, a friendship grant in a class specification also grants accessibility to the private and protected members in the specification of the functions granted access (see Section 2.3 [Accessibility of Class Members in Specifications], page 16).

A class specification may specify a friendship grant with a \texttt{member-declarator} or a \texttt{member-declaration}, starting with the \texttt{decl-specifier friend}. The syntax is the same as in C++. See Section 7.2 [Class Member Specifications], page 186 for the syntax of \texttt{member-declarator} and \texttt{member-declaration}. See Section 5.2 [Declaration Specifiers], page 57 for the syntax of \texttt{decl-specifier}.

For example, adding the following to the specification of \texttt{Money} (see Section 7.1.2 [A Design with a Nontrivial Trait (Money)], page 175) would grant friendship to operator \texttt{**=} with parameters of types \texttt{Money&} and \texttt{double}.

\begin{verbatim}
friend Money& operator **= (Money& p, double scalar);
\end{verbatim}

Since friendship grants are not themselves affected by their placement in the \texttt{public}, \texttt{private}, or \texttt{protected} sections of a specification, they may appear anywhere a \texttt{member-declarator} may appear (see Chapter 7 [Class Specifications], page 167 for the syntax).

The specification of a function granted friendship cannot appear inside the class. For example, the specification of the overload of \texttt{**=} for \texttt{Money&} and \texttt{double} parameters could not be given in the class \texttt{Money}. Instead, it would appear elsewhere. For example, it could be given in a separate specification module, such as the following.
An implementation of a Larch/C++ specification may have more friendship grants than appear in the specification. This is because a friendship grant records a C++ design decision (see Section 10.3 [Specifying Protected and Private Interfaces], page 263 for more discussion on this point). This means that you may specify a function which might be granted friendship in some implementation, without giving the friendship grant in the specification. For example, one might specify *= as above, and implement it as a friend function by granting it friendship in the code, all without writing the friendship grant in the specification of Money. Thus the friendship grant is only needed in the specification if one wishes to explicitly record a detailed design decision.

Although Larch/C++ fully supports the granting of friendship as a way to record detailed design decisions, friendship grants are necessary far less often than some C++ programmers think. This is especially the case for overloads of << for output. As a general rule, you should try to specify enough member functions so that << can be programmed without access to private data members, and thus without a friendship grant. When specifying for clients (as opposed to recording detailed design), you should only put a friendship grant in the Larch/C++ specification when it is clear that the friendship grant will be needed in every conceivable implementation.
A C++ template specifies a family of related classes or functions (see Chapter 8 [Stroustrup91]). One way to think of a template is as a fancy macro for generating classes or functions. For example, in C++ one can have a template class `Set`, and use it to generate the types `Stack<int>`, `Stack<float>`, `Stack<Stack<String>>`, and so on. The types `int`, `float`, and `Stack<String>` in this example are called the template arguments. The types `Stack<int>` and so on are said to be instantiations of the template `Set`. The template `Stack` is specified in terms of a formal (i.e., dummy) argument type.

In Larch/C++ templates usually also take traits as arguments [Edwards-etal94] [Ernst-etal91] [Goguen84]. Hence the above instantiations would look like `Stack<int /*@ using int @*/>`, `Stack<float /*@ using float @*/>`, etc.

Larch/C++ also includes a way to specify the interface and behavior required of a type parameter. This is done using a where-seq.

The syntax of template-declarations is very the same as the proposed C++ standard [Stroustrup95], except that Larch/C++ adds to the C++ syntax. One addition is the expects-clause, which is used to specify trait parameters. Another is the where-seq, which specifies the required capabilities of type parameters. See Chapter 5 [Declarations], page 54 for the syntax of init-declarator. See Section 6.13 [Specifying Higher-Order Functions], page 157 for the syntax of expects-clause.

The declaration should declare or specify a class or function; that is only class or function template specification are allowed (see Section r.14 of [Stroustrup91]).

There are two kinds of template-parameters. The most common is a type-parameter, which usually has the form `class identifier`. In this form, the `identifier` is treated as a type name in the
body of the template class or function being declared. The C++ syntax uses `class` for declaring such formal type parameters, but that does not mean that the actual template argument has to be the type of a class: it can be any type that satisfies the requirements placed on the `identifier` in the `where-seq`.

A template can also be declared as a template argument. This is new syntax introduced in the coming C++ standard [Stroustrup95] [ANSI95].

The second form of a `template-parameter` allows constant expressions to be passed to templates (see Chapter 8 [Stroustrup91]). See Section 5.2 [Declaration Specifiers], page 57 for the syntax of `decl-specifier`. See Section 5.4 [Declarators], page 65 for the syntax of a `declarator`. See Section 5.1 [Initializers], page 56 for the syntax of an `initializer`.

### 8.1 Example Template without Requirements

The template class `Stack` specified below is an example of the specification of a template class. This example is simple in the sense that it has minimal expectations of its type parameter. It does, however, illustrate some of the subtle features of working with contained objects in a template. The `expects-clause` below says that the trait `contained_objects(T)` (see Section 7.5 [Contained Objects], page 207) must be passed as a trait actual parameter when this template is instantiated. A discussion on how this expected trait is used to define `contained_objects` for instances of the `Stack` template is found after the example.

```cpp

template <class T /*@ expects contained_objects(T) @*/ >
class Stack {
public:
    //@ uses SimpleStackTrait(T for E, Stack<T> for C);

    Stack() throw();
    //@ behavior {
    //@    constructs self;
    //@    ensures liberally self’ = empty;
    //@ } 

    void push(T e) throw();
    //@ behavior {
    //@    modifies self;
    //@    ensures liberally self’ = push(e, self`);
    //@ }
```
T pop() throw();
//@ behavior {
//@   modifies self;
//@   ensures self' = pop(self');
//@ }

T top() const throw();
//@ behavior {
//@   requires ~isEmpty(self');
//@   ensures result = top(self');
//@ }

};

An instantiation of SimpleStack, such as the following

SimpleStack<int& /*@ using MutableObj(int) @*/>

must pass a trait that satisfies the specification of the expected trait, when the actual parameters are substituted for the formals. In this case, the LSL sort for int& is Obj[int], and thus the trait expected is contained_objects(Obj[int]). The actual trait passed, MutableObj(int) satisfies the theory of this expected trait. See Section 8.3 [Instantiation of Templates], page 251 for more about instantiations.

Note that the uses-clause for this example is, contrary to our usual practice to this point, found inside the class definition. This is because the trait used depends on the particular type parameter. Trying to put the uses clause outside the template definition would result in an error, because then the type parameter T would not be visible as a type. However, because the trait used depends on the type parameter, clients that instantiate Stack will themselves have to use an appropriate trait. (See Section 8.3 [Instantiation of Templates], page 251 for an example.)

The trait for SimpleStack is given below. It simply uses the trait Stack from Guttag and Horn-ing’s LSL Handbook (page 170 of [Guttag-Horning93]), and specifies the trait function contained_objects. It includes the trait PureValue(C) to specify that eval on stack values should be the identity function (see Section 6.2.1 [State Functions], page 103), and that the potential subobjects within a stack should not affect the implicit preconditions for member functions (see Section 6.2.2 [Allocated and Assigned], page 107). See Section 7.5 [Contained Objects], page 207 for the trait PureValue.
The trait *container_objs* is often useful in specifying templates that are containers. (In this context, a container is an object whose abstract values satisfy the LSL Handbook trait *Container*, see page 177 of [Guttag-Horning93].) It defines the contained objects of a stack to be the contained objects of the elements of the stack. For example, if the actual template parameter used for T is an object type, such as `int&`, then all of those elements of the stack will be contained objects. Similarly, if the actual template parameter used for T is an pointer type, such as `int *`, then the objects pointed at by the elements of the stack will be the contained objects.

If one has a sort of abstract values with a membership test, but nothing like the trait functions *head* and *tail*, then one can use the following weaker trait instead of *container_objs*. 
with_member_objs(E,C): trait
    includes contained_objects(C)
    assumes HasMembership(E,C), contained_objects(E)
    asserts
    \forall e: E, c: C, st: State
    (e \in c) => (contained_objects(e,st) \subseteqq contained_objects(c, st))

The trait HasMembership assumed by the above trait is given below.

HasMembership(E,C): trait
    introduces
    __ \in __ : E, C -> Bool

8.2 Requirements on Template Parameters

The syntax of the where-seq is as follows.

where-seq ::= where-clause [ where-clause ] ...
where-clause ::= where type-arg-name is complete-class-name ;
    | where type-arg-name is where-body ;
where-body ::= [ class | { member-seq } }

type-arg-name ::= identifier | original-class-name | template-class-instance

Each where-clause in a where-seq constrains the type-arg-name following the keyword where. Each where-clause in a where-seq should constrain a different type-arg-name, and all the type-arg-
names must be a type parameters. These can be either identifiers following the keyword class in
type-parameter or an instantiation of a class template declared as a type-parameter.
Not every type parameter needs to be constrained in a where-clause; leaving a type parameter unconstrained means that any actual type (or template) argument is acceptable. For example, the template `Stack` above (see Section 8.1 [Example Template without Requirements], page 240) can be instantiated with a type that is not a class, such as `int`, as well as with class types.

There are three kinds of constraints that a where-clause can place on a type argument. First two use the simplest syntactic form, which involves the keyword `is`. When the complete-class-name following `is` names a class, as opposed to a signature, then the actual argument must be a subtype of the given class name. When the complete-class-name names a signature instead, then the actual parameter must conform to the signature given.

To illustrate the use of subtyping, consider the following example. In it, `Expr` is a template class, and the template function `eval` can work with any actual type `SI` that is a subtype of the following.

\[
\text{SimpleStack}\langle\text{int using NoContainedObjects(int)}\rangle
\]

That is one can instantiate `eval` as follows

\[
eval\langle\text{Stack}\langle\text{int /}@\text{ using NoContainedObjects(int)}\rangle /@\rangle
\]

or as follows

\[
eval\langle\text{FastStack}\langle\text{int /}@\text{ using NoContainedObjects(int)}\rangle /@\rangle
\]

as long as `FastStack` is publicly derived from `Stack`. This allows the function `eval` to treat `SI` as if it were exactly

\[
\text{SimpleStack}\langle\text{int using NoContainedObjects(int)}\rangle
\]

in its body. Thus one has subtype polymorphism [Leavens-Weihl90] [Leavens91] [Leavens-Weihl95].
The trait Eval_Trait used in this example is just a stub. A more interesting specification would be needed in reality.

If a signature is used instead of a class following is, then the where-clause specifies that the actual template argument must have a specification that conforms to the given specification. For the types, this means that the actual parameter has all the operations of the given signature, and that the types correspond [Baumgartner-Russo95]. The specification of each operation in the actual parameter must also satisfy the given specification. [[Need to lay this out in detail]]

For example, suppose we have the following signature defined. This signature describes a type with an == operator defined on it. All elements of a signature are implicitly public, so there is no need to use the public: access specifier. The expected trait must satisfy the speciﬁca-
tion `Equality(Elem)`, which the actual parameter trait must define the meaning of `. The trait `Equality` is from Guttag and Horning’s LSL handbook (p. 193 of [Guttag-Horning93]).

```cpp

template <class Elem /*@ expects Equality(Elem) @*/>
signature Equivalence {
//@ bool operator ==(Elem x, Elem y);
//@ behavior {
//@ ensures returns /
 result = (x = y);
//@ }
//@ }

Then we can use the above signature to describe the requirements on the template parameter for a set template as follows. The set template has an expected trait, `SimpleSetRequirement(Elem)`. Note that there is a formal parameter name for this trait, `ElemTrait`, since the actual parameter has to be passed to the instantiation of the template signature `Equivalence`. More discussion of this set template follows.

```cpp
// @(#)$Id: SimpleSet2.lh,v 1.3 1998/09/24 16:36:45 leavens Exp $

#include "Equivalence.lh"

template <class Elem /*@ expects SimpleSetRequirement(Elem) ElemTrait @*/>
//@ where Elem is Equivalence<Elem using *ElemTrait>

class Set {
public:
//@ uses SimpleSetTrait(Elem for E, Set<Elem> for C);

Set() throw();
//@ behavior {
//@ constructs self;
//@ ensures liberally self’ = empty;
//@ }

void insert(Elem e) throw();
//@ behavior {
//@ modifies self;
//@ ensures liberally self’ = self^ \U {e};
//@ }

bool is_in(Elem e) const throw();
//@ behavior {
//@ ensures result = (e \in self);
//@ }

};
```
In the \texttt{Set} template’s \textit{where-clause}, the requirement is that \texttt{Elem} satisfy the following signature.

\begin{verbatim}
Equivalence<Elem using ElemTrait>
\end{verbatim}

By the definition of \texttt{Equivalence}, this means that the type argument \texttt{Elem} must have the C++ operator == defined on it; furthermore == must have the expected behavior. One can thus instantiate \texttt{Set} with types such as \texttt{int} by writing \texttt{Set<int /*@ using int @*/}, where the second \texttt{int} is the name of the trait for the C++ type \texttt{int} (see Section 11.1.5 [int Trait], page 272). This makes the trait \texttt{int} the actual parameter that must satisfy the required trait \texttt{SimpleSetRequirement}, which is given below.

\begin{verbatim}
SimpleSetRequirement(E): trait
    includes contained_objects(E), Equality(E)
\end{verbatim}

However, an instantiation such as

\begin{verbatim}
Set<Set<int /*@ using int @*/ /*@using SimpleSetTrait(int, Set<int>)@*/>
\end{verbatim}

is not legal unless operator == is also specified for the type \texttt{Set<int /*@ using int @*/}}.

The trait, \texttt{SimpleSetTrait} used in the interface specification of the above example is as follows. Note how it defines the trait function \texttt{contained_objects} by using the trait \texttt{container_objs defined above.}

\begin{verbatim}
% @(#)Id: SimpleSetTrait.lsl,v 1.2 1997/07/31 17:39:41 leavens Exp $
SimpleSetTrait(E, C): trait
    includes ChoiceSet, container_objs(choose for head, rest for tail),
    PureValue(C)
\end{verbatim}

If a named signature is not needed, one can explicitly write out the description of the operations in the \textit{where-clause}. Again, this kind of \textit{where-clause} specifies that the actual template argument must have a specification that conforms to the given specification. This is just like the \textit{where-clause} in Larch/CLU [Wing87]. For example, we could rewrite the simple set example above as follows.

\begin{verbatim}
\end{verbatim}
template <class Elem /*@ expects SimpleSetRequirement(Elem) @*/>  
//@ where Elem is  
//@   bool operator ==(Elem x, Elem y);  
//@ behavior {  
//@     ensures returns \ result = (x = y);  
//@ }  
//@ }; 

class Set {
public:  
//@ uses SimpleSetTrait(Elem for E, Set<Elem> for C); 

Set() throw();  
//@ behavior {  
//@   constructs self;  
//@   ensures liberally self' = empty;  
//@ }  

void insert(Elem e) throw();  
//@ behavior {  
//@   modifies self;  
//@   ensures liberally self' = self^ \ U \{e\};  
//@ }  

bool is_in(Elem e) const throw();  
//@ behavior {  
//@   ensures result = (e \in self);  
//@ }  

};

When the optional keyword \texttt{class} is used in this explicit form of a \texttt{where-body} it specifies that the actual type parameter must be a class type, not just any\ C++\ type. The use of the \texttt{class} keyword also enables all the syntax and semantics of class specifications, allowing the use of \texttt{self}, and the specification of constructors, destructors etc. Normally one would, however, only specify public member functions, and so one should use the keyword \texttt{public:} at the beginning.

As shown above, in general, the actual parameter must provide a trait that matches the expected trait theory used in the specification\cite{Goguen84}\cite{Ernst-etal91}\cite{Edwards-etal94}. In Larch/C++, this theory is provided by the use of the keyword \texttt{using} to pass traits that were “expected”. See Section 6.13 [Specifying Higher-Order Functions], page 157 for the syntax of the \texttt{expects-clause}.

As a more complete example, we specify a priority queue. Recall that a priority queue is a container with elements drawn from a totally-ordered type. We will provide an operation \texttt{Largest} to extract the largest element, and an operation \texttt{Insert} to insert an element into the priority queue.
A trait describing the theory of the elements is expected. This trait, which is given below, says that the actual trait parameter must define the `contained_objects` trait function and a total order on the elements. For the trait giving the ordering on the elements, we use the trait `TotalOrder` from section A.12 of [Guttag-Horning93].

```lsl
PriorityQueueRequirement(E): trait
   includes contained_objects(E), TotalOrder(E for T)
```

For the abstract model of priority queues, we use the trait `PriorityQueueTrait`, given below. This uses the trait `PriorityQueue` from page 175 of [Guttag-Horning93].

```lsl
PriorityQueueTrait: trait
   assumes TotalOrder(Elem for T)
   includes PriorityQueue(Elem for E, PQ[Elem] for C),
      container_objs(Elem for E, PQ[Elem] for C),
      PureValue(PQ[Elem])
```

The specification of the template class `PriorityQueue` uses an `expects-clause` to say that a trait that satisfies `PriorityQueueRequirement(Elem)` must be passed, as described above. The operator `<=` specified in the `where-clause` must be defined for the type `Elem`; as it would be needed in the template class’s implementation.

```lsl
template <class Elem /*@ expects PriorityQueueRequirement(Elem) @*/>
   @ where Elem is {
      //@ bool operator <= (Elem x, Elem y);
      //@ behavior {
      //@   ensures returns /
      //@     result = (x <= y);
      //@ }
      //@ }
   }

class PriorityQueue {
   public:
      //@ uses PriorityQueueTrait(PriorityQueue<Elem> for PQ[Elem]);
```
PriorityQueue() throw();
//@ behavior {
//@ modifies self;
//@ ensures liberally self’ = empty;
//@ }

PriorityQueue(const PriorityQueue&lt;Elem&gt;& oth) throw();
//@ behavior {
//@ modifies self;
//@ ensures liberally self’ = oth\any;
//@ }

virtual ~PriorityQueue() throw();
//@ behavior {
//@ trashes self;
//@ ensures allocated(self, post);
//@ ensures redundantly assigned(self, post) => unchanged(self);
//@ example unchanged(self);
//@ example ~assigned(self,post);
//@ }

virtual void Insert(Elem e) throw();
//@ behavior {
//@ modifies self;
//@ ensures liberally self’ = add(e, self\^);
//@ }

virtual Elem Largest() const throw();
//@ behavior {
//@ requires len(self\^) >= 1;
//@ ensures result = head(self\^);
//@ }

virtual Elem RemoveLargest() throw();
//@ behavior {
//@ requires len(self\^) >= 1;
//@ modifies self;
//@ ensures result = head(self\^) \ self’ = tail(self\^);
//@ }

virtual bool IsEmpty() const throw();
//@ behavior {
//@ ensures result = isEmpty(self\any);
//@ }

virtual long int Length() const throw();
//@ behavior {
//@ ensures liberally result = len(self\any);
//@ }
virtual long int Count(Elem e) const throw();
//@ behavior {
//@   ensures liberally result = count(self\any);
//@ }
};

The destructor in the above specification is somewhat interesting. It is specified explicitly, because there are virtual functions in the specification (see Section 7.2.2 [Destructors], page 190). This particular specification allows an implementation to trash the abstract value of \texttt{self}. However, as a destructor cannot deallocate \texttt{self} (that is the job of C++), trashing in this instance simply permits the destructor to make the abstract value become unassigned (as the second example in the specification demonstrates). See Section 6.3.2 [The Trashes Clause], page 126 for details of the semantics of the \texttt{trashes-clause}.

One final thing about the above specification. If the specification were written using subtype polymorphism, then the \texttt{expects-clause} would have been omitted, and the match of the actual-parameter theory to the formal parameter theory would have occurred during the proof of behavioral subtyping.

### 8.3 Instantiation of Templates

The syntax for the instantiation of a template adds to the C++ syntax the ability to pass traits to an instantiation. The traits passed would be those expected by a template; that is those that are mentioned in an \texttt{expects-clause}. See Section 5.2.3 [Type Specifiers], page 59 for the syntax of \texttt{type-specifier}. See Section 5.4 [Declarators], page 65 for the syntax of \texttt{abstract-declarator}. See Section 6.13 [Specifying Higher-Order Functions], page 157 for the syntax of \texttt{using-trait-list}.

\begin{verbatim}
template-instance ::= template-class-instance | template-non-class-instance
template-class-instance ::= template-class-name template-instance-actuals
template-non-class-instance ::= template-non-class-name template-instance-actuals
template-instance-actuals ::= < [ template-argument-list ] [ using-trait-list ] >
template-argument-list ::= template-argument [ , template-argument ] ...
template-argument ::= template-argument [ , template-argument ] ...
type-id ::= type-specifier-seq [ abstract-declarator ]
explicit-instantiation ::= template declaration
explicit-specialization ::= template < > declaration
\end{verbatim}
Each template-argument must match the type specified in the declaration of the template-class-name. To be legal, each template-argument that is a C++ type must be specified to satisfy any restrictions placed on it by the where-seq in the specification of the template-class-name. See Section 8.2 [Requirements on Template Parameters], page 243 for details. Furthermore, the traits mentioned in the using-trait-list should satisfy the corresponding traits in the expects-clause in the template being instantiated. The expected traits are matched positionally.

For example, Stack<int /*@ using int @*/> and

\[
\text{Stack\langle Stack\langle int /*@ using int @*/ >}
\]

\[
/ *@ using SimpleStackTrait(int, Stack\langle int \rangle) @*/ >
\]

are both instances of the template class Stack (see Section 8.1 [Example Template without Requirements], page 240). However, the second instantiation is not legal (see Section 8.2 [Requirements on Template Parameters], page 243 for the specification against which this statement is made), unless the operator == required by the template Set is provided for the type Set<int>.

Since the trait used for a class template depends on the actual parameters that instantiate the template, in order to write specifications as a client of a class template, one must give a uses-clause for each instance being used. In the following somewhat contrived example, the two uses-clauses are needed to give meaning to the appropriate overloading of the trait functions used in the specification of pop_the_top_stack. These uses-clauses could also have been put within the fun-spec-body. Notice that when instantiating a template for LSL only, we do not give the using-trait-list.
extern Stack<Stack<int /*@ using int @*/>
    /*@ using SimpleStackTrait(int, Stack<int>) @*/>
myStackofStacks;

//@ uses SimpleStackTrait(int for E, Stack<int> for C);
//@ uses SimpleStackTrait(Stack<int> for E, Stack< Stack<int> > for C);

extern void pop_the_top_stack() throw();
//@ behavior {
//@ requires assigned(myStackofStacks, pre) /
//@    ~isEmpty(myStackofStacks^)
//@    /
//@    ~isEmpty(top(myStackofStacks^));
//@ modifies myStackofStacks;
//@ ensures myStackofStacks’ = push(pop(top(myStackofStacks^)),
//@    pop(myStackofStacks^));
//@ };

9 Specification Modules

The unit of specification in Larch/C++ is the *interface*. Note the difference between the *uses-clause*, which specifies an abstract model through some traits, and the *using-declaration* and the *using-directive*, which have to do with C++ “namespaces” [Stroustrup95]. (See Section 5.5 [CPP Namespace and Using Declarations], page 78 for the syntax of the latter.)

An *interface* consists of a possibly-empty sequence of top-level declaration forms.

\[\text{interface ::= top-level [ top-level ] ...} \]
\[\text{top-level ::= uses-clause | spec-decl | depends-clause} \]
\[\quad | \text{represents-clause | invariant-clause | constraint-clause} \]
\[\quad | \text{declaration | using-declaration | using-directive} \]

A C++ program file implements an *interface* if it has the same interface (see Section 2.2 [Interfaces], page 15) and if for each specified *declaration* there is a definition that satisfies it. The *spec-decls* do not have to be implemented. See Section 9.1 [Ghost Variables], page 254 for a description of their uses.

See Chapter 7 [Class Specifications], page 167 for the syntax and semantics of *invariant-clause* an and a *constraint-clause*. (Used at the top level, such clauses allow one to state invariant and history properties of global variables.) See Section 5.5 [CPP Namespace and Using Declarations], page 78 for the syntax and semantics of *using-declaration* and *using-directive*.

The other top-level declaration forms are explained below.

9.1 Ghost Variables

A *spec-decl* consists of a declaration preceded by the keyword *spec*.

\[\text{spec-decl ::= spec declaration} \]

Such declarations declare things which are may be useful for specification purposes, but which do not have to be implemented. In the literature, variables declared in this way are called *ghost variables* or *specification variables*. (The syntax for this feature is borrowed from LCL [Tan 94].)
An example of an interface that uses a *spec-decl* is the following. The *spec-decl* is used to declare the `system_clock` as a volatile variable. (It is declared volatile because it changes state in ways not controlled by a program.) This is declared using a *spec-decl* because, while in a real system there would be a clock, it might not be directly accessible as a volatile variable named `system_clock`. Thus a *spec-decl* is used for the global declaration of `system_clock`, not a *extern* decl. If the global declaration was marked *extern*, a corresponding declaration would have to appear in the implementation, and this would, in effect, require the implementation to have a variable named `system_clock`. Furthermore, without this *spec-decl*, the function `time` would be a constant function, as it takes no input arguments. (The argument pointer `tloc` is part of the standard C library.)

```
// @(#) $Id: time.lh,v 1.11 1997/07/31 04:02:27 leavens Exp $

// the following defines time_t
#include "types.lh"

//@ spec volatile time_t system_clock;

extern time_t time(time_t * tloc) throw();
//@ behavior {
//@ requires allocated(tloc, pre);
//@ modifies *tloc;
//@ ensures result = system_clock^ /	loc’ = system_clock^;
//@ also
//@ requires isNull(tloc);
//@ ensures result = system_clock^;
//@ }
```

See Chapter 7 [Class Specifications], page 167 for several examples that use specification variables to specify the abstract models of classes.

### 9.2 The Uses Clause

The *uses-clause* tells Larch/C++ what traits are used to provide the vocabulary for specifying behavior. See Section 4.14 [LSL Constants], page 51 for the syntax of *lsl-constant*.
uses-clause ::= uses trait-list ;
trait-list ::= trait [ , trait ] . . .
trait ::= trait-name [ ( renaming ) ]
trait-name ::= simple-id
renaming ::= replace-list | lsl-sort-list [ , replace-list ]
replace-list ::= replace [ , replace ] . . .
replace ::= lsl-sort for lsl-formal
lsl-sort-list ::= lsl-sort [ , lsl-sort ]
lsl-sort ::= simple-id [ lsl-instance-actuals ]
     | built-in-type-name
     | lsl-constant
lsl-instance-actuals ::= [ lsl-sort-list ]
     | < lsl-sort-list >
lsl-formal ::= lsl-sort
     | simple-id [ : lsl-signature ]
ls1-signature ::= [ lsl-sort-list ] -> lsl-sort

The uses-clause with its renamings should be thought of as producing a new trait, which is used by Larch/C++. The syntax for renaming matches that of LSL [Guttag-Horning93], and has the same meaning as in LSL. This “trait” may have a trait-name that would be illegal in LSL, such as Set<int>.
10 Refinement

A refinement is a (possibly) stronger version of a specification. That is, a refinement may have fewer implementations that satisfy it than the original specification. This chapter describes how to specify a refinement of an existing specification, without rewriting the existing specification. Leaving the original specification alone allows you to specify different perspectives on the same class or function. Having different perspectives may be helpful to the reader; for example, you can provide an informal specification to orient the reader, a formal specification without implementation-specific details for use by client, and also a specification that records design decisions for use by maintainers of an implementation. Having the refinements is better than having a single specification, which would necessarily be overly-detailed for some readers.

In Larch/C++, refinements can be given for any declaration or member-declaration (see Section 7.2 [Class Member Specifications], page 186).

The syntax of a refinement-declaration is as follows.

\[
\text{refinement-declaration ::= [ refine-prefix ] declaration} \\
\text{refine-prefix ::= refine refinable-id [ with replace-list ] by} \\
\text{refinable-id ::= original-class-name | typedef-class-name} \\
\quad | \text{class-key identifier} \\
\quad | \text{unqualified-id} \\
\quad | \text{template-class-name | template-non-class-name} \\
\quad | \text{original-namespace-name | namespace-alias-name}
\]

See Section 9.2 [The Uses Clause], page 255 for the syntax of replace-list.

The declaration in a refinement-declaration must use the same identifier as the refinable-id. The refinable-id must name a class, struct, union, template, function, or namespace that has already been defined.

The basic semantics is that the C++ program module that includes the refinement-declaration must have a definition that satisfies the specification given in the refinement-declaration's declaration. The exact notion of satisfaction depends on the kind of declaration, but the idea is that the implementation must satisfy both the original specification and the specifications explicitly given in the refinement-declaration's declaration. The subsections below describe examples of refinement for various kinds of declarations.
10.1 Function Refinement

Refinement of functions specified at the top-level works by the same mechanism as specification inheritance (see Section 7.9 [Inheritance of Specifications and Subtyping], page 215). That is, an implementation of the refinement must satisfy the original specification, as well as any additional specification cases.

(See Section 10.2 [Class Refinement], page 260 for how to refine the specification of a member function in a class by refining the class as a whole. See Section 10.5 [Namespace Refinement], page 267 for how to refine the specification of a function declared in a namespace.)

For example, consider the following informal specification.

```
//@ refine SetToRMS
//@ by
extern void SetToRMS(double & v, double x, double y) throw();
//@ behavior {
//@ requires informally "x and y are not too big";
//@ modifies v;
//@ ensures informally "v' is an approximation to"
//@ "the root mean square of x and y";
//@ }
```

This can be refined into a formal specification as follows.

```
#include "SetToRMS-informal.lh"

//@ refine SetToRMS
//@ by
extern void SetToRMS(double & v, double x, double y) throw();
//@ behavior {
//@ requires (x*x) + (y*y) < DBL_MAX;
//@ modifies v;
//@ ensures approx(v' * v', rational((x*x) + (y*y) / 2.0), 1/100);
//@ }
```

The above specification allows either the positive or the negative root to be placed in v. We could refine the above specification further by specifying that the positive root only is to be placed
in \( v \). This is done below. Since this additional specification is added to the specification above, there is no need to repeat the postcondition above. (The precondition must be repeated, because otherwise the function would be required to terminate normally even when the precondition is not met. The modifies-clause also must be repeated, because otherwise the specification would become unsatisfiable.)

```
//@ refine SetToRMS
//@ by
extern void SetToRMS(double & v, double x, double y) throw();
//@ behavior {
//@ requires (x*x) + (y*y) < DBL_MAX;
//@ modifies v;
//@ ensures v' >= 0.0;
//@ }
```

A function that satisfies the above refinement must return a positive approximation to the root mean square of \( x \) and \( y \). This can be seen clearly in the following slight desugaring, where all the specification cases have been gathered into one specification. The idea is that the function has to satisfy all of these spec-cases. (See Section 6.10 [Case Analysis], page 143 for the meaning of such a specification and a further desugaring.)

```
//@ refine SetToRMS
//@ by
extern void SetToRMS(double & v, double x, double y) throw();
//@ behavior {
//@ requires informally "x and y are not too big"
//@ modifies v;
//@ ensures informally "v' is an approximation to"
//@ "the root mean square of x and y";
//@ also
//@ requires (x*x) + (y*y) < DBL_MAX;
//@ modifies v;
//@ ensures approx(v' * v', rational((x*x) + (y*y) / 2.0), 1/100);
//@ also
//@ requires (x*x) + (y*y) < DBL_MAX;
//@ modifies v;
```
Chapter 10: Refinement

[72x734]//@ ensures v' >= 0.0;
//@ }

[[[Does one ever need to use the with refine-list with a function refinement?]]]

10.2 Class Refinement

Class refinement allows one to strengthen a class specification, by possibly changing the abstract model and adding additional cases to the member function specifications. One use of refinement is to increase the formality of a specification; that is, one can refine an informal specification by a formal one. Another use of refinement is to specify the protected or private interface of a class, by refining a specification of its public interface. (See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for examples of how that is done.)

Refinement works by the same mechanism as specification inheritance (see Section 7.9 [Inheritance of Specifications and Subtyping], page 215), but unlike subtyping, refinement also applies to constructors and non-virtual member functions. Also, with refinement, both the refined and the refining classes must have the same name. Because the classes may have the same name, it is possible to use the same set of abstract values for each.

The interface of the refinement may have more members than the class it refines. This is useful, for example, when exposing details about protected and/or private members in the refinement. Furthermore, the interface of a member function in the refinement may allow fewer exceptions to be thrown than in the refined specification. For example the refinement may give an exception-decl when none was given originally (see Section 6.11 [Exceptions], page 148). (Recall that in C++, not giving an exception-decl allows all exceptions to be thrown.)

As an example of refinement, consider the following informal specification of a class IntSet. Note that although informal descriptions are used, the modifies-clause is easily made formal in each function.

```cpp
//@ @ ensures v' >= 0.0;
//@ }

[[[Does one ever need to use the with refine-list with a function refinement?]]]

10.2 Class Refinement

Class refinement allows one to strengthen a class specification, by possibly changing the abstract model and adding additional cases to the member function specifications. One use of refinement is to increase the formality of a specification; that is, one can refine an informal specification by a formal one. Another use of refinement is to specify the protected or private interface of a class, by refining a specification of its public interface. (See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for examples of how that is done.)

Refinement works by the same mechanism as specification inheritance (see Section 7.9 [Inheritance of Specifications and Subtyping], page 215), but unlike subtyping, refinement also applies to constructors and non-virtual member functions. Also, with refinement, both the refined and the refining classes must have the same name. Because the classes may have the same name, it is possible to use the same set of abstract values for each.

The interface of the refinement may have more members than the class it refines. This is useful, for example, when exposing details about protected and/or private members in the refinement. Furthermore, the interface of a member function in the refinement may allow fewer exceptions to be thrown than in the refined specification. For example the refinement may give an exception-decl when none was given originally (see Section 6.11 [Exceptions], page 148). (Recall that in C++, not giving an exception-decl allows all exceptions to be thrown.)

As an example of refinement, consider the following informal specification of a class IntSet. Note that although informal descriptions are used, the modifies-clause is easily made formal in each function.

```cpp
//@ @ ensures v' >= 0.0;
//@ }

[[[Does one ever need to use the with refine-list with a function refinement?]]]

10.2 Class Refinement

Class refinement allows one to strengthen a class specification, by possibly changing the abstract model and adding additional cases to the member function specifications. One use of refinement is to increase the formality of a specification; that is, one can refine an informal specification by a formal one. Another use of refinement is to specify the protected or private interface of a class, by refining a specification of its public interface. (See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for examples of how that is done.)

Refinement works by the same mechanism as specification inheritance (see Section 7.9 [Inheritance of Specifications and Subtyping], page 215), but unlike subtyping, refinement also applies to constructors and non-virtual member functions. Also, with refinement, both the refined and the refining classes must have the same name. Because the classes may have the same name, it is possible to use the same set of abstract values for each.

The interface of the refinement may have more members than the class it refines. This is useful, for example, when exposing details about protected and/or private members in the refinement. Furthermore, the interface of a member function in the refinement may allow fewer exceptions to be thrown than in the refined specification. For example the refinement may give an exception-decl when none was given originally (see Section 6.11 [Exceptions], page 148). (Recall that in C++, not giving an exception-decl allows all exceptions to be thrown.)

As an example of refinement, consider the following informal specification of a class IntSet. Note that although informal descriptions are used, the modifies-clause is easily made formal in each function.

```cpp
//@ @ ensures v' >= 0.0;
//@ }

[[[Does one ever need to use the with refine-list with a function refinement?]]]
The above specification is refined by the following. Note that since all of the function specifications that would be inherited were specified informally above, each function is respecified below. The trait, ChoiceSet used to specify the abstract values of the specification-only class Set<int> is found on page 176 of [Guttag-Horning93].

```cpp
#include "IntSetInformal.lh"
//@ spec template <class T> class Set;
//@ uses ChoiceSet(int for E, Set<int> for C), NoContainedObjects(Set<int>); 
//@ refine IntSet
//@ by
class IntSet {
  public:
    //@ spec Set<int> value;
    //@ depends self on value;
    //@ represents self by self\any = value\any;

    IntSet() throw();
    //@ behavior {
    //@ constructs value;
    //@ ensures value’ = empty;
    //@ }
```
void insert(int e) throw();
//@ behavior {
//@ modifies value;
//@ ensures value' = value \ U \{e\};
//@ }

bool is_in(int e) const throw();
//@ behavior {
//@ ensures result = (e \in (value\any));
//@ }

int pick() throw();
//@ behavior {
//@ requires ~isEmpty(value^);
//@ modifies value;
//@ ensures result \in value^ \U not(result \in value');
//@ }

Notice the depends-clause in the specification above. This is needed to allow the data member value to be modified, when the informal specification allows only self to be modified.

The represents-clause in the above specification says that, the abstract value of self in any state is the same as the value of value in that state.

The specification of pick above is somewhat loose, in that it does not require only the result to be deleted from the post-state value of self. We can correct this oversight by an additional refinement, shown below. The refinement also limits the exceptions that can be thrown by pick and specifies an additional member function size.

#include "IntSet.lh"

//@ refine IntSet
//@ by
class IntSet {
public:
  int pick() throw();
//@ behavior {
//@ requires ~isEmpty(value^);
//@ modifies value;
//@ ensures result \in value^ \U value' = delete(result, value^);
//@ }
This second refinement of `IntSet` uses the same set of specification variables (and hence abstract values) as the first.

When making a refinement that exposes more information about a type, for example, when adding operations of the protected or private interface, it is useful to change the set of abstract values. See Section 10.3 [Specifying Protected and Private Interfaces], page 263 for an example. However, before seeing how to do that, it may be useful to review at how exposed data members can be specified (see Section 7.11 [Specifying Exposed Data Members], page 235), since they are common in protected and private specifications.

### 10.3 Specifying Protected and Private Interfaces

Larch/C++ is the only specification language (that we know of) that allows you to specify the protected and private interfaces of a C++ class. (LM3, see Chapter 6 of [Guttag-Horning93] and [Jones91], has a similar ability with the its partial revelation mechanism that mimics Modula-3.) Specifying the protected interface documents (some of) the information needed to program and verify a derived class. Specifying a private interface documents implementation design decisions, and also specifies information needed to program and verify friend functions.

The difficulty with specifying the protected interface is that one often needs to specify exposed data members; this is nearly always a necessity when specifying the private interface. To do this one uses the idea described above (see Section 7.11 [Specifying Exposed Data Members], page 235).

However, most of the time a class specification is read, it will be read by clients, who are only concerned with the public interface. Hence, it is very handy to use the refinement mechanism to specify the protected interface of a class as a refinement of its public interface (see Section 10.2 [Class Refinement], page 260), and to specify the private interface (if at all), as a refinement of the protected interface.

If one is specifying the traits that model the class's abstract values explicitly, then one has to provide a simulation function (often called an abstraction function in this context [Hoare72a])

```c++
int size() throw();
//@ behavior {
//@   ensures result = size(value\any);
//@ }
```
to map the abstract values of the refined specification to those of the specification being refined. See Section 7.9.2 [Inheritance with Explicitly-Given Traits and Weak Subtyping], page 223, for an example of how to use simulation functions. The only difference, in the case of refinement, is that one will sometimes have to rename the sort being refined if one is changing the set of abstract values; to do this, one uses a refine-prefix containing a with replace-list, where the replace-list changes the name of the old abstract values to some other name. For example, one might write the following.

```
refine MyType with OldMyType for MyType by
...
```

Then the trait that defines the new sort for MyType can refer to both MyType and OldMyType. For example, it can use both sorts to define a simulation function.

However, in most cases, it is more convenient to use specification variables and let Larch/C++ worry about the traits, the renaming, and the simulation function. As an example, consider refining the specification of the type IntSet given above (see Section 10.2 [Class Refinement], page 260). Suppose we wish to implement IntSet using a private integer list and integer data members, the_elems and the_size. To do this, the specification of the private interface of IntSet would be written as follows, starting the refinement from the specification of the public interface in IntSet2.lcc given above.

```c++
// @(#)Id: IntSetPrivate.lh,v 1.16 1999/03/04 03:32:22 leavens Exp $

#include "IntSet2.lh"
#include "IntList.lh"

//@ refine IntSet
//@ by
class IntSet {
private:
  IntList the_elems;
  int the_size;

//@ depends value on the_elems, the_size;
//@ represents value by
//@ \A i: int (i \in value\any = i \in the_elems\any)
//@ \\ size(value\any) = the_size\any;
// Invariants documenting design decisions (rep invariants).
//@ invariant \A i: int (i \in the_elems\any
//@ => count(i, the_elems\any) = 1);
//@ invariant the_size\any = len(the_elems\any);

};
```
Notice that, in this refinement of \texttt{IntSet}, the ghost variable \textbf{value} is declared to depend on the private variables \texttt{the\_elems} and \texttt{the\_size}. The relation between the specification-only variable, \texttt{value}, and \texttt{the\_elems} is given in the \texttt{represents-clause} that follows the \texttt{depends-clause}. It states that \texttt{the\_elems} contains the same elements as \texttt{value}, and that the value of \texttt{the\_size} is equal to the size of \texttt{value} in any visible state.

A desugared form of the above specification, is given by the following. This desugaring uses the same kind of rewriting as the desugaring for specification inheritance (see Section 7.9 [Inheritance of Specifications and Subtyping], page 215). In addition to copying the cases from the other parts of this specification, several redundant postconditions have been added to help make the effect of this last refinement clearer.

```cpp
#include "IntList.lh"
// from IntSet.lh
//@ spec template <class T> class Set;
//@ uses ChoiceSet(int for E, Set<int> for C), NoContainedObjects(Set<int>);
class IntSet {
  private:
    IntList the\_elems;
    int the\_size;

    //@ depends value on the\_elems, the\_size;
    //@ represents value by
    //@ \A i: int (i \in value\any = i \in the\_elems\any)
    //@ \A size(value\any) = the\_size\any;

    //@ Invariants documenting design decisions (rep invariants).
    //@ invariant \A i: int (i \in the\_elems\any =>
    //@ count(i, the\_elems\any) = 1);
    //@ invariant the\_size\any = len(the\_elems\any);

  public:
    // meaning of inherited specifications follows.

    //@ spec Set<int> value;
    //@ depends self on value;
```
IntSet() throw();
//@ behavior {
//@ constructs self; // from IntSetInformal.lh
//@ ensures informally "self’ is {}";
//@ also
//@ constructs value; // from IntSet.lh
//@ ensures value’ = empty;
//@ ensures redundantly the elems’ = empty \ the size’ = 0; // by invar.
//@ }

void insert(int e) throw();
//@ behavior {
//@ modifies self; // from IntSetInformal.lh
//@ ensures informally "e is added to self";
//@ also
//@ modifies value; // from IntSet.lh
//@ ensures value’ = value^ \ {e};
//@ ensures redundantly e \in the elems’; // by the invariant
//@ }

bool is_in(int e) const throw();
//@ behavior {
//@ // from IntSetInformal.lh
//@ ensures informally "result is true just when e is a member of self"
//@ also
//@ ensures result = (e \in (value\any)); // from IntSet.lh
//@ ensures redundantly result = (e \in (the elems\any)); // by invar.
//@ }

int pick() throw();
//@ behavior {
//@ requires informally "self is not {}"; // from IntSetInformal.lh
//@ modifies self;
//@ ensures informally "result is some member of self"
//@ "and result is also deleted from self";
//@ also
//@ requires “isEmpty(value”); // from IntSet.lh
//@ requires redundantly “the size = 0”; // by the invariant
//@ modifies value;
//@ ensures result \in value^ \ not(result \in value’);
//@ ensures redundantly not(result \in the elems’); // by the invariant
//@ also
//@ requires “isEmpty(value”); // from IntSet2.lh
//@ requires redundantly “the size = 0”; // by the invariant
//@ modifies value;
//@ ensures result \in value^ \ value’ = delete(result, value’);
//@ }

int size() throw();
//@ behavior {
//@ ensures result = size(value\any); // from IntSet2.lh
//@ ensures redundantly result = the_size\any; // by the invariant
//@ }
};

10.4 Template Refinement

[[[This is the same as either function or class refinement. An example is needed.]]]

10.5 Namespace Refinement

[[[More work is needed to describe namespace refinement. The idea is similar to that described for classes above. New declarations can be added, as with classes; however, there is already in C++ a form for extending a namespace with new declarations, which should be used if no existing declarations in the namespace is to be strengthened. There is no trait for a namespace, so the renaming doesn’t apply.]]]

10.6 Nested Refinement

The following syntax allows refinement of member declarations within classes.

\[
\text{refinement-member-decl ::= [ refine-prefix ] member-declaration}
\]

[[[Needs explanation and examples.]]]
11 Built-in Types

There is a standard set of traits associated with the C++ built-in types and type constructors. They provide LSL sorts and trait functions for C++ built-in types and types constructed from them. This chapter serves two purposes: (1) it is a reference to the built-in traits for Larch/C++ specifiers, and (2) it is a formal specification of the C++ built-in types. We expect that the formal specification of the C++ built-in types would be useful in formal program verification, as well as an aid to writing C++ code.

11.1 Integer Types

C++ has four fundamental types to represent integers of different sizes: char, short, int, and long. For each of these types, there is a corresponding unsigned type to represent unsigned integer with the same number of bits as the plain (signed) type. It is required that unsigned integers obey the laws of arithmetic modulo $2^n$, where $n$ is the number of bits in the representation (see Section 3.6.1 of [Ellis-Stroustrup90]). However, the signed types have an infinite number of abstract values, most of which are not representable on a computer. One has to use range assertions (e.g., inRange or comparisons to such limits as INT_MIN and INT_MAX) if one wishes to ensure that the abstract value is representable.

In this section we describe the abstract values of C++ integer types, by giving the traits used to model them in Larch/C++. The common foundation for the various integer traits is the trait Integer found in the LSL Handbook (Appendix A of [Guttag-Horning93]). This trait defines unbounded integers with usual integer operations (+, -, *, div, mod, etc.).

See Section 4.13 [Literals], page 47 for the syntax of literals that denote abstract values of the sorts specified here.

11.1.1 Signed Trait

The trait signed specifies four signed integer types and several conversion functions between them; the actual specifications come from the included traits char, short, int, and long. It also put some constraints on the size of integer types; that is, char is a subrange of short, and short is a subrange of int, which in turn is a subrange of long.
signed: trait

includes char, short, int, long

introduces
to_short: char -> short
to_int: short -> int
to_long: int -> long

asserts
\forall c: char, s: short, i:int
to_short(0) == 0;
to_short(succ(c)) == succ(to_short(c));
to_short(pred(c)) == pred(to_short(c));
to_int(0) == 0;
to_int(succ(s)) == succ(to_int(s));
to_int(pred(s)) == pred(to_int(s));
to_long(0) == 0;
to_long(succ(i)) == succ(to_long(i));
to_long(pred(i)) == pred(to_long(i));
LONG_MIN <= to_long(INT_MIN);
INT_MIN <= to_int(SHRT_MIN);
SHRT_MIN <= to_short(CHAR_MIN);
to_short(CHAR_MAX) <= SHRT_MAX;
to_int(SHRT_MAX) <= INT_MAX;
to_long(INT_MAX) <= LONG_MAX

11.1.2 Short Integer Trait

The trait short is paradigmatic for the other integer traits. It is defined as follows.

short(short): trait

includes Integer(short), % from LSL handbook
   Between(short), NoContainedObjects(short)

introduces
   SHRT_MIN, SHRT_MAX: -> short
   inRange: short -> Bool
asserts
\forall s: short
  \text{SHRT\_MIN} == (- \text{SHRT\_MAX}) - 1;
  \text{SHRT\_MIN} <= 0 \land 1 <= \text{SHRT\_MAX};
  \text{inRange}(s) == (\text{SHRT\_MIN} <= s \land s <= \text{SHRT\_MAX});
implies
\forall s: short
  \text{inRange}(s) == \text{between}(\text{SHRT\_MIN}, s, \text{SHRT\_MAX});

The trait functions \text{SHRT\_MIN} and \text{SHRT\_MAX} are supposed to denote the minimum and maximum values of type \text{short} on a particular machine; these are typically implemented by \text{C++} macros in standard header files.

The trait \text{Between} defines some useful shorthands. It is shown below.

11.1.3 Long Integer Trait

The trait for long integers is defined by renaming sorts and trait functions of the trait \text{short} (see Section 11.1.2 [Short Integer Trait], page 270).
11.1.4 Char Trait

The trait for characters is defined by renaming sorts and trait functions of the trait `short` (see Section 11.1.2 [Short Integer Trait], page 270).

```lsl
% use the trait char_literals for literals
char(char): trait
  includes short(char, CHAR_MIN for SHRT_MIN, CHAR_MAX for SHRT_MAX)
```

11.1.5 int Trait

The trait `int` is somewhat different from others in that it also defines conversion functions between `int` and `Bool`. The trait functions `to_int` and `to_bool` respectively convert `Bool` values to sort `int` and `int` values to sort `Bool`. (The sorts `Bool` and `bool` are synonyms. The sort `int` is not a synonym of the LSL sort `Int`, because they have different properties.) As in C++, 0 is treated as `false` and all non zero values are regarded as `true`.

```lsl
int(int): trait
  includes short(int, INT_MIN for SHRT_MIN, INT_MAX for SHRT_MAX)
  introduces
    to_bool, to_LSL_Bool: int -> Bool
    to_int: Bool -> int
  asserts
    \forall i: int
    to_int(false) == 0;
    ~(to_int(true) = 0);
    to_bool(i) == ~(i = 0);
    to_LSL_Bool(i) == to_bool(i);
```

11.1.6 Unsigned Integer Trait

C++ unsigned types (`unsigned char`, `unsigned short`, `unsigned int`, and `unsigned long`) represent unsigned integers with the same number of bits as their corresponding `signed` integer.
The following traits specify the abstract values of the types unsigned char, unsigned short, unsigned int, and unsigned long. The included trait IntCycle(first, last, N) found in the LSL Handbook, defines a finite subrange of integers from first to last. The subrange includes 0 and wraps at succ(last), thus it obeys the laws of arithmetic modulo last.

\begin{verbatim}
unsignedShort: trait
    includes IntCycle(0, USHRT_MAX, unsignedShort),
           NoContainedObjects(unsignedShort)

unsignedChar: trait
    includes IntCycle(0, UCHAR_MAX, unsignedChar),
           NoContainedObjects(unsignedChar)

unsignedInt: trait
    includes IntCycle(0, UINT_MAX, unsignedInt),
           NoContainedObjects(unsignedInt)

unsignedLong: trait
    includes IntCycle(0, ULONG_MAX, unsignedLong),
           NoContainedObjects(unsignedLong)
\end{verbatim}
A C++ unsigned integer constant, an integer constant with suffix u or U (see Section 4.13 [Literals], page 47) is a term of sort unsignedInt; for example, 2U is treated as a synonym of succ(succ(0)).

### 11.1.7 Summary of Trait Functions for Integer Traits

The following tables shows trait functions defined for integer types. The sort S is a meta sort representing any of char, short, int, long, unsignedChar, unsignedShort, unsignedInt, and unsignedLong. One exception is in the signature of the trait function inRange in which the sort S stands for any signed integer sorts (e.g., char, int, etc.).

<table>
<thead>
<tr>
<th>Trait Functions</th>
<th>Signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_MIN, CHAR_MAX</td>
<td>char -&gt; char</td>
</tr>
<tr>
<td>SHRT_MIN, SHRT_MAX</td>
<td>short -&gt; short</td>
</tr>
<tr>
<td>INT_MIN, INT_MAX</td>
<td>int -&gt; int</td>
</tr>
<tr>
<td>LONG_MIN, LONG_MAX</td>
<td>long -&gt; long</td>
</tr>
<tr>
<td>UCHAR_MAX</td>
<td>unsignedChar -&gt; unsignedChar</td>
</tr>
<tr>
<td>USHRT_MAX</td>
<td>unsignedShort -&gt; unsignedShort</td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>unsignedInt -&gt; unsignedInt</td>
</tr>
<tr>
<td>ULONG_MAX</td>
<td>unsignedLong -&gt; unsignedLong</td>
</tr>
<tr>
<td>to_short</td>
<td>char -&gt; short</td>
</tr>
<tr>
<td>to_bool, to_LSL_Bool</td>
<td>int -&gt; Bool</td>
</tr>
<tr>
<td>to_int</td>
<td>Bool -&gt; int</td>
</tr>
<tr>
<td>to_int</td>
<td>short -&gt; int</td>
</tr>
<tr>
<td>to_long</td>
<td>int -&gt; long</td>
</tr>
<tr>
<td>to_unsignedShort</td>
<td>unsignedChar -&gt; unsignedShort</td>
</tr>
<tr>
<td>to_unsignedInt</td>
<td>unsignedShort -&gt; unsignedInt</td>
</tr>
<tr>
<td>to_unsignedLong</td>
<td>unsignedInt -&gt; unsignedLong</td>
</tr>
<tr>
<td>0, 1</td>
<td>-&gt; S</td>
</tr>
<tr>
<td>succ, pred</td>
<td>-&gt; S</td>
</tr>
<tr>
<td>inRange</td>
<td>S -&gt; Bool</td>
</tr>
<tr>
<td>-, abs</td>
<td>S -&gt; S</td>
</tr>
<tr>
<td>-, +, *</td>
<td>S, S -&gt; S</td>
</tr>
<tr>
<td>div, mod, min, max</td>
<td>S, S -&gt; S</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>S, S -&gt; Bool</td>
</tr>
<tr>
<td>between</td>
<td>S, S, S -&gt; Bool</td>
</tr>
<tr>
<td>strictly_between</td>
<td>S, S, S -&gt; Bool</td>
</tr>
<tr>
<td>between</td>
<td>S, S, S, S -&gt; Bool</td>
</tr>
<tr>
<td>strictly_between</td>
<td>S, S, S, S -&gt; Bool</td>
</tr>
<tr>
<td>=, ~=</td>
<td>S, S -&gt; Bool</td>
</tr>
</tbody>
</table>
11.2 Wide Characters

The abstract values of wide characters, the C++ type `wchar_t` are modeled by the following trait. Since this type is implementation dependent, this trait doesn’t say much.

```plaintext
wchar_t: trait
  includes char(wchar_t, WCHAR_MIN for CHAR_MIN, WCHAR_MAX for CHAR_MAX)
```

11.3 Floating Point Types

In this section we describe the abstract values of C++ floating point types, by giving the traits used to model them in Larch/C++. The common foundation for the various floating point traits is the trait `FloatingPoint` found in the LSL Handbook (Appendix A of [Guttag-Horning93]).

See Section 4.13 [Literals], page 47 for the syntax of literals that denote abstract values of the sorts specified here.

The abstract values of `float` types are described by the trait `float`. The names FLT_MIN, FLT_MAX, and FLT_EPSILON are given in section 3.2.2c of [Ellis-Stroustrup90], as to be defined in the header `<float.h>`. The names of the similar constants used for doubles and long doubles are in the same place.

```plaintext
float(float): trait
  includes FloatingPoint(to_float for float, FLT_MIN for smallest,
    FLT_MAX for largest, FLT_EPSILON for gap, float for F),
  NoContainedObjects(float)
```

The abstract values of `double` types are described by the trait `double`.

```plaintext
double: trait
  includes float(to_double for to_float, DBL_MIN for FLT_MIN,
    DBL_MAX for FLT_MAX, DBL_EPSILON for FLT_EPSILON,
    double for float)
```

The abstract values of `long double` types are described by the trait `longDouble`.
11.4 bool

The latest draft of the C++ standard [ANSI95] includes the type bool (see also [Stroustrup95]).

The abstract values of bool variables are modeled by the LSL sort Bool. This exception to the rule that sort names are the same as type names is made because the Booleans are so basic, and because having two separate sorts with conversions was inherently error-prone.

The trait Boolean that defines the sort Bool, and hence its synonym bool is found on page 161 of [Guttag-Horning93]. It is implicitly included in every LSL trait, and also by Larch/C++.

Larch/C++ also uses a built-in trait that defines some of the C++ Boolean operators as lsl-ops (see Section 6.1.5 [LSL Operator Terms], page 92). This trait is given below. See Section 6.1.2 [Logical Connectives], page 87 for details on the syntax of logical connectives in Larch/C++ with better precedence than these.
Technically the identification of the `bool` and `Bool` is accomplished at the LSL level by having the following synonym declaration in the Larch/C++ LSL initialization file `lslinit.lsi`.

```
synonym Bool bool
```

This synonym declaration is standard with the LSL initialization file that ships with Larch/C++ and the LSL checker. The complete initialization file is given below.

```
commentSym %
% @((#)Id: lslinit.lsi,v 1.2 1995/11/03 05:56:37 leavens Exp $
%
% Larch Shared Language (LSL) Initialization File
%
% Original from page 223 in Guttag and Horning, ‘‘Larch: Languages and Tools for Formal Specification’’
%
% Modifications
%
% 93.11.09 Garland at MIT Add synonym: formulas for equations
% 93.11.26 Garland at MIT Add synonym: <=> for \iff
% 94.05.23 Garland at MIT Remove [ as openSym, ] as closeSym
% Remove synonyms: & for \and, | for \or,
% ! for \not, != for \neq
% Remove synonyms used for checking LCL
% Add synonym: with for \forall
% 95.01.03 Leavens at ISU took out Bool synonym
% 95.11.02 Leavens at ISU put back Bool synonym bool

idChar ,
opChar "!#$&@| singleChar ;
openSym { \langle \langle closeSym } \rangle \rangleangle
selectSym .
simpleId \bot \top

synonym \and /
synonym \or /
synonym \implies =>
synonym \iff <=>
synonym \not not
synonym \eq =
synonym \neq ~=
synonym \arrow ->
```
11.5 void

Although many programmers do not think of void as a type, and those that do often think of it as a type with no values, the type void is properly modeled as a type with exactly one value. This is modeled by the following trait in Larch/C++. (See Section 6.11 [Exceptions], page 148 for the trait NoInformation.)

```lsl
% $Id: void.lsl,v 1.2 1995/11/13 16:24:07 leavens Exp $
void : trait
    includes NoInformation(void, theVoid for it)
    implies
      \forall x, y: void
      x == theVoid;
      x == y;
```

One reason to have a value of the void type is to have a model for the abstract value of a “void pointer”, that is a pointer of type void*. The abstract value of an object pointed to by such a pointer exists, but has no information content. Therefore the abstract value of such an object is theVoid, which is the only abstract value of type void.

11.6 Enumeration Types

The abstract values of an enumeration type in C++ are given by a trait constructed according to that type’s declaration (see Section 5.3 [Enumeration Declarations], page 64).

For example, consider the following declaration.

```c
enum day_of_week { sun=1, mon, tues, wed, thurs, fri, sat };```
The trait for the `day_of_week` example would be as follows. See Section 11.1 [Integer Types], page 269 for the trait `int`.

```plaintext
\% @(#)Id: day_of_week_Trait.lsl,v 1.6 1997/06/03 20:29:59 leavens Exp $

day_of_week_Trait: trait
  includes int, NoContainedObjects(day_of_week)
  introduces
    sun, mon, tues, wed, thurs, fri, sat: -> day_of_week
  to_int: day_of_week -> int

asserts
  day_of_week generated by sun, mon, tues, wed, thurs, fri, sat
  day_of_week partitioned by to_int: day_of_week -> int

equations
  to_int(sun) == 1;
  to_int(mon) == 2;
  to_int(tues) == 3;
  to_int(wed) == 4;
  to_int(thurs) == 5;
  to_int(fri) == 6;
  to_int(sat) == 7;

implies
  \forall d1, d2: day_of_week
    (d1 = d2) == (to_int(d1) = to_int(d2))
  converts to_int: day_of_week -> int
```

The trait corresponding to an enumeration declaration is implicitly used in any specification module in which the declaration appears. For example, the trait `day_of_week_Trait` would be used in the module in which the declaration of `day_of_week` appears. (See Section 2.7 [Types and Sorts], page 20 for more details on using a trait in a specification.)

### 11.7 Array Types

The abstract value of a C++ array is a mapping from indexes to objects. As in C++, the index is 0-based. For an array `a`, `a[i]` denotes the i-th object of the array `a`, where `0 <= i <= maxIndex(a)` and counting starts (as in C++) from zero (0). The trait function `maxIndex` represents the upper bound of an array. This basic idea is captured by the following trait.
Val_Array(Elem): trait

includes int, % trait for C++ built-in type int.
    Set(int, Set[int], int for Int)

introduces
    create: int -> Arr[Elem]
    allocated, allocatedUpTo: Arr[Elem], int -> Bool
    allAllocated: Arr[Elem] -> Bool
    allocatedInRange: Arr[Elem], int, int -> Bool
    assign: Arr[Elem], int, Elem -> Arr[Elem]
    __[__]: Arr[Elem], int -> Elem
    maxIndex, minIndex: Arr[Elem] -> int
    legalIndex: Arr[Elem], int -> Bool
    size: Arr[Elem] -> int
    slice: Arr[Elem], int, int -> Arr[Elem]
    slice_helper: Arr[Elem], int, int, Arr[Elem], Set[int] -> Arr[Elem]

asserts
    sort Arr[Elem] generated by create, assign
    sort Arr[Elem] partitioned by size, allocated, __[__]

\forall a: Arr[Elem], i,j,n,siz: int, e: Elem
    \neg allocated(create(n), j);
    allocated(assign(a,i,e), j)
    == legalIndex(a,j) \land (i = j \lor \neg allocated(a, j));
    allAllocated(a) == allocatedInRange(a, 0, maxIndex(a));
    allocatedInRange(a, i, j)
    == i > j \lor (allocated(a, i) \land \neg allocatedInRange(a, i+1, j));
    allocatedUpTo(a, siz) == allocatedInRange(a, 0, siz-1);
    assign(a,i,e)[j] == (if i = j then e else a[j]);
    maxIndex(assign(a,i,e)) == maxIndex(a);
    maxIndex(create(n)) == n - 1;
    minIndex(a) == 0;
    legalIndex(a,i) == (0 <= i) \land (i <= maxIndex(a));
    size(create(n)) = n;
    size(assign(a,i,e)) = size(a);

\forall a, a2: Arr[Elem], i, j, k: int, s: Set[int], e: Elem
    slice(a,i,j)
    == if i <= j then slice_helper(a, i, j, create(j-i), {}) 
        else create(0);
    slice_helper(create(k), i, j, a2, s) == a2;
    slice_helper(assign(a,k,e), i, j, a2, s)
    == if i <= k \land k <= j \land \neg(k \in s) 
        then slice_helper(a, i, j, assign(a2,k,e), insert(k, s))
        else slice_helper(a, i, j, a2, s);
implies
sort Arr[Elem] partitioned by maxIndex, allocated, __[__]
\forall a: Arr[Elem], i, j: int, e1, e2: Elem
assign(assign(a,i,e1),i,e2) == assign(a,i,e2);
i ~ j => assign(assign(a,i,e1),j,e2) = assign(assign(a,j,e2),i,e1);
allocated(a,i) => legalIndex(a,i);
size(a) == maxIndex(a) + 1;
j < 0 => (slice(a,i,j) = create(0));

converts allocated: Arr[Elem], int -> Bool,
allAllocated, allocatedInRange,
maxIndex, minIndex, legalIndex, size: Arr[Elem] -> int,
__[__]
exempting \forall n, i: int
create(n)[i]

The abstract values of arrays are defined using the trait Array given below. It includes Val_Array twice. The inclusion of Val_Array(Obj[Elem]) defines the sort Arr[Obj[Elem]] as a pre-array of objects; this is the abstract value of a C++ array. The inclusion of Val_Array(Elem) defines the sort Arr[Elem] which is a pre-array of values; this is used when a state function is applied to an array (see Section 6.2.1 [State Functions], page 103).
asserts
\forall a: Arr[Loc[Elem]], i, j, k, n, siz: int, e: Loc[Elem], st: State, si: Set[int]
allocated(a, i, st) == allocated(a, i) \\/
allocated(a[i], st);
assigned(a, i, st) == allocated(a, i) \\/
assigned(a[i], st);
allocated(a, st) == allAllocated(a, st);
assigned(a, st) == allAssigned(a, st);
eval(create(i), st) == create(i);
eval(assign(a, i, e), st) == assign(eval(a, st), i, eval(e, st));
contained_objects(a, st)
== contained_without_indexes(a, st, {}):Set[int]);
contained_without_indexes(create(i), st, si) == {};
contained_without_indexes(assign(a, i, e), st, si)
== if i \in si then contained_without_indexes(a, st, si)
else \{asTypeTaggedObject(e)\}
\U contained_without_indexes(a, st, insert(i, si));
objectsInRange(a, i, j) == objectsInRangeHelper(slice(a, i, j), {});
objectsInRangeHelper(create(k), si) == {};
objectsInRangeHelper(assign(a, k, e), si)
== if k \in si then objectsInRangeHelper(a, si)
else \{asTypeTaggedObject(e)\}
\U objectsInRangeHelper(a, insert(k, si));
objectsUpTo(a, siz) == objectsInRange(a, 0, siz-1);

implies
\forall a: Arr[Loc[Elem]], i: int, e: Loc[Elem], st: State
size(contained_objects(a, st)) <= (maxIndex(a) + 1);
asTypeTaggedObject(e) \in contained_objects(assign(a, i, e), st);

converts
allocated: Arr[Loc[Elem]], int, State \to Bool,
assigned: Arr[Loc[Elem]], int, State \to Bool,
contained_objects: Arr[Loc[Elem]], State \to Set[TypeTaggedObject],
objectsInRange, objectsInRangeHelper, objectsUpTo

The trait \texttt{Array} also includes the trait \texttt{ArrayAllocatedAuxFuns} to define various predicates to test whether objects are allocated.
ArrayAllocatedAuxFuns(T): trait
assumes int, State, ArrayAllocAuxSig(T)
introduces
   allAllocated: T, State -> Bool
   allocatedInRange: T, int, int, State -> Bool
   allocatedUpTo: T, int, State -> Bool
asserts
   \forall p: T, i,j,siz: int, st: State
   allAllocated(p, st)
   == allocatedInRange(p, minIndex(p), maxIndex(p), st);
   allocatedInRange(p, i, j, st)
   == i > j \lor (allocated(p, i, st) \land allocatedInRange(p, i+1, j, st));
   allocatedUpTo(p, siz, st) == allocatedInRange(p, 0, siz-1, st);
implies
converts
   allAllocated: T, State -> Bool,
   allocatedInRange: T, int, int, State -> Bool,
   allocatedUpTo: T, int, State -> Bool

Some signature assumptions used by the trait ArrayAllocatedAuxFuns are recorded in the trait ArrayAllocAuxSig.

ArrayAllocAuxSig(T): trait
assumes int, State
introduces
   allocated: T, int, State -> Bool
   minIndex, maxIndex: T -> int

The trait Array also includes the trait ArrayAssignedAuxFuns to define various predicates to test whether objects are assigned.
ArrayAssignedAuxFuns(T): trait
   includes
      ArrayAllocatedAuxFuns(T,
       assigned for allocated: T, int, State -> Bool,
       allAssigned for allAllocated,
       assignedInRange for allocatedInRange,
       assignedUpTo for allocatedUpTo)

When an array type is used in a Larch/C++ specification, an instantiation of the trait \texttt{Array} is automatically used in the specification. In this instantiation either \texttt{Obj} or \texttt{ConstObj} is substituted for \texttt{Loc}, with the latter used if the elements are \texttt{const}, and the value sort is substituted for \texttt{Elem}.

For example, an array declaration of the form

```
typedef Spigot Spigot_Vec[17];
```

makes the interface specification use the trait \texttt{Array(Obj, Spigot)}. The same holds true for an array variable declaration (see Section 5.4.3 [Array Declarations], page 69). For example, consider the following.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Trait used</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ai[3];</td>
<td>Array(Obj, int)</td>
</tr>
<tr>
<td>const int cai[3];</td>
<td>Array(ConstObj, int)</td>
</tr>
<tr>
<td>int *aip[3];</td>
<td>Array(Obj, Ptr[Obj[int]])</td>
</tr>
</tbody>
</table>

Recall that arrays used as formal parameters are equivalent to pointers; for such uses of array types, the equivalent pointer traits are used. (See Section 11.8 [Pointer Types], page 287 for details on the abstract values of pointers.)

The objects directly contained in a one-dimensional array are the elements of the array.

A multi-dimensional array is modeled as an array of arrays. More precisely, it is a mapping from integers to arrays. For example, a two-dimensional array is a mapping from integers to arrays, and a three-dimensional array is treated as a mapping from integers to two-dimensional arrays (hence it is like an array of arrays of arrays), and so on. For example, \texttt{int ai[3][4]} declares an array of arrays of integers, and \texttt{ai[i]} denotes i-th array and \texttt{(ai[i])[j]} denotes j-th integer of the i-th
array of ai; maxIndex(ai) is 2 and for 0 <= i <= 2, maxIndex(ai[i]) is 3. Note that ai is of sort Arr[Arr[Obj[int]]], ai[i] is of sort Arr[Obj[int]], and (ai[i])[j] is of sort Obj[int] (see Section 5.4.8 [Summary of Declarations], page 76).

The trait used in defining multi-dimensional arrays is the trait MultiDimensionalArray that follows.

% @(#) $Id: MultiDimensionalArray.lsl,v 1.27 1999/01/12 22:40:17 leavens Exp $
% Supplement for higher dimensions of an array in C++

MultiDimensionalArray(SubObjArr, SubValArr): trait

assumes State, SubArray(SubObjArr, SubValArr)

includes Val_Array(SubObjArr), Val_Array(SubValArr),
   NoContainedObjects(Arr[SubValArr]),
   ArrayAllocatedAuxFuns(Arr[SubObjArr]),
   ArrayAssignedAuxFuns(Arr[SubObjArr]),
   contained_objects(Arr[SubObjArr])

introduces
   allocated, assigned: Arr[SubObjArr], int, State -> Bool
   eval: Arr[SubObjArr], State -> Arr[SubValArr]
   contained_without_indexes: Arr[SubObjArr], State, Set[int]
      -> Set[TypeTaggedObject]
   objectsInRange: Arr[SubObjArr], int, int -> Set[TypeTaggedObject]
   objectsInRangeHelper: Arr[SubObjArr], Set[int] -> Set[TypeTaggedObject]
   objectsUpTo: Arr[SubObjArr], int -> Set[TypeTaggedObject]

asserts
   \forall a: Arr[SubObjArr], i, j, k, n, siz: int, e: SubObjArr, st: State,
      si: Set[int]
   allocated(a, i, st) == allocated(a, i) \land allAllocated(a[i], st);
   assigned(a, i, st) == allocated(a, i) \land allAssigned(a[i], st);

   eval(create(n): Arr[SubValArr], st) == create(n);
   eval(assign(a, i, e), st) == assign(eval(a, st), i, eval(e, st));

   contained_objects(a, st)
      == contained_without_indexes(a, st, {}: Set[int]);
   contained_without_indexes(create(i) : Arr[SubObjArr], st, si) == {};
   contained_without_indexes(assign(a, i, e), st, si)
      == if i \in si then contained_without_indexes(a, st, si)
         else contained_objects(e, st)
         \cup contained_without_indexes(a, st, insert(i, si));
objectsInRange(a, i, j) == objectsInRangeHelper(slice(a,i,j), {});
objectsInRangeHelper(create(k), si) == {};
objectsInRangeHelper(assign(a,k,e), si)
  == if k \in si then objectsInRangeHelper(a, si)
      else objectsInRange(a[k], 0, maxIndex(a[k]))
          \U objectsInRangeHelper(a, insert(k, si));
objectsUpTo(a, siz) == objectsInRange(a, 0, siz-1);

implies
\forall a: Arr[SubObjArr], st: State
  size(contained_objects(a, st)) <= (maxIndex(a) + 1);
converts
  allocated: Arr[SubObjArr], int, State -> Bool,
  assigned: Arr[SubObjArr], int, State -> Bool,
  contained_objects: Arr[SubObjArr], State -> Set[TypeTaggedObject],
  objectsInRange: Arr[SubObjArr], int, int -> Set[TypeTaggedObject],
  objectsUpTo: Arr[SubObjArr], int -> Set[TypeTaggedObject]

The trait SubArray is included by MultiDimensionalArray as a shorthand for some signature assumptions.

% @(#) $Id: SubArray.lsl,v 1.14 1995/11/06 07:19:07 leavens Exp $

SubArray(SubObjArr, SubValArr): trait
  assumes State
  includes contained_objects(SubObjArr)
  introduces
    allAllocated, allAssigned: SubObjArr, State -> Bool
    eval: SubObjArr, State -> SubValArr
    objectsInRange: SubObjArr, int, int -> Set[TypeTaggedObject]
    maxIndex: SubObjArr -> int

Instantiations of the trait MultiDimensionalArray are implicitly used in the interface specification module in which the declaration appears. For example, consider the following.
### Declarations

<table>
<thead>
<tr>
<th>Traits to be used</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x[3][4];</td>
</tr>
<tr>
<td>Array(Obj, int),</td>
</tr>
<tr>
<td>MultiDimensionalArray(Arr[Obj[int]], Arr[int])</td>
</tr>
<tr>
<td>const int cx[3][4];</td>
</tr>
<tr>
<td>Array(ConstObj, int),</td>
</tr>
<tr>
<td>MultiDimensionalArray(Arr[ConstObj[int]], Arr[int])</td>
</tr>
<tr>
<td>int x[5][3][4];</td>
</tr>
<tr>
<td>Array(Obj[int], int),</td>
</tr>
<tr>
<td>MultiDimensionalArray(Arr[Obj[int]], Arr[int]),</td>
</tr>
<tr>
<td>MultiDimensionalArray(Arr[Arr[Obj[int]]], Arr[Arr[int]])</td>
</tr>
</tbody>
</table>

A one-dimensional array is a collection of objects (not values), so to describe the abstract value of an element one has to apply a state-fcn to that element. For example, \((a[i])^-\) denotes the pre-state value of \(a[i]\). See Section 5.4.3 [Array Declarations], page 69 for an overview. See Section 6.2.1 [State Functions], page 103 for the details.

There is a shorthand notation for applying a state function element-wise to an array, See Section 6.2.1 [State Functions], page 103.

See Section 6.2.3 [The Modifies Clause], page 110 for a shorthand that allows one to assert that one can mutate any element object of an array, not just a particular element object.

### 11.8 Pointer Types

The Larch/C++ model of C++ pointers is based on a simple idea, to which a model of the 0 pointer is added.

The simple idea is that a non-null pointer is an offset into a sequence of one or more adjacent objects. Thus, most pointers, excepting the 0 pointer, have an abstract value that is an index into an array, with two bounding indexes. The following trait describes this idea, and follows ideas in LCL (see Chapter 5 of [Guttag-Horning93]). The idea is that a pointer is modeled as a tuple of an index and an array. (The array may just have one element for a pointer to a variable.) The pointer is thus pointing at the indexed element of the array.

```%
% @(#)$Id: PrePointer.lsl,v 1.7 1995/06/20 22:49:38 leavens Exp $%
% Model of pointers as index into an array.%
% This follows the idea in LCL [page 60, Guttag-Horning93].%

PrePointer(Elem): trait```
assumes Val_Array(Elem), int

% This would be
% "Ptr[Elem] tuple of idx: int, locs: Arr[Elem]"
% but don't want set_locs, "generated by" and "partitioned by" clauses

introduces

[__, __] : int, Arr[Elem] -> Ptr[Elem]
&__ : Elem -> Ptr[Elem] % use if not in an array
address_of : Arr[Elem], int -> Ptr[Elem] % use if in an array
__.idx: Ptr[Elem] -> int
__.locs:Ptr[Elem] -> Arr[Elem]
locs: Ptr[Elem] -> Arr[Elem]
index: Ptr[Elem] -> int
set_idx: Ptr[Elem], int -> Ptr[Elem]
__ + __: Ptr[Elem], int -> Ptr[Elem]
__ - __: Ptr[Elem], int -> Ptr[Elem]
__ <__, __<=__, __>__, __>=__: Ptr[Elem], Ptr[Elem] -> Bool

asserts \forall p: Ptr[Elem], i,j: int, a,a1: Arr[Elem],
              st: State, l: Elem
&l == [0,assign(create(1), 0, l)];
address_of(a,i) == [i,a];
([i,a] = [j,a1]) == (i = j) /\ (a = a1);
([i,a]).idx == i;
([i,a]).locs == a;
locs(p) == p.locs;
index(p) == p.idx;
set_idx([i,a], j) == [j,a];
p + i == set_idx(p, p.idx + i);
p - i == set_idx(p, p.idx - i);
([i,a] < [j,a]) == (i < j);
([i,a] > [j,a]) == (i > j);
([i,a] <= [j,a]) == (i <= j);
([i,a] >= [j,a]) == (i >= j);

implies IsPO(<=, Ptr[Elem]), IsPO(>=, Ptr[Elem])

In C++, a pointer can be either pointing to something, or it can be a 0 pointer. The following
trait, PointerWithNull, adds to the pointers modeled by PrePointer, the NULL pointer, which is
the abstract value of the 0 pointer in C++. (The use of 0 as a pointer as well as an integer makes
overloading resolution in a term difficult. The name chosen, NULL, should be fairly intuitive.)
PointerWithNull(Elem): trait
  includes PrePointer(Elem)

  introduces
  NULL: -> Ptr[Elem]
  isNull, notNull, isLegal, isValid: Ptr[Elem] -> Bool
  allocated, allAllocated: Ptr[Elem] -> Bool
  minIndex, maxIndex: Ptr[Elem] -> int
  legalIndex, validUpTo, allocated, allocatedUpTo: Ptr[Elem], int -> Bool
  validInRange, allocatedInRange: Ptr[Elem], int, int -> Bool
  __:Ptr[Elem] -> Elem
  ___:Ptr[Elem], int -> Elem
  slice: Ptr[Elem], int, int -> Arr[Elem]

  asserts
  sort Ptr[Elem] generated by NULL, [__ , ___ ]
  sort Ptr[Elem] partitioned by isNull, .idx, .locs
  \\forall p: Ptr[Elem], i,j,siz: int
  isNull(p) == (p = NULL);
  notNull(p) == ~(p = NULL);
  legalIndex(p,i) == if isNull(p) then false
  else ((minIndex(p) <= i) /\ (i <= maxIndex(p)));
  isLegal(p) == isNull(p) \/= allocated(p.locs, p.idx);
  isValid(p) == notNull(p) \/= allocated(p.locs, p.idx);
  allocated(p) == isValid(p);
  allocated(p,i) == allocated(p+i);
  allAllocated(p) == isValid(p) \/= allAllocated(p.locs);
  validInRange(p,i,j) == if i > j then true
  else isValid(p+i) \/= validInRange(p,i+1,j);
  validUpTo(p,siz) == validInRange(p, 0, siz-1);
  allocatedInRange(p, i, j) == validInRange(p,i,j);
  allocatedUpTo(p,siz) == allocatedInRange(p, 0, siz-1);

  notNull(p) => (minIndex(p) = -(p.idx));
  notNull(p) => (maxIndex(p) = maxIndex(p.locs) - p.idx);
  allocated(p) => (*p = p.locs[p.idx]);
  allocated(p,i) => (p[i] = p.locs[p.idx + i]);

  notNull(p) => (slice(p, i, j) = slice(p.locs, p.idx+i, p.idx+j));
implies $\forall p: \text{Ptr}[\text{Elem}], i: \text{int}$
notNull(p) $\Rightarrow$ (minIndex(p) + maxIndex(p) = maxIndex(p.locs));
isValid(p) $\iff$ notNull(p) $\land$ isLegal(p);
allocated(p,i) $\Rightarrow$ (p[i] = *(p + i));
allocated(p,i) $\Rightarrow$ (p[i] = (locs(p))[p.idx + i]);
isValid(p) $\Rightarrow$ isLegal(p);
allocated(p) $\Rightarrow$ isLegal(p);

converts .idx, .locs, locs, index, isNull, notNull, isLegal, isValid,
allocated: \text{Ptr}[\text{Elem}] $\rightarrow$ \text{Bool}, allAllocated: \text{Ptr}[\text{Elem}] $\rightarrow$ \text{Bool},
minIndex: \text{Ptr}[\text{Elem}] $\rightarrow$ \text{int}, maxIndex: \text{Ptr}[\text{Elem}] $\rightarrow$ \text{int},
legalIndex: \text{Ptr}[\text{Elem}], \text{int} $\rightarrow$ \text{Bool},
allocated: \text{Ptr}[\text{Elem}], \text{int} $\rightarrow$ \text{Bool},
validUpTo, allocatedUpTo: \text{Ptr}[\text{Elem}], \text{int} $\rightarrow$ \text{Bool},
validInRange, allocatedInRange: \text{Ptr}[\text{Elem}], \text{int}, \text{int} $\rightarrow$ \text{Bool},
slice: \text{Ptr}[\text{Elem}], \text{int}, \text{int} $\rightarrow$ \text{Arr}[\text{Elem}]

exempting $\forall i,j: \text{int}$
NULL.idx, NULL.locs, locs(NULL),
index(NULL), minIndex(NULL), maxIndex(NULL), slice(NULL, i, j)

The symbol $*$ is one way to dereference a valid pointer. Thus $*$p produces the object that p is currently pointing to. Another way to achieve a similar effect is to use subscripts; for example, p[0] and $*$p mean the same thing, as do *(p + i) and p[i], for an int i. The trait functions minIndex(p) and maxIndex(p) respectively denote the maximum number of objects before and after $*$p. The concept of a valid pointer is captured by the trait function isValid.

The trait functions attempt to make pointers act like array names, as is the case in C++. That is, with a pointer, p, one can write expressions such as p[i], or the equivalent *(p + i), if i is between minIndex(p) and maxIndex(p). Also, the trait functions legalIndex, validRange, and validUpTo can be used to describe the legal indexes into the array into which the pointer points. These are particularly useful, therefore, for arrays passed as parameters.

However, there are some differences between array names and pointers. A notational difference is that, with an array name, a, one can only use the form a[i], not *(a + i), in Larch/C++. Besides this notational distinction, another difference from C++ is that in Larch/C++ the notation a[i] is not defined for negative numbers i. A more profound difference is that an array name (a value of a sort such as Arr[Obj[T]]) cannot be null or invalid, whereas a pointer can be null or invalid.

The trait PointerWithNull is included by the traits Pointer and PointerToArray below. The first defines pointers to objects, and the second defines pointers to arrays. The main purpose of both is to define the trait functions eval, contained_objects, and others that are used for syntactic shorthands and for describing the objects pointed at by a pointer.
% @(#)Id: Pointer.lsl,v 1.35 1995/12/24 02:51:14 leavens Exp $  
% Pointers to single objects (perhaps within an array, but not to arrays)  

Pointer(Loc, Val): trait  

includes Array(Loc, Val), PointerWithNull(Loc[Val]),  
PointerAllocatedAuxFuns(Ptr[Loc[Val]]),  
PointerAssignedAuxFuns(Ptr[Loc[Val]]),  
contained_objects(Ptr[Loc[Val]])  

assumes TypedObj(Loc, Val)  

introduces  
allocated, assigned: Ptr[Loc[Val]], int, State -> Bool  
isLegal, isValid, nullOrAssigned: Ptr[Loc[Val]], State -> Bool  
eval: Ptr[Loc[Val]], State -> Arr[Val]  
objectsInRange: Ptr[Loc[Val]], int, int -> Set[TypeTaggedObject]  
objectsUpTo: Ptr[Loc[Val]], int -> Set[TypeTaggedObject]  

asserts  
\forall p: Ptr[Loc[Val]], i,j,siz: int, st: State  
  allocated(p, i, st) == allocated(p, i) \&\& allocated(*(p+i), st);  
  assigned(p, i, st) == allocated(p, i) \&\& assigned(*(p+i), st);  
  isValid(p, st) == isNull(p) \&\& allocated(p, st);  
  isLegal(p, st) == isNull(p) \&\& assigned(p, st);  
  nullOrAssigned(p, st) == isNull(p) \&\& assigned(p, st);  
  eval(p, st) == if isValid(p) then eval(p.locs, st) else create(0);  
  contained_objects(p, st)  
    == if ~isValid(p) then {}  
    else contained_objects(p.locs, st);  
  objectsInRange(p, i, j) == if isValid(p) then objectsInRange(p.locs, p.idx + i, p.idx + j)  
    else {};  
  objectsUpTo(p, siz) == objectsInRange(p, 0, siz-1);  

implies  
\forall p: Ptr[Loc[Val]], i: int, st: State  
  assigned(p, st) => allocated(p, st);  
  allocated(p, st) => notNull(p) \&\& isLegal(p, st);  
  contained_objects(NULL, st) == \{};  
  size(contained_objects(p, st)) <= (maxIndex(p.locs) + 1);
converts
allocated: Ptr[Loc[Val]], int, State -> Bool,
assigned: Ptr[Loc[Val]], int, State -> Bool,
isLegal: Ptr[Loc[Val]], State -> Bool,
isValid: Ptr[Loc[Val]], State -> Bool,
nul1OrAssigned: Ptr[Loc[Val]], State -> Bool,
eval: Ptr[Loc[Val]], State -> Arr[Val],
contained_objects: Ptr[Loc[Val]], State -> Set[TypeTaggedObject],
objectsInRange:Ptr[Loc[Val]], int, int -> Set[TypeTaggedObject],
objectsUpTo: Ptr[Loc[Val]], int -> Set[TypeTaggedObject]
exempting
\forall p: Ptr[Loc[Val]]
  \begin{align*}
  & \text{eval}(p, \text{bottom}), \text{eval}(p, \text{emptyState})
  \end{align*}

The trait Pointer also includes the trait PointerAllocatedAuxFuns to define various predicates to test whether objects are allocated.

\texttt{% @(#)$Id: PointerAllocatedAuxFuns.lsl,v 1.2 1995/07/26 04:26:19 leavens Exp$}

\texttt{PointerAllocatedAuxFuns(PtrT): trait}
\texttt{\hspace{1cm} assumes PointerAllocAuxSig(PtrT)}
\texttt{\hspace{1cm} includes ArrayAllocatedAuxFuns(PtrT)}
\texttt{\hspace{1cm} introduces}
\texttt{\hspace{2.5cm} allocated: PtrT, State -> Bool}
\texttt{\hspace{1cm} asserts}
\texttt{\hspace{2.5cm} \forall p: PtrT, st: State}
\texttt{\hspace{3cm} allocated(p, st) == allocated(p, 0, st);}
\texttt{\hspace{1cm} implies}
\texttt{\hspace{2.5cm} converts}
\texttt{\hspace{3cm} allocated: PtrT, State -> Bool}

Some assumptions used by the trait PointerAllocatedAuxFuns are recorded in the trait PointerAllocAuxSig.

\texttt{% @(#)$Id: PointerAllocAuxSig.lsl,v 1.2 1995/07/26 04:26:19 leavens Exp$}

\texttt{PointerAllocAuxSig(PtrT): trait}
\texttt{\hspace{1cm} includes ArrayAllocAuxSig(PtrT)}
\texttt{\hspace{1cm} introduces}
\texttt{\hspace{2.5cm} __+__: PtrT, int -> PtrT}
The trait `Pointer` also includes the trait `PointerAssignedAuxFuns` to define various predicates to test whether objects are assigned.

% @(#)$Id: PointerAssignedAuxFuns.lsl,v 1.2 1995/07/26 04:26:19 leavens Exp $

`PointerAssignedAuxFuns(PtrT): trait`

`includes PointerAllocatedAuxFuns(PtrT,`
  `assigned for allocated: PtrT, State -> Bool,`
  `assigned for allocated: PtrT, int, State -> Bool,`
  `assignedUpTo for allocatedUpTo,`
  `allAssigned for allAllocated,`
  `assignedInRange for allocatedInRange)`

The following trait is similar to the `Pointer` trait, but is used for pointers to arrays or multidimensional arrays.


`PointerToArray(SubObjArr, SubValArr): trait`

`includes MultiDimensionalArray(SubObjArr, SubValArr),`
  `PointerWithNull(SubObjArr),`
  `PointerAllocatedAuxFuns(Ptr[SubObjArr]),`
  `PointerAssignedAuxFuns(Ptr[SubObjArr]),`
  `contained_objects(Ptr[SubObjArr])`

`introduces`

`allocated, assigned: Ptr[SubObjArr], int, State -> Bool`
`isLegal, isValid, nullOrAssigned: Ptr[SubObjArr], State -> Bool`
`eval: Ptr[SubObjArr], State -> Arr[SubValArr]`
`objectsInRange: Ptr[SubObjArr], int, int -> Set[TypeTaggedObject]`
`objectsUpTo: Ptr[SubObjArr], int -> Set[TypeTaggedObject]`
asserts
\forall p: \text{Ptr}[\text{SubObjArr}], i, j,\text{sz}: \text{int}, \text{st}: \text{State}
allocated(p, i, \text{st}) == allocated(p, i) \land \text{allAllocated}(*(p+i), \text{st});
assigned(p, i, \text{st}) == allocated(p, i) \land \text{allAssigned}(*(p+i), \text{st});
isLegal(p, \text{st}) == \text{isNull}(p) \lor \text{allocated}(p, \text{st});
isValid(p, \text{st}) == \text{allocated}(p, \text{st});
nullOrAssigned(p, \text{st}) == \text{isNull}(p) \lor \text{assigned}(p, \text{st});
eval(p, \text{st}) == \text{if isValid}(p) \text{ then eval}(p.\text{locs}, \text{st}) \text{ else create}(0);
\text{contained_objects}(p, \text{st})
== \text{if } \sim \text{isValid}(p) \text{ then } \{\}
  \text{ else contained_objects}(p.\text{locs}, \text{st});
objectsInRange(p, i, j)
== \text{if } \sim \text{isValid}(p) \text{ then } \{\}
  \text{ else objectsInRange}(p.\text{locs}, p.\text{idx} + i, p.\text{idx} + j);
objectsUpTo(p, \text{sz}) == objectsInRange(p, 0, \text{sz}-1);

implies
\forall \text{st}: \text{State}
\text{contained_objects}(\text{NULL}, \text{st}) == \{\};

converts
allocated: \text{Ptr}[\text{SubObjArr}], \text{int}, \text{State} \rightarrow \text{Bool},
assigned: \text{Ptr}[\text{SubObjArr}], \text{int}, \text{State} \rightarrow \text{Bool},
isLegal: \text{Ptr}[\text{SubObjArr}], \text{State} \rightarrow \text{Bool},
isValid: \text{Ptr}[\text{SubObjArr}], \text{State} \rightarrow \text{Bool},
nullOrAssigned: \text{Ptr}[\text{SubObjArr}], \text{State} \rightarrow \text{Bool},
eval: \text{Ptr}[\text{SubObjArr}], \text{State} \rightarrow \text{Arr}[\text{SubValArr}],
\text{contained_objects}: \text{Ptr}[\text{SubObjArr}], \text{State} \rightarrow \text{Set}[\text{TypeTaggedObject}],
\text{objectsInRange}: \text{Ptr}[\text{SubObjArr}], \text{int}, \text{int} \rightarrow \text{Set}[\text{TypeTaggedObject}],
\text{objectsUpTo}: \text{Ptr}[\text{SubObjArr}], \text{int} \rightarrow \text{Set}[\text{TypeTaggedObject}]
exempting
\forall p: \text{Ptr}[\text{SubObjArr}]
eval(p,\text{bottom}), \text{eval}(p,\text{emptyState})

For a pointer to a member (declared using the ::* operator), the following trait is used. It is just like the trait for \text{Pointer}, except it uses sorts of the form \text{PtrMbr}[\text{Obj}[T]] instead of \text{Ptr}[\text{Obj}[T]].

% @(#)$Id: PointerToMember.lsl,v 1.1 1995/12/23 03:06:03 leavens Exp $% Pointers to members that are single objects
% (perhaps within an array, but not to arrays)
PointerToMember(Loc, Val): trait
  includes Pointer(Loc, Val, PtrMbr for Ptr)

For pointers to members that are arrays, the following trait is used.
% @(#)Id: PointerToMemberArray.lsl,v 1.1 1995/12/23 03:07:28 leavens Exp $
% Pointers to members that are single objects
% (perhaps within an array, but not to arrays)
PointerToMemberArray(Loc, Val): trait
  includes PointerToArray(Loc, Val, PtrMbr for Ptr)

When a pointer type is used in a Larch/C++ specification, an instantiation of one or more of the traits above is automatically used in the specification. For example, when a type such as the one in the following typedef

typedef T *tpointer;

or the equivalent formal parameter type T [], is mentioned in a Larch/C++ specification, the trait Pointer(Obj, T) will be implicitly used in the interface specification module in which it appears. This also applies to declarations of global variables and formal parameters, as shown in the following table.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Used Traits (not counting MutableObj and ConstObj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *ip;</td>
<td>Pointer(Obj, int)</td>
</tr>
<tr>
<td>int ip[];</td>
<td>Pointer(Obj, int)</td>
</tr>
<tr>
<td>const int *ip;</td>
<td>Pointer(ConstObj, int)</td>
</tr>
<tr>
<td>const int ip[];</td>
<td>Pointer(ConstObj, int)</td>
</tr>
<tr>
<td>int *const ip;</td>
<td>Pointer(Obj, int)</td>
</tr>
<tr>
<td>int ** ip;</td>
<td>Pointer(Obj,.Ptr[Obj[int]]), Pointer(Obj, int)</td>
</tr>
<tr>
<td>int **x[10];</td>
<td>Array(Obj,.Ptr[Obj[int]]), Pointer(Obj, int)</td>
</tr>
<tr>
<td>int *const x[10];</td>
<td>Array(ConstObj, Ptr[Obj[int]])</td>
</tr>
<tr>
<td>int (*x)[10];</td>
<td>PointerToMemberArray(Arr[Obj[int]], Arr[int]),</td>
</tr>
<tr>
<td></td>
<td>Array(Obj, int)</td>
</tr>
<tr>
<td>const int x[][10];</td>
<td>PointerToArray(Arr[ConstObj[int]], Arr[int]),</td>
</tr>
<tr>
<td></td>
<td>Array(ConstObj, int)</td>
</tr>
<tr>
<td>int (*x)[10][20];</td>
<td>PointerToArray(Arr[Arr[Obj[int]]], Arr[Arr[int]]),</td>
</tr>
</tbody>
</table>
|                   | Array(Obj, int),
|                   | MultiDimensionalArray(Arr[Obj[int]], Arr[int]])   |

When the specification mentions a pointer to member type, the traits used are like those above, except that each use of Pointer above is changed to PointerToMember, and each use of PointerToArray is changed to PointerToMemberArray.
11.9 Character Strings

Character strings are not really a type in C++, but merely a convention for using character arrays in which a null (0) character is found. To help state the pre- and postconditions of functions that deal with C++ strings, Larch/C++ provides the following LSL trait. Unlike the other traits in this chapter, the traits in this section are not implicitly used by Larch/C++. You have to write a uses clause to use one or more of them. (see Section 9.2 [The Uses Clause], page 255 for how to use a trait).

The following trait can be used with the C++ types char * and the corresponding array types. Recall that the abstract values of pointers of type char * have sort Ptr[Obj[char]] (see Section 11.8 [Pointer Types], page 287). Recall that the abstract values of arrays of characters have sort Arr[Obj[char]] (see Section 11.7 [Array Types], page 279). However, recall that, when used in a formal parameter declaration, both char * and char [] stand for the sort Ptr[Obj[char]] (see Section 6.1.8.1 [Sorts for Formal Parameters], page 94).

% $Id: cpp_char_string.lsl,v 1.25 1995/11/06 16:02:59 leavens Exp $
% Compare Guttag and Horning’s book (Springer-Verlag, 1994), page 64
cpp_char_string: trait
    includes int, char, TypedObj(Obj, char),
        Pointer(Obj, char),
        String(char, String[char], int for Int)
    introduces
        null: -> char
        nullTerminated: Ptr[Obj[char]], State -> Bool
        nullTerminated: Arr[Obj[char]], State -> Bool
        throughNull: Ptr[Obj[char]], State -> String[char]
        throughNull: Arr[Obj[char]], State -> String[char]
        uptoNull: Ptr[Obj[char]], State -> String[char]
        uptoNull: Arr[Obj[char]], State -> String[char]
        sameCharsThroughNull: Ptr[Obj[char]], State, Ptr[Obj[char]], State -> Bool
        sameCharsThroughNull: Arr[Obj[char]], State, Ptr[Obj[char]], State -> Bool
        lengthToNull: Ptr[Obj[char]], State -> int
        lengthToNull: Arr[Obj[char]], State -> int
        objectsToNull: Ptr[Obj[char]], State -> Set[Object]
        objectsToNull: Arr[Obj[char]], State -> Set[Object]
        legalStringIndex: Ptr[Obj[char]], State, int -> Bool
        legalStringIndex: Arr[Obj[char]], State, int -> Bool
asserts
\forall p, p1, p2: Ptr[Obj[char]], st, st1, st2: State, i: int,
a, a1, a2: Arr[Obj[char]]
null == 0;

nullTerminated(p, st) ==
  assigned(p, st) \&\& (eval(*p, st) = null \&\& nullTerminated(p+1, st));
nullTerminated(a, st) == nullTerminated(address_of(a, 0), st);

nullTerminated(p, st)
 => (throughNull(p, st)
   = (if eval(*p, st) = null then {null}
     else eval(*p, st) \precat throughNull(p+1, st)));
throughNull(a, st) == throughNull(address_of(a, 0), st);

nullTerminated(p, st)
 => (uptoNull(p, st)
   = (if eval(*p, st) = null then empty
     else eval(*p, st) \precat uptoNull(p+1, st)));
uptoNull(a, st) == uptoNull(address_of(a, 0), st);

(nullTerminated(p1, st1) \&\& nullTerminated(p2, st2))
 => (sameCharsThroughNull(p1, st1, p2, st2) ==
     (throughNull(p1, st1) = throughNull(p2, st2)));
sameCharsThroughNull(a1, st1, p2, st2) ==
sameCharsThroughNull(address_of(a1, 0), st1, p2, st2);
sameCharsThroughNull(p1, st1, a2, st2) ==
sameCharsThroughNull(address_of(a1, 0), st1, address_of(a2, 0), st2);
sameCharsThroughNull(a1, st1, a2, st2) ==
sameCharsThroughNull(address_of(a1, 0), st1, address_of(a2, 0), st2);

nullTerminated(p, st)
 => (lengthToNull(p, st) = len(throughNull(p, st)) - 1);
lengthToNull(a, st) == lengthToNull(address_of(a, 0), st);

nullTerminated(p, st)
 => (objectsToNull(p, st) = (if eval(*p, st) = null then {widen(*p)}
      else {widen(*p)} \ U objectsToNull(p+1, st)));
objectsToNull(a, st) == objectsToNull(address_of(a, 0), st);

nullTerminated(p, st)
 => (legalStringIndex(p, st, i) = (0 <= i \&\& i < lengthToNull(p, st)));
legalStringIndex(a, st, i) == legalStringIndex(address_of(a, 0), st, i);

implies converts null, nullTerminated: Ptr[Obj[char]], State -> Bool,
nullTerminated: Arr[Obj[char]], State -> Bool
Chapter 11: Built-in Types

The above trait also gives a good example of when one needs to use states explicitly in a trait. The trait function `nullTerminated`, for example, can only tell if a null character is in a string by examining the abstract values of the character objects in a given state. See Section 2.8.2 [Formal Model of States], page 30 for more about states. See Section 2.8.1 [Formal Model of Objects], page 24 for more about the trait `TypedObj` that defines `eval`.

A trait that renames some of the sorts used in the `cpp_string` trait is given below. This trait is useful when dealing with the C++ types `const char *` and the corresponding array types. Recall that the abstract values of such pointers have sort `Ptr[ConstObj[char]]`, and the abstract values of the corresponding arrays have sort `Arr[ConstObj[char]]`.

```plaintext
% @(#)$Id: cpp_const_char_string.lsl,v 1.5 1995/01/04 03:16:31 leavens Exp $
cpp_const_char_string: trait
    includes cpp_char_string(ConstObj for Obj)
```

As an example of how to use these traits, the following is the specification of a `strcpy` function, which copies the characters in `s2` into `s1`. (The following specification may or may not specify `strcpy` from any particular standard or library.)

```plaintext
// @(#)$Id: strcpy.lh,v 1.12 1997/06/03 20:30:21 leavens Exp $
extern char* strcpy(char *s1, const char *s2) throw();
//@ behavior {
//@ uses cpp_char_string, cpp_const_char_string;
//@ requires nullTerminated(s2, pre)
//@ /
//@ allocatedUpTo(s1, lengthToNull(s2, pre) + 1, pre);
//@ modifies objectsInRange(s1, 0, lengthToNull(s2, pre));
//@ ensures result = s1 /
//@ nullTerminated(s1,post)
//@ /
//@ sameCharsThroughNull(s1, post, s2, pre);
//@ }
```

The Larch/C++ release also contains a trait `cpp_unsignedChar_string` for dealing with the C++ types `unsigned char *` and `unsigned char []` as strings. There is also a trait `cpp_wchar_t_string` for dealing with the C++ types `wchar_t *` and `wchar_t []` as strings. Both of these traits are similar to `cpp_const_string`. The following trait can be used to include all the relevant traits.
11.10 Structure and Class Types

A struct (the C++ keyword for structure) or class is automatically regarded as a collection of objects of various types (See Sections 3.6.2 and 9 of [Ellis-Stroustrup90]). This interpretation can be overridden by the specifier, who may provide a trait to define the abstract values by hand (see Chapter 7 [Class Specifications], page 167 for details). However, if the user does not supply a trait that defines a sort with the same name as the struct or class, then Larch/C++ will provide one automatically, as described in this section. In the rest of this section, we assume that the user has not provided such a trait.

Each object in a structure is denoted by a member name in C++. The corresponding idea in the LSL values is called a field name. For example, the following declarations

```cpp
struct Entry { char *sym; int val; };
Entry e;
```

declare a struct variable e of type Entry with two members. See Section 5.4.4 [Structure and Class Declarations], page 69 for more details on the sort of e in such a declaration. We now describe the abstract values that are automatically specified for structs and classes.

The automatically constructed abstract value is a fixed-length tuple with fields and sorts as appropriate. For each structure type declaration, the trait defining its abstract model is implicitly used in any specification module in which the declaration appears. For example, the above declaration of the type Entry is modeled by the following LSL trait.

```idl
Entry_Trait: trait
  assumes char, Pointer(Obj, char), int
  includes MutableObj(Ptr[Obj[char]]), MutableObj(int),
      ConstObj(Ptr[Obj[char]]), ConstObj(int)
  includes Entry_Pre_Trait(Entry, Obj, Val[Entry]),
      Entry_Pre_Trait(Const[Entry], ConstObj, Val[Entry])
```
In the above trait, the assumed traits are those included automatically by Larch/C++ to model the types explicitly mentioned in the declaration. The first set of included traits models the fields both for normal (see Section 5.4.4 [Structure and Class Declarations], page 69) and constant declarations see Section 5.4.7 [Constant Declarations], page 75). In this example, the traits assumed for the field types are an char (see Section 11.1.4 [Char Trait], page 272), an instantiation of Pointer (see Section 11.8 [Pointer Types], page 287), and int (see Section 11.1 [Integer Types], page 269). See Section 2.8.1 [Formal Model of Objects], page 24 for the traits MutableList, and ConstList.

The trait Entry_Trait defines three sorts: Entry, Const[Entry], and Val[Entry]. Since the theories of Entry and Const[Entry] are nearly identical, they are defined by including the trait Entry_Pre_Trait (given below) with two different renamings.

The sort Entry is used for the abstract values of global variables objects of type Entry. The sort Const[Entry] is used for the abstract values of global variables objects of type const Entry. The sort Val[Entry] is the abstract values of value parameters of type Entry (or type const Entry). The sort Entry is a tuple of two objects, the sort Const[Entry] is a tuple of two constant objects, while the sort Val[Entry] has no contained objects.

The following is the trait Entry_Pre_Trait.

% @(#) $Id: Entry_Pre_Trait.lsl,v 1.1 1998/08/27 15:11:53 leavens Exp $

Entry_Pre_Trait(Entry, Loc, Val): trait

assumes char, Pointer(Obj, char), int,
TypedObj(Loc, Ptr[Obj[char]]), TypedObj(Loc, int),
contained_objects(Loc[Ptr[Obj[char]]]), contained_objects(Loc[int])

includes NoContainedObjects(Val), contained_objects(Entry)

Entry tuple of sym: Loc[Ptr[Obj[char]]], val: Loc[int]
Val tuple of sym: Ptr[Obj[char]], val: int

introduces
  eval: Entry, State -> Val
  allocated, assigned: Entry, State -> Bool
asserts
\[ \forall \text{entry: Entry, opoc: Loc[Ptr[Obj[char]]]}, \, \oi: \text{Loc[int]}, \, \text{st: State} \]
\[
\text{contained_objects([opoc,oi], st)}
\]
\[== \{\text{asTypeTaggedObject(opoc)}\} \cup \{\text{asTypeTaggedObject(oi)}\};\]
\[
\text{eval(entry, st)} == [\text{eval(entry.sym, st)}, \text{eval(entry.val, st)}];\]
\[
\text{allocated(entry, st)}
\]
\[== \text{allocated(entry.sym, st)} \land \text{allocated(entry.val, st)};\]
\[
\text{assigned(entry, st)}
\]
\[== \text{assigned(entry.sym, st)} \land \text{assigned(entry.val, st)};\]

implies

converts

\[
\text{contained_objects: Entry, State} \rightarrow \text{Set[TypeTaggedObject]},\]
\[
\text{eval: Entry, State} \rightarrow \text{Val},\]
\[
\text{allocated: Entry, State} \rightarrow \text{Bool}, \text{assigned: Entry, State} \rightarrow \text{Bool} \]

The tuple notation is a LSL shorthand explained in Chapter 4 of [Guttag-Horning93]. It should not pose any problems, as tuples are built by listing the values, in the order of their declaration in the trait, within square brackets ([ ]). The fields are extracted with the familiar dot (.) notation.

The trait function eval allows state functions (see Section 6.2.1 [State Functions], page 103) to be applied to abstract values of sort Entry or Const[Entry]; it forms a value consisting of the values of the component objects in the given state.

The objects contained in a structure’s abstract value are the fields of the structure. This is described by the trait function contained_objects.

The fields of the automatically-specified abstract value can be referenced by terms using the dot notation (.), as in C++. Note however, that for global variables and reference parameters, a state function must be applied first, to extract the abstract value from the variable. For example, one can write e'.val which has sort Obj[int], but not e.val. The following table shows terms involving the global variable e and their sorts.

<table>
<thead>
<tr>
<th>Term</th>
<th>Sort</th>
<th>Term</th>
<th>Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>Obj[Entry]</td>
<td>e</td>
<td>Entry</td>
</tr>
<tr>
<td>e'.sym</td>
<td>Obj[Ptr[Obj[char]]]</td>
<td>e'.sym</td>
<td>Ptr[Obj[char]]</td>
</tr>
<tr>
<td>e'.val</td>
<td>Obj[int]</td>
<td>e'.val</td>
<td>int</td>
</tr>
<tr>
<td>e</td>
<td>Entry</td>
<td>e</td>
<td>Val[Entry]</td>
</tr>
</tbody>
</table>
See Section 6.1.8.1 [Sorts for Formal Parameters], page 94 for a discussion of the sort of a formal parameter structure (and its fields).

Note that a structure or class type name in C++ is used as a sort name in Larch/C++. This is so Larch/C++ mimics C++ by-name type checking for structure and class types.

There is no distinction made between the public, protected, and private members of a structure or class for purposes of automatically defining a trait for their abstract values.

Note that if the structure or class has member functions, they are also part of the automatically constructed abstract value. See Section 11.12 [Function Types], page 305 for a discussion of their abstract values. See Section 7.1.1 [A First Class Design (Person)], page 168 for an example of such a trait.

11.11 Union Types

In C++, a union is “a structure whose member objects all begin at offset zero (i.e., member objects overlap in memory) and whose size is sufficient to contain any of its member objects” (Section 9.5 of [Ellis-Stroustrup90]). For example, the following declarations

```c
union U {int i_var; char *char_p;};
U x;
```

declare a union variable `x` of type `U` with two members. See Section 5.4.5 [Union Declarations], page 72 for more details on the sort of `x` in such a declaration. In this section we describe the automatically-constructed trait that models the abstract values of unions. (This model can be overridden by the specifier, who may provide a trait to define the abstract values. This is done by using a trait which has a sort with the same name as the union’s sort. See Chapter 7 [Class Specifications], page 167 for details. In the rest of this section, we assume that the user has not provided such a trait.)

The abstract value of a C++ union is a tagged discriminated union, with field names and sorts for its declaration. All the fields of a union that are simple variables are modeled by typed locations that widen to the same untyped object (See Section 2.8.1 [Formal Model of Objects], page 24). That is, changing the value in one location (e.g., in the field `i_var`) may change the value in the other locations (e.g., in the field `char_p`). This is just what would happen in C++, and hence each location depends on the others [Chalin95].
For each union type declaration, the trait defining its abstract model is implicitly used in any specification module in which the declaration appears. For example, the above declaration of the type $U$ is modeled by the following LSL trait.

```lsl
% @(#)$Id: U_Trait.lsl,v 1.17 1997/07/31 21:14:02 leavens Exp $

U_Trait: trait
  assumes int, Pointer(Obj, char)
  includes MutableObj(int), MutableObj(Ptr[Obj[char]]),
    ConstObj(int), ConstObj(Ptr[Obj[char]])
  includes U_Pre_Trait(U, Obj, Val[U]),
    U_Pre_Trait(Const[U], ConstObj, Val[U])
```

In the above trait, the assumed traits are those included automatically by Larch/C++ to model the types explicitly mentioned in the declaration. The first set of included traits models the “fields” both for normal (see Section 5.4.5 [Union Declarations], page 72) and constant declarations see Section 5.4.7 [Constant Declarations], page 75). In this example, the traits assumed for the field types are an `char` (see Section 11.1.4 [Char Trait], page 272), an instantiation of `Pointer` (see Section 11.8 [Pointer Types], page 287), and `int` (see Section 11.1 [Integer Types], page 269). See Section 2.8.1 [Formal Model of Objects], page 24 for the traits `MutableObj`, and `ConstObj`.

The trait $U_Trait$ defines three sorts: $U$, $Const[U]$, and $Val[U]$. Since the theories of $U$ and $Const[U]$ are nearly identical, they are defined by including the trait $U_Pre_Trait$ (given below) with two different renamings.

The sort $U$ is used for the abstract values of global variables objects of type $U$. The sort $Const[U]$ is used for the abstract values of global variables objects of type $const U$. The sort $Val[U]$ is the abstract values of value parameters of type $U$ (or type $const U$). The sort $U$ is a LSL union of two locations, both of which share the same untyped object. The sort $Const[U]$ is a LSL union of two constant locations, both of which share the same untyped object. The sort $Val[U]$ has no contained objects.

The following is the trait $U_Pre_Trait$.

```lsl
% @(#)$Id: U_Pre_Trait.lsl,v 1.2 1997/07/31 21:14:14 leavens Exp $

% Improved in response to a criticism of Chalin's [Chalin95]

U_Pre_Trait(U, Loc, Val): trait
  assumes int, Pointer(Obj, char),
    TypedObj(Loc, int), TypedObj(Loc, Ptr[Obj[char]]),
    contained_objects(Loc[int]), contained_objects(Loc[Ptr[Obj[char]])
```
includes NoContainedObjects(Val),
  contained_objects(U)

U union of i_var: Loc[int], char_p: Loc[Ptr[Obj[char]]]
Val union of i_var: int, char_p: Ptr[Obj[char]]

introduces
  eval: U, State -> Val
  allocated, assigned: U, State -> Bool

asserts
\forall st: State, u: U, s: Loc[Ptr[Obj[char]]], i: Loc[int]

  % both tags share the same untyped object
  widen(i_var(i).char_p) == widen(i_var(i).i_var);
  widen(char_p(s).i_var) == widen(char_p(s).char_p);

  % a union with a given tag contains the object tagged with each sort
  contained_objects(i_var(i), st)
    == {asTypeTaggedObject(i)}
    \U {asTypeTaggedObject(narrow(widen(i)):Loc[Ptr[Obj[char]]])};
  contained_objects(char_p(s), st)
    == {asTypeTaggedObject(s)}
    \U {asTypeTaggedObject(narrow(widen(s)):Loc[int])};

  eval(i_var(i), st) == i_var(eval(i, st));
  eval(char_p(s), st) == char_p(eval(s, st));

  allocated(i_var(i), st) == allocated(i, st);
  allocated(char_p(s), st) == allocated(s, st);
  assigned(i_var(i), st) == assigned(i, st);
  assigned(char_p(s), st) == assigned(s, st);

implies
converts
  eval: U, State -> Val,

The shorthand notation for union definitions in LSL traits is explained in Chapter 4 of [Guttag-Horning93]. The notation for referring to a “field” of a union is the same as in C++, but there is also a way to create a union with a particular tag; for example, i_var(i) creates a U value with tag i_var.

[![Fields that are arrays overlap as dictated by C++. An example might be good here.]]}
The parts of a union abstract value can be referenced by terms using the dot notation (.), as in C++. Note however, that for global variables and reference parameters, a state function must be applied first, to extract the abstract value from the variable (see Section 6.2.1 [State Functions], page 103). For example, one can write \( x'.i\_\text{var} \) which has sort \( \text{Obj}[\text{int}] \), but not \( x.i\_\text{var} \). A state function cannot be applied to a union abstract value, because it is not clear what tag should apply. The following table shows terms involving the global variable \( x \) and their sorts. Note that both of the last two lines will be defined in every state, although the exact values obtained by changing the value with one type and reading it out using another are not specified.

<table>
<thead>
<tr>
<th>Terms</th>
<th>Sorts</th>
<th>Terms</th>
<th>Sorts</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( \text{Obj}[\text{U}] )</td>
<td>( x' )</td>
<td>( \text{U} )</td>
</tr>
<tr>
<td>( x.i_\text{var} )</td>
<td>( \text{Obj}[\text{int}] )</td>
<td>( \text{tag}(x') )</td>
<td>( \text{U}_\text{tag} )</td>
</tr>
<tr>
<td>( x.\text{char}_p )</td>
<td>( \text{Obj}[\text{Ptr}[\text{Obj}[\text{char}]]] )</td>
<td>( x'.\text{char}_p' )</td>
<td>( \text{Ptr}[\text{Obj}[\text{char}]] )</td>
</tr>
</tbody>
</table>

### 11.12 Function Types

The abstract model of a C++ function is given by the sort \( \text{cpp\_function} \) in the following trait. This model is very abstract, because the real semantics of C++ functions is given by the function specification mechanisms of Larch/C++ (see Chapter 6 [Function Specifications], page 82).

```lsl
% @(#)$Id: cpp_function.lsl,v 1.5 1997/06/03 20:49:42 leavens Exp $ % C++ functions
cpp_function(TYPE): trait
    includes List(TYPE, List[TYPE]), NoContainedObjects(cpp_function)
    introduces
      returnType: cpp_function -> TYPE
      argumentTypes, exceptionTypes: cpp_function -> List[TYPE]
```

The sort \( \text{TYPE} \) in the trait above stands for a Larch/C++ sort. But the trait functions are more suggestive than useful. See Section 6.13 [Specifying Higher-Order Functions], page 157 for how to compare functions against Larch/C++ function specifications, and for how to specify functions that take or return pointers to C++ functions.

Member functions are modeled in Larch/C++ as values of the sort \( \text{cpp\_member\_function} \) in the following trait.

```lsl
```
% @(#)$Id: cpp_member_function.lsl,v 1.3 1995/12/23 02:51:47 leavens Exp $  
% C++ member functions 

cpp_member_function(TYPE): trait 
  includes cpp_function(cpp_member_function for cpp_function) 
  introduces 
    selfType: cpp_member_function -> TYPE 
    isConst, isVolatile: cpp_member_function -> Bool
Appendix A  Grammar Summary

The following is a summary of the context-free grammar for Larch/C++. See Chapter 3 [Syntax Notation], page 35 for the notation used. In the first section below, grammatical productions are to be understood lexically. That is, no white space (see Section 4.1 [White Space], page 36) may intervene between the characters of a token.

A.1 Lexical Conventions

```plaintext
microsyntax ::= lexeme [ lexeme ] ...  
lexeme ::= white-space | comment | annotation-marker 
    | pragma | token  
white-space ::= non-nl-white-space | newline  
non-nl-white-space ::= a blank, tab, vertical tab, carriage return, or formfeed character  
newline ::= a newline character  
comment ::= C-style-comment | C++-style-comment  
C-style-comment ::= /* [ C-style-body ] C-style-end  
| stars-non-slash [non-star-slash] ...  
non-star-slash ::= non-star 
| stars-non-slash  
stars-non-slash ::= * [ * ] ... non-slash  
non-at-star ::= any character except @ or *  
non-star ::= any character except *  
non-slash ::= any character except /  
C-style-end ::= [ * ] ... */  
C++-style-comment ::= // newline 
    | // non-at-newline [ non-newline ] ... newline  
non-newline ::= any character except a newline  
non-at-newline ::= any character except @ or newline  
annotation-marker ::= //@ | /*@ | @*/  
pragma ::= # non-nl-white-space pragma [ non-newline ] ...  
| _attribute_ [ non-semi-newline ] ...  
| _asm_ | _const_ | _inline_  
| _signed_ | _typeof_ | _volatile_  
| _extension_  
non-semi-newline ::= any character except a semicolon (;) or newline  
token ::= identifier | simple-id  
    | keyword | context-dependent-keyword  
    | special-symbol | predicate-keyword  
    | informal-comment | literal | lsl-constant  
identifier ::= letter | letter-or-digit | ident( letter [ letter-or-digit ] ... )  
letter ::= _, a through z, or A through Z
```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter-or-digit ::= letter | digit
identifier ::= letter [ letter-or-digit ] ...
keyword ::= term-only-keyword | stmt-exp-only-keyword
  | always-keyword
term-only-keyword ::= fresh | informally | liberally
  | on | nothing | post
  | pre | redundantly | result
  | returns | self | then
  | thrown | throws | trashed
  | unchanged
stmt-exp-only-keyword ::= break | case | catch
  | const_cast | continue | default
  | delete | do | dynamic_cast
  | goto | new | reinterpret_cast
  | return | static_cast | switch
  | throw | try | typeid
  | while
always-keyword ::= abstract | accesses | also
  | any | asm | assert
  | auto | be | behavior
  | by | calls | char
  | class | constraint | constructs
  | const | depends | double
  | else | ensures | enum
  | everything | example | expects
  | explicit | extern | float
  | for | friend | if
  | inline | int | invariant
  | is | let | long
  | modifies | mutable | namespace
  | operator | private | program
  | protected | public | reach
  | refine | register | represents
  | requires | satisfies | short
  | signature | signed | simulates
  | sizeof | spec | static
  | struct | template | this
  | throw | trashes | typedef
  | typename | union | unsigned
  | uses | using | virtual
  | void | volatile | wchar_t
  | weakly | where | with
context-dependent-keyword ::= typedef-non-class-or-enum-name
  | typedef-class-name | typedef-enum-name
  | original-namespace-name | namespace-alias-name
  | original-class-name | original-enum-name
  | template-non-class-name | template-class-name
typedef-non-class-or-enum-name ::= identifier
typedef-class-name ::= identifier
typedef-enum-name ::= identifier
original-namespace-name ::= identifier
namespace-alias-name ::= identifier
original-class-name ::= identifier
original-enum-name ::= identifier
template-non-class-name ::= identifier
template-class-name ::= identifier
special-symbol ::= always-special-symbol
   | C++-decl-symbol | C++-operator-symbol
   | predicate-special-symbol | lsl-op
always-special-symbol ::= ( | ) | { | } | [ | ]
   | = | ; | : | :: | ,
   | ? | . | * | ‘…’
C++-operator-symbol ::= < | > | * | & | ~
new-definition ::= new | delete
   | [ non-nl-white-space ] ... [] | delete [ non-nl-white-space ] ... []
   | + | - | * | / | % | ^ | & | |
   | ! | = | < | > | -= | *= | /= | %=
   | ^= | &= | |= | << | >> | >>= | <<= | == | !=
   | <= | >= | && | ||
   | ++ | -- | , | ->* | ->
   | () | []

predicate-symbol ::= ^ | ' | \< | \>

lsl-op ::= \ identifier | star-or-op-char [ op-char ] ...

star-or-op-char ::= * | op-char

op-char ::= − | = | < | > | + | − | /
   | ! | # | $ | & | ? | @ | ‘…’

predicate-keyword ::= \A | \and | \any | \E | \eq | \exists | \forall
   | \implies | \langle | \neq | \obj | \or | \pre | \post | \rangle
   | = | == | != | ! | ^ | \| | \| | =>
informal-comment ::= ( % informal-comment-body | %)
informal-comment-body ::= non-percent-right-paren [ non-percent-right-paren ] ...
non-percent-right-paren ::= non-percent
   | percents-non-right-paren
non-percent ::= any character except %
percents-non-right-paren ::= % | [%] | ... | non-percent
non-right-paren ::= any character except )
literal ::= integer-constant | floating-constant
   | character-constant | string-literal [ string-literal ] ...
   | abstract-string-literal
integer-constant ::= decimal-constant | octal-constant | hex-constant
decimal-constant ::= one-to-nine [ digit ] ... [ integer-suffix ]
one-to-nine ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
octal-constant ::= 0 [ octal-digit ] ... [ integer-suffix ]
octal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex-constant ::= 0 hex-indicator hex-digit [ hex-digit ] ... [ integer-suffix ]
hex-indicator ::= x | X
hex-digit ::= digit | a | b | c | d | e | f | A | B | C | D | E | F
integer-suffix ::= long-suffix | unsigned-suffix
  | long-suffix unsigned-suffix | unsigned-suffix long-suffix
long-suffix ::= l | L
unsigned-suffix ::= u | U
floating-constant ::= fractional-constant [ exponent-part ] [ float-suffix ]
  | digit [ digit ] ... exponent-part [ float-suffix ]
fractional-constant ::= [ digit ] ... digit [ digit ] ...
  | digit [ digit ] ...
exponent-part ::= exponent-indicator [ sign ] digit [ digit ] ...
exponent-indicator ::= e | E
sign ::= - | +
float-suffix ::= f | F | l | L
character-constant ::= [ L ] ' char-const-char '
char-const-char ::= normal-char | std-esc | '
normal-char ::= any character except ' , " , \ or a newline
std-esc ::= \n // newline LF
  | \t // horizontal tab HT
  | \v // vertical tab VT
  | \b // backspace BS
  | \r // carriage return CR
  | \f // form feed FF
  | \a // alert BEL
  | \ // backslash \n \ ? // question mark ?
  | \' // single quote '
  | " // double quote "
  | \ octal-code // octal code o, oo, ooo
  | \x hex-digit [ hex-digit ] ... // hex code xhh...
octal-code ::= octal-digit | octal-digit octal-digit
  | octal-digit octal-digit octal-digit
string-literal ::= [ L ] " [ string-character ] ... "
string-character ::= normal-char | escape-sequence | '
escape-sequence ::= std-esc | non-std-esc
non-std-esc ::= \ non-escape-code
non-escape-code ::= any character that cannot follow \ in a std-esc
abstract-string-literal ::= A " [ string-character ] ... "
lsl-constant ::= decimal-constant | character-constant

A.2 Declarations
declaration ::= [ decl-specifier-seq ] [ init-declarator-list ] ; [ fun-spec-body ]
| [ [ decl-specifier-seq ] ctor-declarator [ fun-spec-body ] ]
| block-declaration
| function-definition
| template-declaration
| explicit-instantiation
| explicit-specialization
| linkage-declaration
| namespace-definition
| refinement-declaration
| extern everything ;
block-declaration ::= asm-definition
| namespace-alias-definition
| using-declaration
| using-directive
init-declarator-list ::= init-declarator [ , init-declarator ] ... 
init-declarator ::= declarator [ initializer ]
initializer ::= = constant-expression | = { initializer-list }
| ( expression-list )
constant-expression ::= exactly as in C++
initializer-list ::= initializer-clause [ , initializer-clause ] ... [ , ]
initializer-clause ::= assignment-expression
| { [ initializer-list ] } 
assignment-expression ::= exactly as in C++
expression-list ::= expression [ , expression ] ... 
expression ::= exactly as in C++
decl-specifier ::= storage-class-specifier
| type-specifier | function-specifier
| friend | typedef
decl-specifier-seq ::= decl-specifier [ decl-specifier ] ... 
storage-class-specifier ::= static | extern | mutable
| auto | register
function-specifier ::= virtual | inline | explicit
type-specifier-seq ::= type-specifier [ type-specifier ] ... 
type-specifier ::= simple-type-name
| enum-specifier | class-specifier
| elaborated-type-specifier | cv-qualifier
cv-qualifier ::= const | volatile
simple-type-name ::= complete-type-name
| [ :: ] nested-name-specifier template template-class-instance
| built-in-type-name
complete-type-name ::= complete-class-name
| complete-non-class-type-name
complete-non-class-type-name ::= [ :: ] [ nested-name-specifier ] non-class-type-name
non-class-type-name ::= enum-name | typedef non-class-or-enum-name
built-in-type-name ::= char | short | int | long | signed | unsigned | float | double
| bool | void
enum-name ::= original-enum-name | typedef-enum-name
enum-class-name ::= [ :: ] [ nested-name-specifier ] class-name
complete-namespace-name ::= [ :: ] [ nested-name-specifier ] namespace-name
nested-name-specifier ::= class-name :: | namespace-name ::
  | nested-name-specifier class-name ::
  | nested-name-specifier namespace-name ::
  | nested-name-specifier template template-class-instance ::
class-name ::= original-class-name | typedef-class-name | template-class-instance
namespace-name ::= original-namespace-name | namespace-alias-name
elaborated-type-specifier ::= class-key [ :: ] [ nested-name-specifier ] identifier
  | typename [ :: ] nested-name-specifier identifier
  | typename [ :: ] nested-name-specifier [ template ] template-class-instance
class-key ::= class | struct | union | signature
denum-specifier ::= enum [ identifier ] { [ enumerator-definition-list ] }
enumerator-definition-list ::= enumerator-definition [ , enumerator-definition ] …
enumerator-definition ::= identifier [ constant-initializer ]
declarator ::= direct-declarator | ptr-operator declarator
direct-declarator ::= id-expression | direct-declarator declarator-qualifier
declarator-qualifier ::= [ [ constant-expression ] ] | param-qualifier
param-qualifier ::= ( [ parameter-declaration-clause ] [ expects-clause ] )
id-expression ::= unqualified-id | qualified-id
unqualified-id ::= identifier
  | operator-function-id | conversion-function-id
  | template-instance
qualified-id ::= nested-name-specifier [ template ] unqualified-id
operator-function-id ::= operator C++-operator-symbol
conversion-function-id ::= operator type-specifier-seq | ptr-operator |
conversion-type-id ::= type-specifier-seq | conversion-declarator |
conversion-declarator ::= ptr-operator | conversion-declarator |
ptr-operator ::= * [ cv-qualifier-seq ] | &
  | [ :: ] nested-name-specifier * [ cv-qualifier-seq ]
cv-qualifier-seq ::= cv-qualifier [ cv-qualifier ] …
abstract-declarator ::= ptr-operator | abstract-declarator
  | direct-abstract-declarator
direct-abstract-declarator ::= | direct-abstract-declarator | declarator-qualifier
  | ( abstract-declarator )
function-definition ::= fun-interface [ fun-spec-body ] | ctor-initializer | fun-body
  | fun-interface [ fun-spec-body ] function-try-block
fun-interface ::= [ decl-specifier-seq ] declarator
  | [ decl-qualifier-seq ] ctor-declarator
  | [ decl-qualifier-seq ] special-function-declarator
decl-qualifier ::= storage-class-specifier | function-specifier | friend | typedef
decl-qualifier-seq ::= decl-qualifier [ decl-qualifier ] …
ctor-initializer ::= : mem-initializer | , mem-initializer | …
mem-initializer ::= mem-initializer-id ( expression-list )
mem-initializer-id ::= complete-class-name | identifier
expression-list ::= expression [ , expression ] …
expression ::= exactly as in C++, but add the following
  | simple-type-name ( [ expression-list ] [ using-trait-list ] )
  | postfix-expression ( [ expression-list ] [ using-trait-list ] )
A.3 Function Specifications

fun-spec-body ::= behavior [ | uses-seq ] [ declaration-seq ] spec-case-seq ]

| behavior program compound-statement
uses-seq ::= uses-clause [ | uses-clause ]...

spec-case-seq ::= spec-case [ also spec-case ]...

spec-case ::= [ let-clause ] req-frame-ens [ example-seq ]

| [ let-clause ] [ requires-clause-seq ] { spec-case-seq } [ ensures-clause-seq ] [ example-seq ]
Appendix A: Grammar Summary

req-frame-ens ::= \[ requires-clause-seq \] ensures-clause-seq
| \[ requires-clause-seq \] frame \[ ensures-clause-seq \]
requires-clause-seq ::= requires-clause requires-clause ...

requires-clause ::= requires \[ redundantly \] pre-cond;
predicate ::= predicate

frame ::= accesses-clause-seq \[ modifies-clause-seq \] \[ trashes-clause-seq \] \[ calls-clause-seq \]
| modifies-clause-seq \[ trashes-clause-seq \] \[ calls-clause-seq \]
| trashes-clause-seq \[ calls-clause-seq \]

ensures-clause-seq ::= ensures-clause \[ ensures-clause \] ...

ensures-clause ::= ensures \[ redundantly \] post-cond;
| ensures \[ redundantly \] liberally post-cond;
predicate ::= term
term ::= if term then term else term
| logical-term

logical-term ::= logical-term logical-opr equality-term
| equality-term

logical-opr ::= \and | \or | \implies | \\ | \\ | \=>
equality-term ::= lsl-op-term \[ eq-opr lsl-op-term \]
| quantifier \[ quantifier \] ... ( term )
| higher-order-comparison
| informal-desc
eq-opr ::= = | == | =\eq | \neq | != | \neq
quantifier ::= quantifier-sym quantified-list
quantifier-sym ::= \A | \forall | \E | \exists
quantified-list ::= varId \[ sort-name \] \[ , varId \[ sort-name \] \] ...

varId ::= identifier

sort-name ::= identifier \[ sort-instance-actuals \]
| class-name \[ built-in-type-name \]
| typedef-non-class-or-enum-name \[ typedef-enum-name \]
sort-instance-actuals ::= \[ sort-or-type \] \[ , sort-or-type \] ...
sort-or-type ::= identifier \[ sort-instance-actuals \] \[ type-id \]
informal-desc ::= informally string-literal \[ string-literal \] ...

informal-comment
lsl-op-term ::= lsl-op \[ lsl-op \] ...
secondary
| secondary \[ lsl-op secondary \] ...
| secondary lsl-op lsl-op ...

secondary ::= primary | primary \[ sc-bracketed \[ : sort-name \] primary \]
sc-bracketed ::= \[ term-list \] \{ \[ term-list \] \}
| \[ term-list \] \} \langle \[ term-list \] \rangle

rangle

term-list ::= term \[ , term \] ...

primary ::= primitive \[ primary-suffix \] ...

primitive ::= ( term )
| varId
| qualified-id
| fcnId \[ term-list \]

primary-suffix ::= selection \[ : sort-name \] state-function
Appendix A: Grammar Summary

**selection ::= . identifier**

**lcpp-primary ::= literal | this | self | result**

| pre | post | any | returns |
| throws ( type-id ) |
| thrown ( type-id ) |
| sizeof ( type-id ) |
| fresh ( term-list ) |
| trashed ( store-ref-list ) |
| unchanged ( store-ref-list ) |

**state-function ::= ^ | \pre | \post | \any | \obj**

**modifies-clause-seq ::= modifies-clause [ modifies-clause ] ...**

**modifies-clause ::= modifies [ redundantly ] store-ref-list ;**

**store-ref-list ::= store-ref [ , store-ref ] ... | nothing | everything**

**store-ref ::= term | reach ( term )**

**trashes-clause-seq ::= trashes-clause [ trashes-clause ] ...**

**trashes-clause ::= trashes [ redundantly ] store-ref-list ;**

**calls-clause-seq ::= calls-clause [ calls-clause ] ...**

**calls-clause ::= calls [ redundantly ] function-names ;**

**function-names ::= everything | nothing**

| function-name [ , function-name ] ... |

**function-name ::= term**

**accesses-clause-seq ::= accesses-clause [ accesses-clause ] ...**

**accesses-clause ::= accesses [ redundantly ] store-ref-list ;**

**let-clause ::= let be-list ;**

**be-list ::= varId : sort-name be term [ , varId : sort-name be term ] ...**

**example-seq ::= example [ example ] ...**

**example ::= example [ liberally ] predicate ;**

**exception-decl ::= throw ( [ type-list ] )**

**type-list ::= type-id [ , type-id ] ...**

**higher-order-comparison ::= lsl-op-term satisfies fun-interface fun-spec-body**

**expects-clause ::= expects expected-trait-list**

**expected-trait-list ::= trait [ simple-id ] [ , trait [ simple-id ] ] ...**

**using-trait-list ::= using trait-or-deref-list**

**trait-or-deref-list ::= trait**

| * simple-id |
| trait-or-deref-list , trait |
| trait-or-deref-list , * simple-id |

**specification-statement ::= fun-spec-body**

| requires-clause |
| accesses-clause |
| modifies-clause |
| trashes-clause |
| calls-clause |
| ensures-clause |
| assert predicate ;**

A.4 Class Specifications
Appendix A: Grammar Summary

```
class-specifier ::= class-head { [ member-seq ] }
class-head ::= class-key [ identifier ] [ base-spec ]
  | class-key nested-name-specifier identifier [ base-spec ]
  | class-key [ nested-name-specifier ] template-class-instance [ base-spec ]
  | abstract class [ nested-name-specifier ] [ identifier ] [ base-spec ]
member-seq ::= [ member-seq ] member-declaration
  | [ member-seq ] access-specifier :
  | [ member-seq ] larch-cpp-clause
larch-cpp-clause ::= uses-clause
  | simulates-clause
  | depends-clause
  | invariant-clause
  | constraint-clause
member-declaration ::= function-definition
  | [ decl-specifier-seq ] [ member-declarator-list ] ; [ fun-spec-body ]
  | [ decl-qualifier-seq ] ctor-declarator ; [ fun-spec-body ]
  | [ decl-qualifier-seq ] special-function-declarator ; [ fun-spec-body ]
  | [ :: ] nested-name-specifier [ template ] unqualified-id
  | using-declaration
  | template-declaration
  | spec-decl
  | refinement-member-decl
member-declarator-list ::= member-declarator [ , member-declarator ] ...
member-declarator ::= declarator [ constant-initializer ]
  | [ identifier ] : constant-expression
constant-initializer ::= = constant-expression
access-specifier ::= public | protected | private
invariant-clause ::= invariant [ redundantly ] predicate ;
constraint-clause ::= constraint [ redundantly ] predicate [ constrained-set ] ;
constrained-set ::= for fun-interface-list
fun-interface-list ::= fun-interface [ , fun-interface ] ...
  | nothing | everything
depends-clause ::= depends store-ref on store-ref-list ;
represents-clause ::= represents store-ref by predicate ;
base-spec ::= ; base-list
base-list ::= base-specifier [ , base-specifier ] ...
base-specifier ::= [ virtual ] [ access-specifier ] complete-class-name
  | access-specifier [ virtual ] complete-class-name
simulates-clause ::= [ weakly ] simulates supertype-list ;
supertype-list ::= supertype [ , supertype ] ...
supertype ::= sort-name [ by fcnId ]

A.5 Template Specifications

template-declaration ::= [ export ] template < template-parameter-list [ expects-clause ] >
```
A.6 Specification Modules

interface ::= top-level [ top-level ] ...
top-level ::= uses-clause | spec-clause | depends-clause
    | represents-clause | invariant-clause | constraint-clause
    | declaration | using-declaration | using-directive
spec-clause ::= spec declaration
uses-clause ::= uses trait-list ;
trait-list ::= trait [ , trait ] ...
trait ::= trait-name [ ( renaming ) ]
trait-name ::= simple-id
renaming ::= replace-list | lsl-sort-list [ , replace-list ]
replace-list ::= replace [ , replace ] ...
replace ::= lsl-sort for lsl-formal
lsl-sort-list ::= lsl-sort [ , lsl-sort ]
lsl-sort ::= simple-id [ lsl-instance-actuals ]
    | built-in-type-name
    | lsl-constant
lsl-instance-actuals ::= [ lsl-sort-list ]
    | < lsl-sort-list >
lsl-formal ::= lsl-sort
    | simple-id [ : lsl-signature ]
lsl-signature ::= [ lsl-sort-list ] -> lsl-sort

A.7 Refinement

refinement-declaration ::= [ refine-prefix ] declaration
refine-prefix ::= refine refinable-id [ with replace-list ] by
refinable-id ::= original-class-name | typedef-class-name
    | class-key identifier
    | unqualified-id
    | template-class-name | template-non-class-name
    | original-namespace-name | namespace-alias-name
refinement-member-decl ::= [ refine-prefix ] member-declaration
Appendix B Bibliography


URL


[Evans96b]

[Goguen84]

[Guaspari-Marceau-Polak90]

[Guttag-Horning-Wing85b]

[Guttag-Horning78]

[Guttag-Horning91b]

[Guttag-Horning93]

[Hall90]

[Hayes93]

[Hesselink92]

[Hoare69]

[Hoare72a]


[Leavens90]

[Leavens91]

[Leavens96]

[Leavens98]

[Ledgard80]

[Leino95]

[Leino95b]

[Lerner91]

[Liskov-Guttag86]

[Liskov-Wing93]
[Liskov-Wing93b]

[Liskov-Wing94]

[Martin-Wing93]

[Meyer88]

[Meyer92]

[Meyer92b]

[Morgan94]

[Morris87]

[Nelson89]

[Parnas72]

[Poetzsch-Heffter97]

[Schmidt86]

[Spivey92]

[Steyaert-etal96]
[Stroustrup91]

[Stroustrup95]


Appendix C Differences

The subsections below detail the differences between Larch/C++ and Larch/C, and between Larch/C++ and C++ itself.

C.1 Differences Between Larch/C++ and Larch/C

Since Larch/C++ is an interface specification language for C++, and Larch/C (LCL, see Chapter 5 of [Guttag-Horning93]) is an interface specification language for C, a fundamental difference is that an implementation of a Larch/C++ specification must be in C++, while an implementation of a LCL specification must be in ANSI C.

The current version of LCL [Evans96b] focuses on a semantic checker that is like the old “lint” for C, but uses specification information to provide enhanced checking. Larch/C++ is evolving towards something similar, but this version of LCL is essentially incompatible with the current version of Larch/C++.

Compared to the version of LCL described in Chapter 5 of [Guttag-Horning93], Larch/C++ has many additions. Instead of listing all the additions here, this section details the incompatibilities between Larch/C++ and the LCL described in [Tan94], and focuses on the changes one would have to make to a LCL specification to make it into a Larch/C++ specification. The following is a list of the most important of these.

- The abstract data type mechanism of LCL is not supported in Larch/C++. Use class specifications (see Chapter 7 [Class Specifications], page 167) instead.
- The semantics of the modifies-clause is different in Larch/C++.
- There is no provision for implicit constraints on names declared as typedefs in Larch/C++.
- The syntax for declaring globals threatened by a function specification is different in LCL than in Larch/C++ (see Section 6.7 [Global Variables], page 134), and no longer required in Larch/C++.
- The syntax -> is not built in as a selection with a defined semantics in Larch/C++ (see Section 6.1.9 [Primary Suffixes], page 98).
- The model for the basic C++ types is different and more detailed in Larch/C++ than in LCL. For example, in Larch/C++ the sort of a formal parameter of type char * is Ptr[Obj[char]] instead of String (see Section 11.8 [Pointer Types], page 287). In general, Larch/C++ models types with abstract values that may contain objects, and does not make any assumptions about how the built-in C++ types are used.
An example of the previous point is the following. If \( x \) is a struct declared as a global variable, the meaning of applying a state-function (See Section 6.2.1 [State Functions], page 103) to \( x \) is different. In LCL, \( x^- \) denotes a tuple of values. In Larch/C++, \( x^- \) denotes a tuple of objects. This allows one to say \( x^- . \text{foo} \) to mean the object that is the field named \text{foo} in the post-state, for example in the modifies clause. (In LCL it may be that the name of a global struct is not considered an object, whereas it is in Larch/C++, see Section 11.10 [Structure and Class Types], page 299.) In Larch/C++, one can write \( x^-^- \) to mean what is meant by \( x^- \) in LCL.

As another example, the abstract values of a union object in Larch/C++ are modeled in such as way as to mimic the sharing of storage in a C++ union (see Section 11.11 [Union Types], page 302).

- Larch/C++ currently supports only the “procedure claims” of LCL in its redundant postconditions; other kinds of LCL claims are not supported and the syntax is different.
- The keyword \text{constraint} is used differently in Larch/C++, where it means the history constraints of [Liskov-Wing94].
- There are various differences in the built-in traits of Larch/C++ and LCL.
- The \text{checks} clause of LCL is not in Larch/C++.

C.2 Differences Between Larch/C++ and C++

It is crazy to compare C++, which is a programming language, and Larch/C++, which is a specification language. It would be like comparing apples and writing (about apples). Nevertheless, Larch/C++ attempts to parse all valid C++ constructs. However, Larch/C++ does not make most of the semantic checks required for a C++ program. Thus, the Larch/C++ parser will parse many improper C++ programs without issuing any error messages. Some of what seem like differences in the syntax from [Stroustrup91] and [Ellis-Stroustrup90] reflect the coming C++ standard [Stroustrup95]. These differences should all be upward compatible with older C++ except that the "implicit int" rule of C in declarations is no longer supported in the new standard.
Appendix D  Deprecated

The subsections below briefly describe the deprecated features of Larch/C++. A feature is deprecated if it is supported in the current release, but slated to be removed from a subsequent release. Such features should not be used.

D.1 Deprecated Syntax

The following syntax is deprecated. (Note that incompatible changes and syntax that is no longer supported is not included in this list.)

\[
\begin{align*}
\text{req-frame-ens} & ::= [ \text{requires-clause} ] [ \text{assuming-clause} ] [ \text{frame} ] \text{ensures-clause} \\
\text{assuming-clause} & ::= \text{assuming pre-cond} ; \\
\text{spec-case} & ::= [ \text{let-clause} ] \text{req-frame-ens} [ \text{example-seq} ] \text{claims-seq} \\
& \quad | [ \text{let-clause} ] [ \text{requires-clause} ] [ \text{assuming-clause} ] \{ \text{spec-case-seq} \} [ \text{example-seq} ] \text{claims-seq} \\
\text{claims-seq} & ::= \text{claims-clause} [ \text{claims-clause} ] \ldots \\
\text{claims-clause} & ::= \text{claims} [ \text{liberally} \mid \text{predicate} ] ;
\end{align*}
\]

To update your specifications: change assuming to requires redundantly, move each claims-seq to just after the existing ensures-clause in the same spec-case and change claims to ensures redundantly and move them to before any examples in the same spec-case.

The \verb|ident{}X| of identifier, the pure virtual function-specifier, the use of multiple spec-cases without also, and the use of informal-descs for a trait, which were previously deprecated, and have now been removed. If you still have these in your specifications, use \verb|ident(X)| instead of \verb|ident{}X|, add the keyword also between spec-cases, and use the C++ = 0 instead of pure virtual. If you have uses informally in your code, change that to the use of a trait. To specify a trait for a class, struct, or union and prevent Larch/C++ from automatically constructing a trait for it, you need to just specify a signature for some trait function that takes or returns the name of the class, struct, or union. It is often enough just to write the following, for a type T.

\[
\text{uses NoContainedObjects}(T)
\]
Example Index

*  
  **=, friend of Money ........................................ 237

A  
abort ......................................................... 154
AbstractString .............................................. 169
AbstractStringTrait trait .................................. 51
add_one ...................................................... 103
add_one.desugared .......................................... 109
AllocatedAssigned trait ..................................... 108
ApplyTwice .................................................. 160
Array trait .................................................. 281
array_add_one ............................................. 105, 114
ArrayAllocAuxFuns trait .................................. 282
ArrayAllocAuxSig trait .................................... 283
ArrayAssignedAuxFuns trait ................................ 283
ArrayForEach ................................................ 162
ArrayIterateProc trait ...................................... 163
ArrayMap .................................................... 158
ArrayMap2 ................................................. 166
cpp_char_string trait ........................................ 296
cpp_const_char_string trait ................................ 298
cpp_function trait ........................................... 305
cpp_member_function trait ................................... 306
cpp_string trait ............................................. 298
cpp_unsignedChar_string trait ................................ 298
cpp wchar_t_string trait ..................................... 298

date, C++ code .................................................. 16
day_of_week_Trait trait ....................................... 279
dealloc_int_obj ............................................... 127
dec_ref ......................................................... 129
declaration ..................................................... 55
decr_ptr ......................................................... 140
default_assignment_op ....................................... 194
default_constructor .......................................... 192
default_copy_ctor ........................................... 193
default_destructor .......................................... 193
default_interfaces ........................................... 192
done_with ...................................................... 128
double trait .................................................. 275

E  
Entry_Pre_Trait trait ......................................... 300
Entry_Trait trait ............................................. 299
Equivalence.lh ................................................ 246
eval ............................................................. 244
Eval_Trait trait ............................................... 245

F  
fact_liberal ................................................... 151
fact_liberal2 .................................................. 153
FactorialTrait trait .......................................... 152
float trait ..................................................... 275
Floor ............................................................ 178
FLT_EPSILON .................................................. 275
FLT_MAX ........................................................ 275
FLT_MIN ........................................................ 275
follows trait ........................................... 203
FreshSemantics trait ................................ 125

H
HasMembership trait .................................. 243
HistoryConstraint trait ................................ 202

I
i_pred trait ........................................... 199
ident macro ........................................... 40
IgnoringTypeTags trait ................................. 119
inc_counter ............................................ 19
inc2 ...................................................... 149
inc3 ...................................................... 151
inc4 ...................................................... 165
int trait .................................................. 272
INT_MAX ................................................. 269, 272
INT_MIN ................................................. 269, 272
interest .................................................... 134
IntHeap ................................................... 2
IntHeapTrait trait ...................................... 4
IntList ................................................... 234
IntSet ................................................... 261
IntSet2 ................................................... 262
IntSetInformal ......................................... 260
IntSetPrivate .......................................... 264
IntSetPrivate2 ......................................... 265
IntVar ................................................... 236
Invariant trait ......................................... 198
Invariant_Visible trait ................................. 199
isqrt ..................................................... 83
isqrt, with claims ...................................... 141
isqrt, with examples .................................. 139
isqrt-informal ......................................... 91
isqrt4 ..................................................... 91

L
linkage_declaration .................................... 80
long trait ................................................. 271
LONG_MAX .............................................. 269, 271
LONG_MIN .............................................. 269, 271
longDouble trait ....................................... 275
lslinit.lsi initialization file ......................... 277

M
make_rat ................................................. 123
make_srat ............................................. 126
make_zero_or_one ..................................... 115
miracle ................................................. 155
ModifiesSemantics trait ............................... 118
Money ..................................................... 179
MoneyBasics trait ...................................... 176
MoneyConstraint trait .................................. 204
MoneyTrait trait ....................................... 177
MTranslation trait ...................................... 204
MultiDimensionalArray trait ......................... 285
MutableMoney .......................................... 224
MutableMoney2 ......................................... 227
MutableMoneyHom trait ................................ 226
MutableMoneyTrait trait .............................. 224
MutableObj trait ....................................... 28

N	namespace_alias ......................................... 79	namespace_definition ................................... 79	new_int ............................................... 110	next_token ............................................. 135
NoContainedObjects trait .............................. 207
NoInformation trait .................................... 150
NoInformationException trait ......................... 150
NoSideEffectsDetFun, trait ............................ 161
NoSideEffectsFun, trait ................................ 159
null ....................................................... 296

O
Overflow ................................................ 150

P
Person ................................................... 171
Person::make_year_older, bad version ............... 198
Person_defaults ....................................... 194
Person_Pre_Trait trait ................................ 173
Person_Trait trait ..................................... 173
Person2 ................................................. 205
PersonInvariant trait .................................. 200
PersonSet .............................................. 210
PersonSetTrait trait ................................... 210
<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PlusAccount</td>
<td>219</td>
</tr>
<tr>
<td>PlusAccount::pay_interest, desugared</td>
<td>221</td>
</tr>
<tr>
<td>Pointer trait</td>
<td>290</td>
</tr>
<tr>
<td>PointerAllocatedAuxFuns trait</td>
<td>292</td>
</tr>
<tr>
<td>PointerAllocAuxSig trait</td>
<td>292</td>
</tr>
<tr>
<td>PointerAssignedAuxFuns trait</td>
<td>293</td>
</tr>
<tr>
<td>PointerToArray trait</td>
<td>293</td>
</tr>
<tr>
<td>PointerToMember trait</td>
<td>294</td>
</tr>
<tr>
<td>PointerToMemberArray trait</td>
<td>294</td>
</tr>
<tr>
<td>PointerWithNull trait</td>
<td>288</td>
</tr>
<tr>
<td>poorly_encrypt</td>
<td>113</td>
</tr>
<tr>
<td>pragmas</td>
<td>38</td>
</tr>
<tr>
<td>PrePointer trait</td>
<td>287</td>
</tr>
<tr>
<td>present_bad</td>
<td>89</td>
</tr>
<tr>
<td>present_good</td>
<td>90</td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>249</td>
</tr>
<tr>
<td>PriorityQueueRequirement trait</td>
<td>249</td>
</tr>
<tr>
<td>PriorityQueueTrait trait</td>
<td>249</td>
</tr>
<tr>
<td>PTranslation trait</td>
<td>200</td>
</tr>
<tr>
<td>PureValue trait</td>
<td>208</td>
</tr>
<tr>
<td>StackError</td>
<td>185</td>
</tr>
<tr>
<td>State trait</td>
<td>33</td>
</tr>
<tr>
<td>State_Basics trait</td>
<td>30</td>
</tr>
<tr>
<td>State_Updates trait</td>
<td>32</td>
</tr>
<tr>
<td>strcpy</td>
<td>298</td>
</tr>
<tr>
<td>SubArray trait</td>
<td>286</td>
</tr>
<tr>
<td>swap</td>
<td>111</td>
</tr>
</tbody>
</table>

T  
- time                         | 255 |
- transfer                     | 136 |
- transfer, desugared          | 136 |
- transfer, with examples      | 138 |
- TrashesSemantics trait       | 131 |
- TypedObj trait               | 25  |
- TypedObjEval trait           | 27  |
- TypePerspectives trait       | 31  |
- TypeTag trait                | 208 |
- TypeTaggedObject trait       | 116 |

U  
- U_Pre_Trait trait            | 303 |
- U_Trait trait                | 303 |
- UCHAR_MAX                    | 273 |
- UINT_MAX                     | 273 |
- ULONG_MAX                    | 273 |
- unsigned trait               | 272 |
- unsignedChar trait           | 273 |
- unsignedInt trait            | 273 |
- unsignedLong trait           | 273 |
- unsignedShort trait          | 273 |
- USHRT_MAX                    | 273 |
- using_declaration            | 79  |
- using_directive              | 80  |

V  
- Val_Array                    | 279 |
- visible trait                | 190 |
- void trait                   | 278 |

W  
- wchar_t trait                | 275 |
- widen1                       | 144 |
<table>
<thead>
<tr>
<th>Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>widen2</td>
<td>145</td>
</tr>
<tr>
<td>widen3</td>
<td>146</td>
</tr>
<tr>
<td>widen4</td>
<td>147</td>
</tr>
<tr>
<td>WidenNarrow trait</td>
<td>26</td>
</tr>
<tr>
<td>with_member_objs trait</td>
<td>242</td>
</tr>
<tr>
<td>withdraw</td>
<td>121</td>
</tr>
<tr>
<td>WithUnassigned trait</td>
<td>26</td>
</tr>
</tbody>
</table>
Concept Index

!  ........................................  44, 45, 276
! = ........................................  44, 46, 87

#  
# ........................................  38, 45
#include ..................................  17

$  
$ ........................................  45

%  
% ........................................  44
% = .......................................  46
% > .......................................  52

&  
& ........................................  44, 45, 65, 66, 108
& & ......................................  44, 276
& = ......................................  44

,  
, ........................................  45, 49, 50, 103

(  
( ......................................  56, 66, 73, 87, 94, 99, 110, 148, 255
( % ......................................  46
( ) ......................................  44, 65

)  
) ......................................  56, 66, 73, 87, 94, 99, 110, 148, 255

*  
* ........................................  44, 45, 65, 66, 112, 157, 290
* in modifies clause ..........................  112
*/ ........................................  37
* = ......................................  44

;  
; ........................................  35, 44, 54, 56, 64, 73, 74, 88, 93, 110, 132, 136, 148,
157, 186, 201, 214, 223, 239, 251, 255

-  
- ........................................  44, 45, 48
-- ......................................  44
- = ......................................  44
- > .......................................  44, 255
\- > * ....................................  44
\- list suffix ................................ 35
\- seq suffix ............................... 35

.  
. ........................................  48, 98
. ........................................  73, 74
. ........................................  35

/  
/ ........................................  44, 45
/ * ......................................  37
/ * */ ....................................  36
/ * @ ....................................  38
// ......................................  35
\ / ......................................  36, 37
\/ ......................................  38
\ = ......................................  44
\ \ ......................................  87

:  
: ........................................  73, 88, 93, 98, 136, 186, 214, 255
:: ......................................  60, 61, 66, 73, 78
::* ......................................  65, 294
:> ......................................  52

;  
; ........................................  54, 78, 81, 82, 110, 127, 132, 133, 136, 138, 164,
186, 196, 201, 213, 223, 243, 255
abstrac
t
abstract . 167, 234
abstract character string . 51
abstract class . 234
abstract data type . 4
abstract program . 164
abstract string constants . 51
abstract value . 4, 30, 103, 115, 168, 269
abstract value, of structure . 167
abstract values, explicitly given by a trait . 181
abstract values, in refinement . 263
abstract values, integer types . 269
abstract-declarator, defined . 66
abstract-declarator, used . 66, 73, 74, 251
abstract-string-literal, defined . 51
abstract-string-literal, used . 47
abstraction, supertype . 8, 215
access . 16
access declaration . 187
access-specifier, defined . 186
access-specifier, used . 186, 214
accesses . 133
accesses-clause, defined . 133
accesses-clause, used . 133, 164
accesses-clause-seq, defined . 133
accessibility . 16
accessible . 16
active . 107
address . 21, 287
ADT . 2
allocated . 107
allocated, for reference parameters . 108
allocated, implicit preconditions . 109
allocation . 123, 188
allocation requirements for reference parameters . 108
also . 82
alternative tokens . 52
always-keyword, defined . 41
always-keyword, used . 40
always-special-symbol, used . 43
and . 52
and . 87
and_eq . 52
annotation . 38
annotation, vs. comment . 36
annotation-marker, defined . 38
annotation-marker, used . 36
anonymous ftp . 10
any . 99, 101
argument type . 239
array . 21, 65, 69, 105, 114, 125, 279
array and pointer, relationship . 95
array formal parameter, and pointer . 95
array traits . 279
array traits, multi-dimensional . 285
array variable . 69
array, constant . 75
array, multi-dimensional . 106
array, objects contained in . 284
arrays, in modifies clause . 112
ASCII . 52
asm . 81
asm-definition, defined . 81
asm-definition, used . 54
assert . 164
assigned . 107
assigned, implicit preconditions . 109
| Assignment operator, implicitly generated | 191 |
| Assignment-expression, see a C++ grammar | 74 |
| Assignment-expression, see a C++ grammar for definition | 56 |
| Assignment-expression, used | 56, 74, 251 |
| Assuming | 328 |
| Assuming-clause, defined | 328 |
| Assuming-clause, used | 328 |
| Assumption | 140 |
| Assumption, redundant precondition | 140 |
| Attribute | 38 |
| Auto | 58 |

**B**
- Base class | 214 |
- Base-list, defined | 214 |
- Base-list, used | 214 |
- Base-spec, defined | 214 |
- Base-spec, used | 167 |
- Base-specifier, defined | 214 |
- Base-specifier, used | 214 |
- Be | 136 |
- Be-list, defined | 136 |
- Be-list, used | 136 |
- Behavior | 82 |
- Behavior program | 164 |
- Behavior program, example | 165 |
- Behavior program, vs. higher-order comparisons | 166 |
- Behavioral interface specification | 2, 15 |
- Behavioral subtype | 8, 215, 223 |
- Behavioral subtype, strong | 231 |
- Behavioral subtype, strong vs. weak | 231 |
- Benefits, of Larch/C++ | 6 |
- Bitand | 52 |
- Bitor | 52 |
- Blank | 36 |
- Block | 18 |
- BNF | 35 |
- Bool | 276 |
- Bool | 276 |
- Boolean trait | 276 |
- Boolean type | 276 |
- Brace | 93 |
- Bracket | 93 |
- Built in type | 60 |
- Built-in type | 269 |
- Built-in-type-name, defined | 60 |
- Built-in-type-name, used | 60, 88 |
- By | 213, 223, 257 |

**C**
- C-style-body, defined | 37 |
- C-style-body, used | 37 |
- C-style-comment, defined | 37 |
- C-style-comment, used | 37 |
- C-style-end, defined | 37 |
- C-style-end, used | 37 |
- C++ | 1 |
- C++ compiler | 10 |
- C++ const string, model of | 298 |
- C++ function, abstract value of | 305 |
- C++ operator symbols | 44 |
- C++ reserved words used in Larch/C++ | 41 |
- C++ string, model of | 296 |
- C++ types, specification of | 269 |
- C++-decl-symbol, used | 43 |
- C++-operator-symbol, defined | 44 |
- C++-operator-symbol, used | 43, 66 |
- C++-style-comment, defined | 37 |
- C++-style-comment, used | 37 |
- Calls | 132 |
- Calls-clause, defined | 132 |
- Calls-clause, used | 132, 164 |
- Calls-clause-seq, defined | 132 |
- Calls-clause-seq, used | 82 |
- Carriage return | 36 |
- Case analysis | 143, 153 |
- Catch | 73 |
- Char | 49, 60, 269, 272 |
- Char * | 50 |
- Char * strings | 296 |
- Char [] strings | 296 |
- Char, wide | 275 |
- Char-const-char, defined | 49 |
- Char-const-char, used | 49 |
- Char[] | 50 |
<table>
<thead>
<tr>
<th>Concept</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>49</td>
</tr>
<tr>
<td>character escape sequence</td>
<td>49</td>
</tr>
<tr>
<td>character string</td>
<td>50, 296</td>
</tr>
<tr>
<td>character string, abstract</td>
<td>51</td>
</tr>
<tr>
<td>character trait</td>
<td>272</td>
</tr>
<tr>
<td>character, wide</td>
<td>49</td>
</tr>
<tr>
<td>character-constant, defined</td>
<td>49</td>
</tr>
<tr>
<td>character-constant, used</td>
<td>47, 51</td>
</tr>
<tr>
<td>characters</td>
<td>269, 272</td>
</tr>
<tr>
<td>characters, wide</td>
<td>275</td>
</tr>
<tr>
<td>checker, for Larch/C++</td>
<td>9, 12</td>
</tr>
<tr>
<td>checker, for LP, obtaining</td>
<td>11</td>
</tr>
<tr>
<td>checker, for LSL</td>
<td>11</td>
</tr>
<tr>
<td>checker, for LSL, obtaining</td>
<td>11</td>
</tr>
<tr>
<td>claim, redundant postcondition</td>
<td>141</td>
</tr>
<tr>
<td>claims</td>
<td>328</td>
</tr>
<tr>
<td>claims-clause, defined</td>
<td>328</td>
</tr>
<tr>
<td>claims-clause, used</td>
<td>328</td>
</tr>
<tr>
<td>claims-seq, defined</td>
<td>328</td>
</tr>
<tr>
<td>claims-seq, used</td>
<td>328</td>
</tr>
<tr>
<td>class</td>
<td>62, 114, 167, 239, 243</td>
</tr>
<tr>
<td>class constraint</td>
<td>299</td>
</tr>
<tr>
<td>class declarations</td>
<td>200</td>
</tr>
<tr>
<td>class for exception</td>
<td>150</td>
</tr>
<tr>
<td>class invariant</td>
<td>196</td>
</tr>
<tr>
<td>class key</td>
<td>62</td>
</tr>
<tr>
<td>class member, implicit preconditions for</td>
<td>109</td>
</tr>
<tr>
<td>class name, complete</td>
<td>60, 61</td>
</tr>
<tr>
<td>class name, qualified</td>
<td>60, 61</td>
</tr>
<tr>
<td>class specification</td>
<td>167</td>
</tr>
<tr>
<td>class specification refinement</td>
<td>260</td>
</tr>
<tr>
<td>class specification, example</td>
<td>171</td>
</tr>
<tr>
<td>class template</td>
<td>239</td>
</tr>
<tr>
<td>class template, refinement of</td>
<td>267</td>
</tr>
<tr>
<td>class types</td>
<td>299</td>
</tr>
<tr>
<td>class types, abstract values</td>
<td>299</td>
</tr>
<tr>
<td>class types, automatically defined traits for</td>
<td>299</td>
</tr>
<tr>
<td>class types, modifies clause sugar</td>
<td>114</td>
</tr>
<tr>
<td>class variable</td>
<td>69</td>
</tr>
<tr>
<td>class vs. structure (in Larch/C++)</td>
<td>167</td>
</tr>
<tr>
<td>class, abstract</td>
<td>234</td>
</tr>
<tr>
<td>class, abstract values of</td>
<td>168</td>
</tr>
<tr>
<td>class, automatically constructed trait for</td>
<td>299</td>
</tr>
<tr>
<td>class, base</td>
<td>243</td>
</tr>
<tr>
<td>class, concrete</td>
<td>234</td>
</tr>
<tr>
<td>class, derived</td>
<td>15, 214</td>
</tr>
<tr>
<td>class, destructor</td>
<td>190</td>
</tr>
<tr>
<td>class, member functions and abstract values</td>
<td>302</td>
</tr>
<tr>
<td>class, nested refinement</td>
<td>268</td>
</tr>
<tr>
<td>class, protection and abstract values</td>
<td>302</td>
</tr>
<tr>
<td>class, visibility and abstract values</td>
<td>302</td>
</tr>
<tr>
<td>class-head, defined</td>
<td>167</td>
</tr>
<tr>
<td>class-head, used</td>
<td>167</td>
</tr>
<tr>
<td>class-key, defined</td>
<td>62</td>
</tr>
<tr>
<td>class-key, used</td>
<td>62, 167, 257</td>
</tr>
<tr>
<td>class-name, defined</td>
<td>61</td>
</tr>
<tr>
<td>class-name, used</td>
<td>61, 78, 88</td>
</tr>
<tr>
<td>class-specifier</td>
<td>167</td>
</tr>
<tr>
<td>class-specifier, defined</td>
<td>167</td>
</tr>
<tr>
<td>class-specifier, used</td>
<td>59</td>
</tr>
<tr>
<td>client</td>
<td>15</td>
</tr>
<tr>
<td>code inheritance</td>
<td>214</td>
</tr>
<tr>
<td>coercion function</td>
<td>223</td>
</tr>
<tr>
<td>comment</td>
<td>36</td>
</tr>
<tr>
<td>comment, defined</td>
<td>37</td>
</tr>
<tr>
<td>comment, used</td>
<td>36</td>
</tr>
<tr>
<td>comment, vs. annotation</td>
<td>36</td>
</tr>
<tr>
<td>compl</td>
<td>52</td>
</tr>
<tr>
<td>complete-class-name, defined</td>
<td>61</td>
</tr>
<tr>
<td>complete-class-name, used</td>
<td>60, 66, 73, 78, 214, 243</td>
</tr>
<tr>
<td>complete-name-space-name, defined</td>
<td>61</td>
</tr>
<tr>
<td>complete-name-space-name, used</td>
<td>66, 78</td>
</tr>
<tr>
<td>complete-template-class-name, defined</td>
<td>73</td>
</tr>
<tr>
<td>complete-template-class-name, used</td>
<td>73, 239</td>
</tr>
<tr>
<td>complete-template-non-class-name, defined</td>
<td>239</td>
</tr>
<tr>
<td>complete-template-non-class-name, used</td>
<td>239</td>
</tr>
<tr>
<td>complete-type-name, defined</td>
<td>60</td>
</tr>
<tr>
<td>complete-type-name, used</td>
<td>60</td>
</tr>
<tr>
<td>compound-statement, defined</td>
<td>73</td>
</tr>
<tr>
<td>compound-statement, used</td>
<td>73, 82</td>
</tr>
<tr>
<td>concrete class</td>
<td>234</td>
</tr>
<tr>
<td>conditional term</td>
<td>86</td>
</tr>
<tr>
<td>conjunction</td>
<td>87</td>
</tr>
</tbody>
</table>
const declarations of globals, sorts of (table) ........... 75
const formal parameter declarations and their sorts
 (table) ........................................ 96
const string, model of C++ ........................................ 298
constant ........................................ 47, 65
constant declarations ........................................ 75
constant object sorts ........................................ 29
constant pointer ........................................ 75
constant reference ........................................ 75
constant string, C++ null terminated ....................... 298
constant, character ........................................ 49
constant, float ........................................ 48
constant, fractional ........................................ 48
constant, integer ........................................ 47
constant, object ........................................ 29
constant-expression, see a C++ grammar for definition ........................................ 56
constant-expression, used ........................................ 56, 66, 186
constant-initializer, defined ........................................ 186
constant-initializer, used ........................................ 64, 186
constants, abstract string ........................................ 51
constants, floating point ........................................ 48
constants, integer ........................................ 48
constants, string ........................................ 50
ConstObj sort generator ........................................ 75
ConstObj sorts ........................................ 29
constrained-set, defined ........................................ 201
constrained-set, how to use ........................................ 204
constrained-set, used ........................................ 201
constraint ........................................ 201
constraint, for a class ........................................ 200
constraint-clause, defined ........................................ 201
constraint-clause, used ........................................ 186, 254
constraints, inheritance of ........................................ 222
constructor ........................................ 110, 113, 188
constructor, copy ........................................ 190
constructor, default ........................................ 189, 191
constructor, implicit ........................................ 191
constructs ........................................ 110, 113
contained objects ........................................ 113, 207, 284, 301
contained objects of template container class ............... 242
contained objects, trait ........................................ 207

contained_objects ........................................ 120, 207, 242
container class, contained objects of ...................... 242
container membership assumption trait ...................... 243
container objects trait ........................................ 242
containment ........................................ 120
context-dependent-keyword, defined ......................... 42
context-dependent-keyword, used ........................................ 39
conversion-function-id, defined ........................................ 66
conversion-function-id, used ........................................ 66
copy constructor ........................................ 190
copy constructor, implicitly generated ...................... 191
correctly implements ........................................ 33
correctness, partial ........................................ 151
correctness, total ........................................ 151
ctor-declarator, defined ........................................ 73
ctor-declarator, used ........................................ 54, 73, 186
ctor-initializer, defined ........................................ 73
ctor-initializer, used ........................................ 73
curly bracket ........................................ 93
cv-qualifier, defined ........................................ 59
cv-qualifier, used ........................................ 59, 66
cv-qualifier-seq, defined ........................................ 66
cv-qualifier-seq, used ........................................ 66

data member ........................................ 235
data member, exposed ........................................ 235
data member, implicit preconditions for ................... 109

data member, public ........................................ 235
data member, specifying in a class .............................. 235
data members ........................................ 188

data members, inheritance of ........................................ 216
datatype ........................................ 4
deallocation ........................................ 123, 126, 190
deallocation, and modifies-clause ........................................ 111
decimal-constant, defined ........................................ 47
decimal-constant, used ........................................ 47, 51
decl-qualifier, defined ........................................ 73
decl-qualifier, used ........................................ 73
decl-qualifier-seq, defined ........................................ 73
decl-qualifier-seq, used ........................................ 73
decl-specifier, defined ........................................ 57
decl-specifier, used ........................................ 57
decl-specifier-seq, defined .......................... 57
decl-specifier-seq, used ......................... 54, 73, 74, 186, 239
declaration ........................................... 18, 54, 58
declaration of exceptions ......................... 148
declaration specifier ................................. 57
declaration, access ................................. 187
declaration, array ................................... 69
declaration, class ................................... 69
declaration, constant ............................... 75
declaration, defined ................................. 54
declaration, for specification purposes only ..... 254
declaration, function ................................. 72
declaration, reference ............................... 67
declaration, refinement of .......................... 257
declaration, samples ................................ 55
declaration, structure (struct) ..................... 69
declaration, union .................................. 72
declaration, used .................................... 78, 80, 239, 254, 257
declaration-seq, defined ........................... 78
declaration-seq, used ............................... 78, 80, 82
declarations and sorts for global variables, summary (table) ..................................... 76
declarations, of globals with const (table) ........ 75
declarations, of globals, and their sorts (table) ... 77
declarator, defined ................................... 66
declarator, used ..................................... 54, 66, 73, 186
declarator-qualifier, defined ...................... 66
declarator-qualifier, used ........................ 66
default constructor .................................. 189, 191
definition ............................................. 18, 58
definition, function .................................. 72
delete .................................................. 44
delete [] .............................................. 44
delete operator vs. destructor ...................... 191
deletion of objects ................................... 190
dependency, example ................................ 218
dependent object ..................................... 110
depends ................................................. 213
depends-clause, defined ............................ 213
depends-clause, example use of .................... 218, 262
depends-clause, used ................................ 186, 254
deprecated .......................................... 328
dereferencing a pointer ............................. 290
derived class ......................................... 15, 214
derived class, how to specify ...................... 218
described, by a modifies-clause .................... 110
described, for a trashes clause ...................... 127
description, informal ................................ 91
design decision, recording ......................... 16, 58, 59, 188, 215, 237
destructor ............................................. 190
destructor, implicitly generated ................... 191
destructor, specification of ........................ 251
detailed design decision, recording ............... 16, 58, 59, 188, 215, 237
detailed design, recording ......................... 214
determinism, in a function specification ........... 83
digit, defined ......................................... 39
digit, used ............................................ 39, 48
direct-abstract-declarator, defined ............... 66
direct-abstract-declarator, used .................... 66
direct-declarator, defined ........................... 66
direct-declarator, used ............................... 66
distinguishing between returning and exceptions .. 148
domain, of a function or relation .................. 84
DOS, port of Larch/C++ for .......................... 10
double ............................................... 48, 60, 275
dtor-name, defined ................................... 73
dtor-name, used ...................................... 73
dynamic binding ...................................... 215

e .................................................. 48
E ...................................................... 48
Eiffel .................................................. 233
elaborated-type-specifier, defined ............... 62
elaborated-type-specifier, used .................... 59
Ellis, M.A. ............................................ 1
else .................................................... 86
ensures ............................................... 82
ensures-clause, defined ............................. 82
ensures-clause, used ................................ 82, 164
ensures-clause-seq, defined ......................... 82
ensures-clause-seq, used ............................ 82
enum .................................................. 62, 64, 278
enum-name, defined .................................. 60
denum-name, used .................................... 60

exponent-indicator, defined .......................... 48
exponent-part, defined .............................. 48

explicit-given traits, and weak behavioral subtyping
explicitly-given trait ................................. 181
explicitly-given traits, and weak behavioral subtyping

extern ............................................... 54, 58, 80, 134
external linkage ............................... 54

F
f .................................................. 48
F .................................................. 48
false ............................................. 276

cnId, defined ....................................... 94
cnId, used ........................................ 94, 223

field ............................................. 290
field, of a class value ............................ 69
field, of a structure value ....................... 69
field, of a union .................................. 72

file ............................................... 17
file suffix for Larch/C++ .......................... 17

float ............................................. 48, 60, 275

float-suffix, defined ............................... 48
float-suffix, used ................................ 48
floating point constants .......................... 48

Floating-constant, defined ....................... 48
Floating-constant, used ......................... 47
FloatingPoint trait ................................ 275

for ............................................... 201, 255
for all ............................................. 88
for some .......................................... 88

formal documentation .............................. 6
formal parameter ................................. 23
formal parameter declarations and their sorts (table)
formal parameter, complex, sorts of (table) .... 95
formal parameters, const, sorts of (table) ..... 97
formal parameters, sorts of (table) ............. 96
formal specification, reasons for using .................. 6
formality, tuning ........................................ 91
formfeed ..................................................... 36
fractional-constant, defined .............................. 48
fractional-constant, used .................................. 48
frame axiom ............................................... 101, 102, 110
frame axioms ............................................... 204
frame, defined ........................................... 82
frame, used ............................................... 82
Fresco ....................................................... 233
fresh .......................................................... 99, 123
friend ......................................................... 15
friend ......................................................... 57, 63, 237
friendship in classes ...................................... 237
ftp .............................................................. 10
fun-body, defined .......................................... 73
fun-body, used ............................................. 73
fun-interface, defined ..................................... 73
fun-interface, used ......................................... 73, 157, 201
fun-interface-list, defined ................................. 201
fun-interface-list, used .................................. 133, 201
fun-spec-body, defined ................................... 82
fun-spec-body, used ....................................... 54, 73, 157, 186
fun-spec-boyd, used ...................................... 164
fun-try-block, used ....................................... 73
function ........................................................ 65, 82
function declaration ....................................... 72
function definition ......................................... 72
function interface .......................................... 82
function specification scope .............................. 19
function template ......................................... 239
function template, refinement of ......................... 267
function type, abstract model of ......................... 305
function, higher-order .................................... 157
function, interface with const ............................ 96
function, member ........................................... 15
function, parameter specification ......................... 157
function, pointer .......................................... 157
function, specification of .................................. 82
function-definition, defined .............................. 73
function-definition, used .................................. 54, 186
function-name, defined .................................... 132
function-name, used ...................................... 132
function-names, defined .................................. 132
function-names, used ..................................... 132
function-specifier, defined ............................... 59
function-specifier, used .................................. 57
function-try-block, defined .............................. 73

G

g++, attribute definitions .................................. 38
gcc, attribute definitions .................................. 38
getting, the LP checker .................................... 11
getting, the LSL checker ................................... 11
ghost variable ................................................ 254
ghost variable, in class specification ..................... 168
global array variable ...................................... 69
global class variable ....................................... 69
global pointer variable .................................... 68
global reference variable .................................. 67
global structure (struct) variable ......................... 69
global union variable ...................................... 72
global variable ............................................. 54, 134
global variables, sorts of (table) .......................... 77
global variables, sorts of, summary (table) .............. 76
global variables, with const, sorts (table) .............. 75
GNU ........................................................... 10
goals .......................................................... 1, 7
grammar, conventions for lists ............................ 35

H

handbook, for LSL .......................................... 4
Handbook, for LSL .......................................... 4
handler, defined ............................................ 73
handler, used ............................................... 73
handler-seq, defined ....................................... 73
handler-seq, used .......................................... 73
hex-constant, defined ..................................... 47
hex-constant, used ........................................ 47
hex-digit, defined .......................................... 47
hex-digit, used ............................................ 47, 49
hex-indicator, defined ..................................... 47
hex-indicator, used ........................................ 47
hiding ........................................................... 78
higher-order functions ..................................... 157
higher-order-comparison, defined ......................... 157
include files ........................................ 265
history constraint, example .......................... 205
history constraint, inheritance for strong behavioral
  subtypes ........................................... 231
history constraints, inheritance of .................. 222
Hoare .................................................. 4, 82
home page, for Larch .................................. 12
homomorphism property ................................ 226

I
id-expression, defined ................................. 66
id-expression, used .................................... 66
ident() .............................................. 39
identifier, defined .................................... 39
identifier, possible meanings ....................... 40
identifier, sort ..................................... 94
identifier, used ..................................... 39, 42, 46, 62, 64, 66, 78, 88, 94,
  98, 167, 186, 239, 243, 257
if then else ......................................... 86
illustrating, function specifications ............... 137
immutable .......................................... 175
implementation ...................................... 33
implementations and invariants .................... 196
implements .......................................... 33
implicit preconditions ............................... 109
implicitly generated assignment operator ......... 191
implicitly generated constructor ................... 191
implicitly generated copy constructor ............. 191
implicitly generated destructor ..................... 191
implicitly generated member functions .......... 191
implicitly generated members, suppressing .... 196
implies ............................................. 87
include files ....................................... 17
including, a trait .................................. 255
incomplete specification, defined ................. 139
incomplete specification, examples for .......... 139
incomplete, function specification ................. 83
incompleteness, in a function specification ..... 83
infix operator ...................................... 45, 92
influences, on Larch/C++ evolution ............... 9
informal specification, example .................. 260
informal, modifies clause .......................... 112
informal-comment, defined ......................... 46
informal-comment, used ............................ 91
informal-comment-body, defined .................. 46
informal-comment-body, used ...................... 46
informal-desc, defined ............................. 91
informal-desc, no longer used for traits ......... 328
informal-desc, used ................................ 87
informally .......................................... 91, 112, 260
inheritance .......................................... 7
inheritance of data members ....................... 216
inheritance of history constraints ................ 222
inheritance of history constraints, for weak behavioral
  subtypes .......................................... 227
inheritance of invariants ........................... 222
inheritance of specifications ....................... 216
inheritance of specifications, desugaring ....... 227
inheritance of specifications, related work ...... 233
inheritance of specifications, strengthening ..... 221
inheritance, not for a uses-clause ................. 263
inheritance, of specifications ..................... 215
inheritance, of specifications, details .......... 231
inheritance, specifying code ....................... 214
init-declarator, defined ........................... 54
init-declarator, used ................................ 54, 74, 239
init-declarator-list, defined ....................... 54
init-declarator-list, used .......................... 54
initial value ........................................ 56
initializer, defined ................................ 56
initializer, used .................................... 54, 74
initializer-clause, defined ......................... 56
initializer-clause, used ............................ 56
initializer-list, defined ............................ 56
initializer-list, used ............................... 56
inline .............................................. 59
instance ............................................ 21
instance value ...................................... 21
instance variables .................................. 188
int .................................................. 47, 60, 269, 272
integer constants .................................. 48
Integer trait ....................................... 269
integer traits, summary ............................ 274
integer types ....................................... 269
integer-constant, defined ......................... 47
integer-constant, used ............................ 47
integer-suffix, defined ............................ 47

Concept Index 341
interface, protected ........................................ 47
integers, unsigned ......................................... 272
interface ...................................................... 7, 15, 54
interface for exceptions ................................... 148
interface of a function with const ......................... 96
interface specification ...................................... 2
interface specification module .............................. 254
interface specifications of built-in types .................. 269
interface, access to members of ............................ 16
interface, defined ........................................... 254
interface, private ........................................... 16
interface, protected ........................................ 16
interface, public ............................................ 15
internal linkage ............................................. 58
international character set support ......................... 52
invariant ....................................................... 196
invariant, for a class ......................................... 196
invariant-clause, defined ................................... 196
invariant-clause, used ....................................... 186, 254
invariants, inheritance of ................................... 222
is ............................................................... 243

K
keyword ......................................................... 40
keyword, defined ............................................. 40
keyword, used ................................................ 39
keyword, using one as an identifier ......................... 39
keywords ....................................................... 41
Keywords in Predicates ...................................... 46
keywords, added to C++ ...................................... 41
keywords, recognized everywhere .......................... 41

L
1 .................................................................. 47, 48
L ................................................................. 47, 48, 49, 50
Larch ............................................................ 2
Larch Prover, obtaining ...................................... 11
Larch Shared Language (LSL) ............................... 2
Larch Shared Language Checker ................................ 9
Larch style specification language ........................... 2
Larch, global home page for ................................ 12
larch-cpp-clause, defined .................................... 186
larch-cpp-clause, used ......................................... 186
Larch-style specification ..................................... 235
Larch/C .......................................................... 7
Larch/C++ reserved words not in C++ ....................... 41
Larch/C++ reserved words recognized everywhere ........ 41
Larch/C++ status and plans ................................... 8
Larch/C++, evolution .......................................... 9
Larch/C++, ftp ................................................ 10
Larch/C++, goals .............................................. 7
Larch/C++, obtaining .......................................... 10
Larch/C++, plans .............................................. 8
Larch/C++, status ............................................. 8
LCL ............................................................... 7
lcpp, specification checker .................................... 12
lcpp-primary, defined ....................................... 99
lcpp-primary, used .......................................... 94
Leino .......................................................... 110, 233
let ............................................................... 136
let scope ......................................................... 19
let-clause, defined ........................................... 136
let-clause, used ............................................... 82
letter, defined ................................................ 39
letter, used .................................................... 39, 40
letter-or-digit, defined ...................................... 39
letter-or-digit, used ......................................... 39, 40
lexeme, defined .............................................. 36
lexeme, used .................................................. 36
lexical conventions ............................................ 36
liberal specification ......................................... 151
liberally ......................................................... 82, 138, 151
linkage ........................................................ 58
linkage, external ............................................. 58
linkage, internal ............................................. 58
linkage-declaration, defined ............................... 80
linkage-declaration, used ................................... 54
list vs. sequence, in grammar .................................. 35
literal .......................................................... 47
literal, defined .............................................. 47
literal, used .................................................. 39, 99
LM3 ........................................................... 101, 233
local scope ..................................................... 19
location ........................................................ 21
logical-opr, defined ......................................... 87
logical-opr, used ............................................. 87

Concept Index 342
logical-term, defined .......................... 87
logical-term, used ................................ 86, 87
long ................................. 47, 60, 269, 271
long double .................................. 48
long double ................. 275
long integer trait .............. 271
long integers ......................... 269
long-suffix, defined .............. 47
long-suffix, used ................. 47
looping, specification of .......... 151
LP, obtaining ............................... 11
LP, the Larch Prover .................... 9
LSL .................................. 2, 45
LSL checker ................................. 2
LSL checker, obtaining .............. 11
LSL checker, use ......................... 11
LSL Handbook .............................. 4
LSL initialization for Larch/C++ ..... 277
LSL operators, reserved ............. 46
LSL trait ................................. 2
LSL tuple .................................. 93, 98
lsl, LSL trait checker ................. 12
lsl-constant, defined ............... 51
lsl-formal, defined ................. 255
lsl-formal, used ......................... 255
lsl-instance-actuals, defined ....... 255
lsl-instance-actuals, used .......... 255
lsl-op, defined ......................... 45
lsl-op, reserved ......................... 46
lsl-op, used ............................ 43, 92
lsl-op-term, defined ................. 92
lsl-op-term, used ....................... 87, 157
lsl-signature, defined .............. 255
lsl-signature, used .................. 255
lsl-sort, defined ...................... 255
lsl-sort, used .......................... 255
lsl-sort-list, defined ............... 255
lsl-sort-list, used .................... 255
lvalue .................................. 21

mem-initializer, used .................... 73
mem-initializer-id, defined ........... 73
mem-initializer-id, used ......... 73
member function ....................... 15
member functions, implicit .......... 191
member, implicit preconditions for .. 109
member, of a structure ............... 299
member, pointer to ................. 68, 294
member-declaration, defined ...... 186
member-declaration, used .......... 186, 268
member-declarator, defined ....... 186
member-declarator, used .......... 186
member-declarator-list, defined ... 186
member-declarator-list, used ...... 186
member-seq, defined ................. 186
member-seq, used ............... 167, 186, 243
message send ............................ 215
microsyntax, defined ............. 36
model, of a struct, user-defined ... 71
model, of objects .................... 24
model, of states ...................... 30
model-oriented specification ....... 2
modifies ..................... 101, 110, 115, 188, 190
modifies clause, and arrays ......... 112
modifies clause, and pointers ...... 112
modifies clause, informal ......... 112
modifies, in constructor .......... 188
modifies, omitted .................... 112
modifies-clause ....................... 101
modifies-clause, defined .......... 110
modifies-clause, in constructor ..... 188
modifies-clause, in destructor ....... 190
modifies-clause, used ............ 110, 164
modifies-clause-seq, defined ...... 110
modifies-clause-seq, used .......... 82
modular reasoning ......................... 215
modular specification ............... 226
modular verification ................. 215
module .................................. 17, 254
modules, namespace .................. 78
MS-DOS, port of Larch/C++ for ...... 10
multi-dimensional array trait ..... 285
multicharacter constant .......... 49
mutable ........................................... 58
mutable object ................................. 28
mutates ......................................... 101
mutation ...................................... 101, 115

N
name spaces, for identifiers .................. 40
name, identity between class and sort name . 181
namespace ...................................... 78
namespace name, complete .................. 61
namespace name, qualified ................. 61
namespace, refinement of .................. 267
namespace-alias-definition, defined .... 78
namespace-alias-definition, used .... 54, 78
namespace-alias-name, defined ........ 42
namespace-alias-name, used ............ 42, 61, 257
namespace-definition, defined ......... 78
namespace-definition, used ............. 54
namespace-name, defined ............... 61
namespace-name, used ................. 61
narrowing ..................................... 26
natural numbers ................................ 272
nested class, refinement of ............ 268
nested-name-specifier, defined ........ 61
nested-name-specifier, used .......... 60, 61, 73
new .............................................. 44
new [] ........................................... 44
newline ........................................ 36, 37
newline, defined .............................. 36
newline, used ................................. 36, 37
nominal type .................................. 8
non-at-newline, defined ................. 37
non-at-newline, used ...................... 37
non-at-star, defined ...................... 37
non-at-star, used ............................ 37
non-class-type-name, used ............ 78
non-escape-code, defined .............. 50
non-escape-code, used ..................... 50
non-newline, defined ..................... 37
non-newline, used ......................... 37, 38
non-nl-white-space, defined ........ 36
non-nl-white-space, used .............. 36, 38
non-nl-whitespace, used ................ 44
non-percent, defined ....................... 46
non-percent, used ........................... 46
non-percent-right-paren, defined .... 46
non-percent-right-paren, used .... 46
non-right-paren, defined ............... 46
non-right-paren, used ..................... 46
non-semi-newline, defined ............ 38
non-semi-newline, used .................. 38
non-slash, defined ......................... 37
non-slash, used ............................... 37
non-star, defined ............................ 37
non-star, used ................................. 37
non-star-slash, defined ................. 37
non-star-slash, used ...................... 37
non-std-esc, defined ...................... 50
non-std-esc, used ............................ 50
non-type-parameter-decl, defined .... 239
non-type-parameter-decl, used .... 239
nondeterminism, in a function specification . 83
nontermination ................................. 151
normal-char, defined ...................... 49
normal-char, used ........................... 49, 50
not ............................................... 52
not_eq ........................................... 52
nothing ........................................ 110, 132, 201
null terminated const string .......... 298
null terminated string ................... 296
nullTerminated trait function .......... 297

O
Obj sorts ........................................ 28
object ........................................... 21, 24, 69, 72, 103, 107
object deallocation ......................... 126
object identifier ............................. 21
object trashing ................................ 126
object, constant .............................. 29
object, contained ............................ 120
object, destruction ......................... 190
object, mutable ............................... 28
object, reachable ............................ 120
object, untyped model of ................ 30
objects, abstract values of ............. 168
objects, constant ............................ 29
objects, contained .......................................... 207
objects, sorts of ........................................... 24
obtaining, the LP checker .................................. 11
obtaining, the LSL checker .................................. 11
octal-code, defined .......................................... 49
octal-code, used .............................................. 49
octal-constant, defined ...................................... 47
octal-constant, used ......................................... 47
octal-digit, defined .......................................... 47
octal-digit, used .............................................. 47, 49
one-to-nine, defined ......................................... 47
one-to-nine, used .............................................. 47
op-char, defined ............................................. 45
op-char, used .................................................. 45
operation ....................................................... 4
operator ......................................................... 65
operator character ........................................... 45
operator symbols, C++ ....................................... 44
operator, infix .................................................. 45
operator, of LSL ................................................ 4
operator, postfix .............................................. 45
operator, prefix ............................................... 45
operator-function-id, defined .............................. 66
operator-function-id, used ................................... 66
or ............................................................. 52
or ............................................................. 87
or_eq ........................................................... 52
original-class-name, defined ................................. 42
original-class-name, used .................................... 42, 61, 243, 257
original-enum-name, defined ................................. 42
original-enum-name, used .................................... 42, 60
original-namespace-name, defined ......................... 42
original-namespace-name, used ............................. 42, 61, 78, 257
OS/2, port of Larch/C++ for .................................. 10
overloading, dynamic ......................................... 215
overloading, of trait functions ................................ 226
overview ........................................................ 1

P

param-qualifier, defined ..................................... 66
param-qualifier, used ........................................ 66, 73
parameter, formal ............................................. 23
parameter, sort ............................................... 94
parameter, to a template ..................................... 239
parameter, value .............................................. 300, 303
parameter-declaration, defined ............................. 74
parameter-declaration, used ................................ 74
parameter-declaration-clause, defined ..................... 74
parameter-declaration-clause, used ......................... 66
parameter-declaration-list, defined ......................... 74
parameter-declaration-list, used ........................... 74
parameter-initializer, defined ............................... 74
parameter-initializer, used ................................... 74
parameters, formal, sorts of (table) ......................... 95
parser, for Larch/C++ ......................................... 9
partial correctness ........................................... 151, 152
percents-non-right-paren, defined ......................... 46
percents-non-right-paren, used ............................. 46
perspectives, recording ....................................... 257
plans, for Larch/C++ .......................................... 8
pointer ......................................................... 21, 65, 68, 125, 287
pointer and array, relationship .............................. 95
pointer dereference .......................................... 290
pointer to a constant ........................................ 75
pointer to member ............................................ 68, 294
pointer variable .............................................. 68
pointer, and array ........................................... 279
pointer, and array formal .................................... 95
pointer, constant ............................................. 75
pointer, deallocation via ..................................... 127
pointer, differences from LCL ............................... 326
pointer, to array ............................................. 106
pointer, to function .......................................... 157
pointer, valid ................................................. 290
pointers, in modifies clause ................................ 112
polymorphic ................................................... 215
polymorphism, subtype ...................................... 215
post .......................................................... 99, 101
post .......................................................... 103
post-cond, defined .......................................... 82
post-cond, used .............................................. 82
post-state ..................................................... 101, 102
post-state, for a history constraint ......................... 201
post-value ..................................................... 103
postcondition ............................ 2, 4, 82
postcondition, redundant ............... 141
postfix operator .......................... 45, 92
postfix-expression, defined and see a C++ grammar .......................... 73
postfix-expression, used .................. 73
pragma ..................................... 38
pragma, defined .......................... 38
pragma, used ............................... 36
pre .......................................... 99, 101
pre ........................................... 103
pre-state .................................. 201
pre-state, for a history constraint ......... 201
pre-value .................................. 103
precondition ................................ 2, 4, 82
precondition, redundant .................. 140
preconditions, implicit ................... 109
predicate, defined ........................ 86
predicate, used ......................... 82, 138, 164, 196, 201
predicate-keyword, used ................. 39
predicate-symbol, defined ............... 45
predicate-symbol, used ................... 43
prefix operator ............................ 45, 92
preprocessing ............................. 17
primary, defined .......................... 93
primary, used ............................... 93
primary-suffix, defined ................... 98
primary-suffix, used ..................... 93
primitive, defined ........................ 94
primitive, used ........................... 93
private ..................................... 15
private .................................... 16, 186, 196
private interface .......................... 16
private members, granting friendship to ........................................ 237
procedure, abstract model of ............ 305
procedure, higher-order .................. 157
program .................................... 82
program, abstract ......................... 164
program, behavior ....................... 164
protected .................................. 16, 186, 196
protected interface ....................... 16
ptr-operator, defined .................... 66
ptr-operator, used ........................ 66
public ....................................... 16, 186
public data member ....................... 235
public interface .......................... 15
punctuation symbols ....................... 44
punctuation symbols, used in C++ declarations ........... 44
pure specifier ................................ 234
pure virtual function ...................... 234
pure virtual, member function ............ 59
pure virtual, removed ...................... 328
Q
qualified-id, defined ..................... 66
qualified-id, used ........................ 66, 78, 94
quantified varId, sort .................... 94
quantified-list, defined .................. 88
quantified-list, used ..................... 88
quantifier scope .......................... 19, 89
quantifier, defined ...................... 88
quantifier, used .......................... 87
quantifier-sym, defined ................... 88
quantifier-sym, used ..................... 88
quantifiers, eliminating ................... 90
R
reach ....................................... 110, 112, 120
reachability ............................... 120
reasoning, modular ....................... 215
reasons, for formal documentation .......... 6
record ...................................... 299
recording detailed design .................. 214
recording detailed design decision ....... 58, 59, 188, 215, 237
recording detailed design decision, recording .......... 16
redundancy, in function specifications .... 137
redundancy, in history constraints ........ 202
redundancy, in postcondition ............. 141
redundancy, in precondition .............. 140
redundantly .............................. 82, 110, 127, 132, 133, 140, 141, 196, 201
reference ................................... 65, 67
reference parameters, implicit requirement .... 108
concept index 347

reserved words for C++ statements and expressions

reserved words, for C++ statements and expressions

reserved words, recognized everywhere

reserved words, used in a term or predicate

result

result, sort of (table)

returning, instead of signalling

returns

rvalue

S

satisfaction

satisfies

satisfies, for function specifications

sc-bracketed, defined

sc-bracketed, used

scope

scope, function specification

scope, let

scope, local

scope, of a let-clause

scope, quantifier

secondary, defined

secondary, used

selection, defined

selection, used

self

self, data members

self, in class specifications

send, of message

sequence vs. specifications

short

short integer trait

short integers

sign, defined

sign, used

signalling an exception

signature

signature, example
signed.............................. 47, 60, 269
signed integer trait .................. 269
simple-id, defined .................. 40
simple-id, used .................. 39, 157, 255
simple-type-name, defined .......... 60
simple-type-name, used ............. 59
simulates.......................... 223
simulates-clause, defined .......... 223
simulates-clause, example of ...... 225
simulates-clause, used ............. 186
simulation function ................ 223
simulation function, homomorphism property ... 226
simulation function, use in inheritance ... 227
simulation functions, and state ........ 232
sizeof................................ 99
sort.................................. 5, 20, 75, 76, 94
sort checking .................. 21
sort of result (table) ............. 100
sort, for a class .................. 181
sort, of C++ strings ................. 296
sort-instance-actuals, defined ...... 88
sort-instance-actuals, used .......... 88
sort-name, defined ................. 88
sort-name, used .................. 88, 93, 98, 136, 223
sort-or-type, defined ............ 88
sort-or-type, used ............... 88
sorts and types, association of, example ... 169, 218
sorts for complex declarations of formal parameters .......... 97
sorts for complex declarations of global variables (table) .... 77
sorts of global variables (table) .... 76
sorts of objects .................. 24
sorts, of constant objects ........ 29
sorts, of global variables using const (table) ..... 75
space, name .................. 78
spec................................. 168, 254
spec-case.......................... 143
spec-case, defined ............... 82, 328
spec-case, removed form .......... 328
spec-case, used ............. 82
spec-case-seq .................. 143
spec-case-seq, defined ............ 82
spec-decl, defined ................ 254
spec-decl, example in class specifications .... 168
spec-decl, used .................. 78, 186, 254
SPEC_PATH environment variable .......... 11
special-function-declarator, defined ... 73
special-function-declarator, used ...... 73, 186
special-symbol, defined ............ 43, 44
special-symbol, used .............. 39
specification declaration .......... 254
specification declaration, for a class, example ...... 169,
218
specification inheritance .......... 215, 216
specification inheritance, details ...... 231
specification module .............. 254
specification of exceptions .......... 148
specification styles .............. 168
specification variable .............. 254
specification variable, in class specification .......... 168
specification variables, and inheritance ...... 216
specification, examples in .......... 137
specification, liberal .............. 151
specification, modular .............. 226
specification, of a class .......... 167
specification, of a function ...... 82
specification, of an interface’s behavior .... 15
specification, of interface behavior .... ... 2
specification, partial .............. 151
specification, refinement of .......... 257
specification-statement, defined .... 164
specification-statement, used .......... 73
square bracket .................. 93
star-or-op-char, defined .......... 45
star-or-op-char, used ............. 45
stars-non-slash, defined .......... 37
stars-non-slash, used ............. 37
state................................. 22, 30, 101, 103, 107
state, and simulation functions ...... 232
state, model of .................. 30
State, sort .................. 30
state, trait for .................. 30
state, values from objects in ....... 103
state-function ................. 101
<table>
<thead>
<tr>
<th>Concept Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>state-function, defined</td>
<td>103</td>
</tr>
<tr>
<td>state-function, used</td>
<td>98</td>
</tr>
<tr>
<td>statement, additional keywords for</td>
<td>41</td>
</tr>
<tr>
<td>statement, defined</td>
<td>73</td>
</tr>
<tr>
<td>statement, used</td>
<td>73</td>
</tr>
<tr>
<td>states, in trait functions</td>
<td>297</td>
</tr>
<tr>
<td>states, names for</td>
<td>101</td>
</tr>
<tr>
<td>static</td>
<td>17, 58</td>
</tr>
<tr>
<td>static type</td>
<td>8</td>
</tr>
<tr>
<td>status, of Larch/C++</td>
<td>8</td>
</tr>
<tr>
<td>std-esc, defined</td>
<td>49</td>
</tr>
<tr>
<td>std-esc, used</td>
<td>49, 50</td>
</tr>
<tr>
<td>stmt-exp-only-keyword, defined</td>
<td>41</td>
</tr>
<tr>
<td>stmt-exp-only-keyword, used</td>
<td>40</td>
</tr>
<tr>
<td>storage-class-specifier, defined</td>
<td>58</td>
</tr>
<tr>
<td>storage-class-specifier, used</td>
<td>57</td>
</tr>
<tr>
<td>store-ref, defined</td>
<td>110</td>
</tr>
<tr>
<td>store-ref, used</td>
<td>110, 127, 213</td>
</tr>
<tr>
<td>store-ref-list, defined</td>
<td>110</td>
</tr>
<tr>
<td>store-ref-list, used</td>
<td>99, 110, 127, 133, 213</td>
</tr>
<tr>
<td>strengthening specifications in derived classes</td>
<td>221</td>
</tr>
<tr>
<td>string</td>
<td>50</td>
</tr>
<tr>
<td>string (of characters)</td>
<td>50</td>
</tr>
<tr>
<td>string (of characters), abstract</td>
<td>51</td>
</tr>
<tr>
<td>string constants</td>
<td>50</td>
</tr>
<tr>
<td>string, abstract</td>
<td>51</td>
</tr>
<tr>
<td>string, C++ null terminated</td>
<td>296</td>
</tr>
<tr>
<td>string, in C++</td>
<td>296</td>
</tr>
<tr>
<td>string, model of C++</td>
<td>296</td>
</tr>
<tr>
<td>string, unsigned character</td>
<td>298</td>
</tr>
<tr>
<td>string, wide</td>
<td>50</td>
</tr>
<tr>
<td>string, wide character</td>
<td>298</td>
</tr>
<tr>
<td>string-character, defined</td>
<td>50</td>
</tr>
<tr>
<td>string-character, used</td>
<td>50, 51</td>
</tr>
<tr>
<td>string-literal, defined</td>
<td>50</td>
</tr>
<tr>
<td>string-literal, used</td>
<td>47, 80, 81, 91</td>
</tr>
<tr>
<td>strong behavioral subtype</td>
<td>231</td>
</tr>
<tr>
<td>strong behavioral subtype, and inheritance of history</td>
<td>231</td>
</tr>
<tr>
<td>constraints</td>
<td>231</td>
</tr>
<tr>
<td>Stroustrup, B.</td>
<td>1</td>
</tr>
<tr>
<td>struct</td>
<td>62, 69, 106, 114, 126, 167</td>
</tr>
<tr>
<td>struct, differences from LCL</td>
<td>299</td>
</tr>
<tr>
<td>struct, specifying your own model of</td>
<td>71</td>
</tr>
<tr>
<td>structure declarations</td>
<td>69</td>
</tr>
<tr>
<td>structure types</td>
<td>299</td>
</tr>
<tr>
<td>structure types, abstract values</td>
<td>299</td>
</tr>
<tr>
<td>structure types, automatically defined traits for</td>
<td>299</td>
</tr>
<tr>
<td>structure types, differences from LCL</td>
<td>326</td>
</tr>
<tr>
<td>structure types, modifies clause sugar</td>
<td>114</td>
</tr>
<tr>
<td>structure variable</td>
<td>69</td>
</tr>
<tr>
<td>structure, abstract value of</td>
<td>167</td>
</tr>
<tr>
<td>structure, automatically constructed trait for</td>
<td>299</td>
</tr>
<tr>
<td>structure, field</td>
<td>299</td>
</tr>
<tr>
<td>structure, member functions and abstract values</td>
<td>302</td>
</tr>
<tr>
<td>structure, protection and abstract values</td>
<td>302</td>
</tr>
<tr>
<td>structure, specification of</td>
<td>167</td>
</tr>
<tr>
<td>structure, state functions applied to</td>
<td>106</td>
</tr>
<tr>
<td>structure, visibility and abstract values</td>
<td>302</td>
</tr>
<tr>
<td>structure, vs class (in Larch/C++)</td>
<td>167</td>
</tr>
<tr>
<td>style guideline</td>
<td>17, 19, 60, 64, 91, 101, 104, 112, 138, 140, 171, 189, 190, 192, 200, 214, 238</td>
</tr>
<tr>
<td>styles, of specification</td>
<td>168</td>
</tr>
<tr>
<td>subclass</td>
<td>15, 214</td>
</tr>
<tr>
<td>subclass, how to specify</td>
<td>218</td>
</tr>
<tr>
<td>subtype</td>
<td>88, 215</td>
</tr>
<tr>
<td>subtype, behavioral</td>
<td>8, 215</td>
</tr>
<tr>
<td>subtype, how to specify</td>
<td>218</td>
</tr>
<tr>
<td>subtype, strong behavioral</td>
<td>231</td>
</tr>
<tr>
<td>subtyping</td>
<td>215</td>
</tr>
<tr>
<td>suffix, for Larch/C++</td>
<td>17</td>
</tr>
<tr>
<td>sugar, syntactic for fresh</td>
<td>125</td>
</tr>
<tr>
<td>sugar, syntactic in modifies-clause</td>
<td>113</td>
</tr>
<tr>
<td>sugars, preconditions in functions</td>
<td>109</td>
</tr>
<tr>
<td>summary of integer traits</td>
<td>274</td>
</tr>
<tr>
<td>summary of sorts for formal parameters (table)</td>
<td>96</td>
</tr>
<tr>
<td>supertype</td>
<td>8, 215</td>
</tr>
<tr>
<td>supertype abstraction</td>
<td>8, 215</td>
</tr>
<tr>
<td>supertype abstraction, in specification</td>
<td>226</td>
</tr>
<tr>
<td>supertype, defined</td>
<td>223</td>
</tr>
<tr>
<td>supertype, used</td>
<td>223</td>
</tr>
<tr>
<td>supertype-list, defined</td>
<td>233</td>
</tr>
<tr>
<td>supertype-list, used</td>
<td>233</td>
</tr>
<tr>
<td>suppressing implicitly generated members</td>
<td>196</td>
</tr>
<tr>
<td>symbols, for C++ operators</td>
<td>44</td>
</tr>
<tr>
<td>symbols, punctuation in C++ declarations</td>
<td>44</td>
</tr>
</tbody>
</table>
trait, for pointers to objects.......................... 290
trait, for struct ....................................... 299
trait, for typed objects.................................. 24
trait, how it determines abstract values.............. 181
trait, used ............................................. 157, 255
trait-list, defined ....................................... 255
trait-list, used .......................................... 251, 255
trait-name, defined ...................................... 255
trait-name, used ........................................ 255
trait-or-deref-list, defined ....................... 157
trait-or-deref-list, used .............................. 157
traits for built-in types ................................ 269
traits for floating point types .......................... 275
traits for integer types .................................. 269
traits, finding .......................................... 11
traits, for classes ..................................... 168
traits, for integer types .................................. 269
traits, using them ..................................... 255
trashed ................................................... 99, 128, 190
trashed, concept of .................................... 126
trashes ..................................................... 127
trashes-clause, defined .................................. 127
trashes-clause, used .................................... 127, 164
trashes-clause-seq, defined .......................... 127
trashes-clause-seq, used ............................... 82
trashing, and modifies-clause ........................... 111
trashing, how to specify ................................. 126
trick, for specifying derived classes .................... 218
true ....................................................... 276
try ....................................................... 73
tuning of formality ...................................... 91
tuple, LSL ............................................... 93, 98
type ....................................................... 20
type definition .......................................... 63
type equivalence ........................................ 21
type name, simple ....................................... 60
type specifier, elaborated .............................. 62
type to sort mapping ................................... 20, 63, 76, 94, 254
type, abstract .......................................... 4
type, built in .......................................... 60
type, built-in ........................................... 269
type, nominal .......................................... 8
type, static ............................................. 8
type-arg-name, defined ................................ 243
type-arg-name, used ................................... 243
type-id, defined ....................................... 251
type-id, used .......................................... 88, 99, 148, 239, 251
type-init, defined ..................................... 239
type-init, used ......................................... 239
type-list, defined ...................................... 148
type-list, used ......................................... 148
type-name, defined .................................... 78
type-name, used ........................................ 78
type-parameter, defined ................................ 239
type-parameter, used ................................... 239
type-specifier, defined .................................. 59
type-specifier, used .................................... 59, 251
type-specifier-seq, defined .......................... 59
type-specifier-seq, used ............................... 57, 66, 73, 251
typedef ................................................. 57, 63
typedef-class-name, defined ....................... 42
typedef-class-name, used .............................. 42, 257
typedef-enum-name, defined ....................... 42
typedef-enum-name, used .............................. 42, 60, 88
typedef-non-class-or-enum-name, defined ............ 42
typedef-non-class-or-enum-name, used ............... 42, 60, 88
typename ............................................. 62, 78, 239

U

u .......................................................... 47
u .......................................................... 47
unassigned ............................................. 107
unassigned, how to specify making an object .......... 126
unchanged ............................................. 99, 121
union .................................................... 62, 72, 126, 167, 302
union declarations ..................................... 72
union variable .......................................... 72
union, abstract value of ................................ 167
union, differences from LCL ............................ 327
union, example trait for ................................ 303
union, specification of ................................ 167
Unix, port of Larch/C++ for ............................ 10
unqualified-id, defined ................................ 66
unqualified-id, used ................................... 66, 78, 186, 257
unsigned ............................................. 47, 60, 269, 272
unsigned char .......................................... 272
unsigned int ........................................ 272
unsigned long ........................................ 272
unsigned short ......................................... 272
unsigned-suffix, defined .......................... 47
unsigned-suffix, used ............................... 47
unsignedChar sort .................................... 273
unsignedInt sort ...................................... 273
unsignedInt trait function ........................ 272
unsignedLong sort .................................... 273
unsignedLong trait function ...................... 272
unsignedShort sort .................................. 273
unsignedShort trait function ..................... 272
untyped objects ...................................... 30
usefulness of Larch/C++ ............................. 6
user-defined model of a struct ................... 71
uses .................................................... 255
uses of Larch/C++ ................................. 255
uses-clause, and class templates .................. 252
uses-clause, defined ............................... 255
uses-clause, in refinement ......................... 263
uses-clause, used ..................................... 82, 216, 243, 254
uses-seq, defined .................................... 82
uses-seq, used ........................................ 82
using .................................................. 78
using .................................................. 157
using .................................................. 251
using-definition, defined .......................... 78
using-definition, used .............................. 78, 216, 254
using-directive, defined ........................... 78
using-directive, used ................................ 78, 254
using-trait-list, defined ........................... 157
using-trait-list, used ............................... 73, 251
utility of Larch/C++ ................................. 6

V
valid pointer .......................................... 290
value .................................................... 21, 30, 103
value parameter ...................................... 300, 303
value, abstract ....................................... 4, 168
value, initial ......................................... 56
variable ............................................... 65
variable, array ....................................... 69
variable, class ........................................ 69
variable, global ...................................... 134
variable, global, declaration of .................... 54
variable, pointer ..................................... 68
variable, reference ................................... 67
variable, structure (struct) ....................... 69
variable, union ....................................... 72
variables, global, sorts of (table) ............... 76
variant record types ................................. 302
varId, defined ....................................... 88
varId, used ........................................... 88, 94, 136
VDM ..................................................... 4
verification ........................................... 269
verification, modular ................................ 215
vertical tab ............................................ 36
viewpoint .............................................. 15
virtual .................................................. 59, 171, 214
virtual function, pure ............................. 234
virtual, pure .......................................... 59
visibility ............................................... 78
visible state .......................................... 104, 196
vocabulary ............................................ 2
void ..................................................... 60, 100, 278
void type ............................................. 278
volatile ............................................... 59

W
wchar_t .................................................. 49, 275
weak behavioral subtype, example ................ 223
weak behavioral subtypes and inheritance of history
  constraints ........................................... 227
weak behavioral subtyping .......................... 223
weakly ................................................... 223
well-defined .......................................... 107
where .................................................... 243
where clause .......................................... 239
where-body, defined .................................. 243
where-body, used ..................................... 243
where-clause, defined ............................... 243
where-clause, used .................................... 243
where-seq, defined ................................... 243
where-seq, used ...................................... 239
white space .......................................... 36
white-space, defined ............................... 36
white-space, used ................................. 36
wide character type interface specification 275
wide characters ................................. 275
wide-character string ......................... 50
widening .......................................... 26
Wills ............................................. 233
Windows 95, port of Larch/C++ for .......... 10
Windows 97, port of Larch/C++ for .......... 10
Windows NT, port of Larch/C++ for ......... 10
with ............................................... 257
word, reserved ................................. 40
worlds, that identifiers live in ............... 40
WWW page, for Larch ............................ 12

X
x .................................................. 47, 49
X .................................................. 47
xor ............................................... 52
xor_eq ........................................... 52

Z
Z .................................................. 4
# Table of Contents

1 **Introduction** .............................................. 1  
   1.1 Larch-style Specifications .......................... 2  
   1.2 What is Larch/C++ Good For? ..................... 6  
   1.3 Status and Plans for Larch/C++ ................. 8  
   1.4 Larch/C++ Tools .................................... 9  
      1.4.1 Obtaining and Installing the Larch/C++ Release .... 10  
      1.4.2 Obtaining LSL and LP .......................... 11  
      1.4.3 Typical Use of the Tools .................... 12  
   1.5 Acknowledgements ................................... 12  

2 **Fundamental Concepts** ................................. 15  
   2.1 Viewpoint ........................................... 15  
   2.2 Interfaces .......................................... 15  
   2.3 Accessibility of Class Members in Specifications .......... 16  
   2.4 Modules and Files ................................. 17  
   2.5 Declarations and Definitions .................... 18  
   2.6 Scope Rules ....................................... 18  
   2.7 Types and Sorts .................................... 20  
   2.8 Objects and Values ................................ 21  
      2.8.1 Formal Model of Objects ..................... 24  
         2.8.1.1 Formal Model of Mutable Objects ........ 28  
         2.8.1.2 Formal Model of Constant Objects ...... 29  
      2.8.2 Formal Model of States ..................... 30  
   2.9 Satisfaction ...................................... 33  

3 **Syntax Notation** ......................................... 35  

4 **Lexical Conventions** .................................... 36  
   4.1 White Space ....................................... 36  
   4.2 Comments .......................................... 36  
   4.3 Annotations ....................................... 38  
   4.4 Pragmas ........................................... 38  
   4.5 Tokens ............................................ 39  
   4.6 Identifiers ....................................... 39  
   4.7 Simple-Ids ....................................... 40  
   4.8 Keywords ......................................... 40  
   4.9 Context-Dependent Keywords ..................... 42
4.10 Special Symbols ......................................................... 43
  4.10.1 Always Special Symbols ........................................ 44
  4.10.2 C++ Declaration Symbols ....................................... 44
  4.10.3 C++ Operator Symbols .......................................... 44
  4.10.4 Predicate Special Symbols ...................................... 45
  4.10.5 LSL Operators .................................................. 45
4.11 Keywords in Predicates ............................................... 46
4.12 Informal Comments .................................................. 46
4.13 Literals ............................................................... 47
  4.13.1 Integer Constants ............................................... 47
  4.13.2 Floating Constants .............................................. 48
  4.13.3 Character Constants ........................................... 49
  4.13.4 String Constants ............................................... 50
  4.13.5 Abstract String Constants .................................... 51
4.14 LSL Constants ....................................................... 51
4.15 Alternative Tokens .................................................. 52

5 Declarations .......................................................... 54
  5.1 Initializers .......................................................... 56
  5.2 Declaration Specifiers ............................................... 57
    5.2.1 Storage Class Specifiers ...................................... 58
    5.2.2 Function Specifiers ........................................... 59
    5.2.3 Type Specifiers ............................................... 59
      5.2.3.1 Simple Type Names ..................................... 60
      5.2.3.2 Class and Namespace Names ............................ 61
      5.2.3.3 Elaborated Type Specifiers ............................ 62
    5.2.4 Friend .......................................................... 63
    5.2.5 Typedef Specifiers ............................................ 63
  5.3 Enumeration Declarations ......................................... 64
  5.4 Declarators .......................................................... 65
    5.4.1 Reference Declarations ...................................... 67
    5.4.2 Pointer Declarations ........................................ 68
    5.4.3 Array Declarations .......................................... 69
    5.4.4 Structure and Class Declarations ........................... 69
    5.4.5 Union Declarations .......................................... 72
    5.4.6 Function Declarations ....................................... 72
    5.4.7 Constant Declarations ...................................... 75
    5.4.8 Summary of Declarations .................................... 76
  5.5 C++ Namespace and Using Declarations .......................... 78
  5.6 Linkage Declarations .............................................. 80
  5.7 Asm Definitions .................................................... 81
6 Function Specifications ............................................. 82

6.1 Predicates ................................................................. 86
  6.1.1 If then else .............................................................. 86
  6.1.2 Logical Connectives ............................................... 87
  6.1.3 Equality Terms and Quantifiers ................................. 87
    6.1.3.1 Equality Terms ............................................. 88
    6.1.3.2 Quantifiers ................................................ 88
  6.1.4 Informal Descriptions ........................................... 91
  6.1.5 LSL Operator Terms .............................................. 92
  6.1.6 Brackets and Braces .............................................. 93
  6.1.7 Primaries ............................................................ 93
  6.1.8 Primitives ........................................................... 94
    6.1.8.1 Sorts for Formal Parameters .............................. 94
  6.1.9 Primary Suffixes .................................................. 98
  6.1.10 Larch/C++ Special Primaries .................................. 99
    6.1.10.1 This and Self ............................................ 99
    6.1.10.2 Result .................................................... 100
    6.1.10.3 Names of States (pre, post, and any) .................. 101
 6.2 Mutation ............................................................... 101
    6.2.1 State Functions ............................................... 103
    6.2.2 Allocated and Assigned ...................................... 107
    6.2.3 The Modifies Clause .......................................... 110
      6.2.3.1 Constructs .............................................. 113
      6.2.3.2 Syntactic Sugars in the Modifies Clause .............. 113
      6.2.3.3 Modifies and Const ................................... 115
      6.2.3.4 Formal Details of the Modifies Clause .............. 116
      6.2.3.5 Reach .................................................. 120
      6.2.3.6 Unchanged ............................................... 121
 6.3 Allocation and Deallocation ....................................... 123
    6.3.1 Fresh ............................................................ 123
    6.3.2 The Trashes Clause ........................................... 126
      6.3.2.1 Trashed ................................................ 128
      6.3.2.2 Formal Details of the Trashes Clause ............... 130
 6.4 The Calls Clause ................................................... 132
 6.5 The Accesses Clause ................................................. 133
 6.6 Default Arguments .................................................. 134
 6.7 Global Variables ..................................................... 134
 6.8 Let Clauses ........................................................... 135
 6.9 Redundancy in Function Specifications .......................... 137
    6.9.1 Examples in Function Specifications ....................... 137
    6.9.2 Redundant Requires Clauses ................................ 140
    6.9.3 Redundant Ensures Clauses or Claims ..................... 141
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.9.4</td>
<td>Redundancy in Frames</td>
<td>143</td>
</tr>
<tr>
<td>6.10</td>
<td>Case Analysis</td>
<td>143</td>
</tr>
<tr>
<td>6.11</td>
<td>Exceptions</td>
<td>148</td>
</tr>
<tr>
<td>6.12</td>
<td>Liberal Specifications</td>
<td>151</td>
</tr>
<tr>
<td>6.12.1</td>
<td>Terminates</td>
<td>152</td>
</tr>
<tr>
<td>6.12.2</td>
<td>Liberal Specification and Case Analysis</td>
<td>153</td>
</tr>
<tr>
<td>6.12.3</td>
<td>Examples of Liberal Specification</td>
<td>154</td>
</tr>
<tr>
<td>6.12.4</td>
<td>Meaning of Function Specifications</td>
<td>156</td>
</tr>
<tr>
<td>6.13</td>
<td>Specifying Higher-Order Functions</td>
<td>157</td>
</tr>
<tr>
<td>6.14</td>
<td>Behavior Programs</td>
<td>164</td>
</tr>
<tr>
<td>7</td>
<td>Class Specifications</td>
<td>167</td>
</tr>
<tr>
<td>7.1</td>
<td>Examples of Class Specifications</td>
<td>168</td>
</tr>
<tr>
<td>7.1.1</td>
<td>A First Class Design (Person)</td>
<td>168</td>
</tr>
<tr>
<td>7.1.2</td>
<td>A Design with a Nontrivial Trait (Money)</td>
<td>175</td>
</tr>
<tr>
<td>7.1.3</td>
<td>A Class with Exceptions (Stack)</td>
<td>183</td>
</tr>
<tr>
<td>7.2</td>
<td>Class Member Specifications</td>
<td>186</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Constructors</td>
<td>188</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Destructors</td>
<td>190</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Implicitly Generated Member Specifications</td>
<td>191</td>
</tr>
<tr>
<td>7.3</td>
<td>Class Invariants</td>
<td>195</td>
</tr>
<tr>
<td>7.4</td>
<td>History Constraints</td>
<td>200</td>
</tr>
<tr>
<td>7.5</td>
<td>Contained Objects</td>
<td>207</td>
</tr>
<tr>
<td>7.6</td>
<td>The Depends Clause</td>
<td>213</td>
</tr>
<tr>
<td>7.7</td>
<td>The Represents Clause</td>
<td>213</td>
</tr>
<tr>
<td>7.8</td>
<td>Specifying Derived Classes</td>
<td>214</td>
</tr>
<tr>
<td>7.9</td>
<td>Inheritance of Specifications and Subtyping</td>
<td>214</td>
</tr>
<tr>
<td>7.9.1</td>
<td>Inheritance of Specifications with Specification Variables</td>
<td>216</td>
</tr>
<tr>
<td>7.9.2</td>
<td>Inheritance with Explicitly-Given Traits and Weak Subtyping</td>
<td>223</td>
</tr>
<tr>
<td>7.9.3</td>
<td>More Details of Specification Inheritance</td>
<td>231</td>
</tr>
<tr>
<td>7.9.3.1</td>
<td>Strong vs. Weak Behavioral Subtyping</td>
<td>231</td>
</tr>
<tr>
<td>7.9.3.2</td>
<td>Simulation Functions that Need a State</td>
<td>232</td>
</tr>
<tr>
<td>7.9.4</td>
<td>Related Work on Inheritance of Specifications</td>
<td>233</td>
</tr>
<tr>
<td>7.10</td>
<td>Abstract Classes</td>
<td>234</td>
</tr>
<tr>
<td>7.11</td>
<td>Specifying Exposed Data Members</td>
<td>235</td>
</tr>
<tr>
<td>7.12</td>
<td>Specifying Friends</td>
<td>237</td>
</tr>
</tbody>
</table>
8 Template Specifications ................................. 239
  8.1 Example Template without Requirements ............... 240
  8.2 Requirements on Template Parameters .................. 243
  8.3 Instantiation of Templates ............................. 251

9 Specification Modules ................................. 254
  9.1 Ghost Variables ..................................... 254
  9.2 The Uses Clause .................................... 255

10 Refinement ............................................ 257
  10.1 Function Refinement .................................. 258
  10.2 Class Refinement .................................... 260
  10.3 Specifying Protected and Private Interfaces ............ 263
  10.4 Template Refinement .................................. 267
  10.5 Namespace Refinement .................................. 267
  10.6 Nested Refinement .................................... 267

11 Built-in Types ........................................ 268
  11.1 Integer Types ....................................... 268
    11.1.1 Signed Trait .................................... 268
    11.1.2 Short Integer Trait ............................... 269
    11.1.3 Long Integer Trait ................................. 270
    11.1.4 Char Trait ........................................ 271
    11.1.5 int Trait ......................................... 271
    11.1.6 Unsigned Integer Trait ............................. 271
    11.1.7 Summary of Trait Functions for Integer Traits ....... 273
  11.2 Wide Characters ...................................... 274
  11.3 Floating Point Types .................................. 274
  11.4 bool .................................................. 275
  11.5 void .................................................. 277
  11.6 Enumeration Types .................................... 277
  11.7 Array Types .......................................... 278
  11.8 Pointer Types ........................................ 286
  11.9 Character Strings .................................... 295
  11.10 Structure and Class Types ............................. 298
  11.11 Union Types ......................................... 301
  11.12 Function Types ...................................... 304
Appendix A  Grammar Summary ......................... 306
  A.1  Lexical Conventions .................................. 306
  A.2  Declarations ........................................... 309
  A.3  Function Specifications ............................... 312
  A.4  Class Specifications .................................. 314
  A.5  Template Specifications ............................... 315
  A.6  Specification Modules ................................ 316
  A.7  Refinement ............................................. 317

Appendix B  Bibliography ................................. 318

Appendix C  Differences .................................... 325
  C.1  Differences Between Larch/C++ and Larch/C ......... 325
  C.2  Differences Between Larch/C++ and C++ ............. 326

Appendix D  Deprecated ................................... 327
  D.1  Deprecated Syntax .................................... 327

Example Index .............................................. 328

Concept Index ............................................. 332