

Software Component Engineering

With Resolve/C++

Volume 2: The Client's View

Draft of June 5, 2007

Bruce W. Weide
Department of Computer Science and Engineering
The Ohio State University
Columbus

Copyright © 2007 by the author. All rights reserved.

Contents

1	ABSTRACT AND CONCRETE INSTANCES	1
1.1	Resolve/C++ Software Components	1
1.1.1	Subroutine Libraries	1
1.1.2	User-Defined Types	2
1.1.3	Classes	3
1.1.4	Components and Catalogs	3
1.1.5	Component File Naming Conventions	4
1.2	Example: Monte Carlo Estimation of π	4
1.2.1	Bringing a Concrete Instance Into Scope	7
1.2.2	Using the Concrete Instance	8
1.3	Anatomy of a Component Family: Random	8
1.3.1	The Kernel Type, Standard Operations, and Kernel Operations	9
1.3.2	The Extends Relation (Abstract-to-Abstract)	13
1.3.3	Using a Concrete Component	15
1.4	Summary	16
2	ABSTRACT AND CONCRETE TEMPLATES: GENERALIZATION	19
2.1	Parameterized Software Components	19
2.1.1	Resolve/C++ Language Support for Templates	21
2.1.2	Collection Components	22
2.1.3	Template Instantiation	22
2.2	Example: Polynomial Evaluation	23
2.2.1	Bringing a Concrete Template Into Scope	30
2.2.2	Instantiating the Concrete Template to Create a Concrete Instance	31
2.2.3	Using the Resulting Concrete Instance	31
2.3	Anatomy of a Component Family: Sequence	34
2.3.1	The Kernel Type, Standard Operations, and Kernel Operations	34
2.3.2	The Extends Relation (Abstract-to-Abstract)	36
2.3.3	Using a Concrete Component	38
2.4	Summary	38
3	ABSTRACT AND CONCRETE TEMPLATES: DECOUPLING	41
3.1	The Problem of Component Dependence	41
3.1.1	Avoidable Dependence Is Bad	41
3.1.2	Avoidable Concrete-to-Concrete Dependence Is Worse	42
3.2	Decoupling the Checks Relation	44
3.2.1	The Defensiveness Dilemma	47
3.2.2	Example: The Natural Family	51
3.2.3	Checking Components and the Checks Relation	53
3.2.4	Implementing a Checking Component	55

3.2.5	Instantiating a Checking Component	57
3.2.6	Combining Generalization and Decoupling: The Specializes Relation	58
3.3	Decoupling the Extends Relation (Concrete-to-Concrete)	62
3.3.1	The Extends Relation (Concrete-to-Concrete): The Layering Approach	63
3.3.2	Instantiating a Concrete Extension	66
3.4	Summary	68

CLIENT-VIEW APPENDICES

Array

Binary_Tree

Id_Name_Table

List

Natural

Partial_Map

Queue

Record

Sequence

Set

Sorting_Machine

Stack

Static_Array

Text

Tree

1 Abstract and Concrete Instances

Suppose your job is to write a Resolve/C++ application program. At first blush, it seems no more complex than the examples in Volume 1. Yet after thinking about it, you conclude that the program calls for functionality beyond what is provided directly by the built-in Resolve/C++ types—not just narrow application-specific capabilities but ones that would be useful in many other situations, too. Some of the required functionality seems so general and valuable that it probably has been written before! Hoping to do as little work as possible, you imagine you might be able to acquire significant pieces of the program from a repository of *common off-the-shelf* (or *COTS*) components.

Volume 2 of *Software Component Engineering* deals with this situation from the perspective of a client programmer. What's different from the scenarios in Volume 1? Now you have at your disposal a significant set of general-purpose software components that far surpass in functionality the Resolve/C++ built-in types. Each component comes with an interface contract specification to describe its behavior, and multiple implementations of that behavior with different performance profiles. This chapter shows you how to use them.

1.1 Resolve/C++ Software Components

The Resolve/C++ taxonomy of software components is this:

- An **abstract instance** is an interface contract specification: a description of behavior in “abstract” terms that do not reveal the details of how that behavior is achieved.
- A **concrete instance** is an implementation of an interface contract: a description of how to achieve the specified behavior.
- An **abstract template** is a *parameterized* interface contract specification: a generator for a group of similar abstract instances.
- A **concrete template** is a *parameterized* implementation of an interface contract: a generator for a group of similar concrete instances.

Every Resolve/C++ software component, then, is an abstract or concrete instance or template. This chapter introduces abstract and concrete instances. Chapters 2 and 3 describe how components that are abstract and concrete templates add substantial generality and power. The appendices summarize the interface contract specifications for a number of general-purpose Resolve/C++ components.

1.1.1 Subroutine Libraries

What specifically is the nature of a software component in Resolve/C++? Based on Volume 1, there is one obvious choice for what an off-the-shelf software component should be: a global operation. Throughout the early history of computing, this is precisely what off-the-shelf software components were. A collection of operations that can be used by other (client) programmers is known as a **subroutine library**: interface contract specifications are provided to client programmers, but implementations generally are kept secret because a client programmer has no reason to see them.

One of the problems with limiting component collections to being subroutine libraries is that the types of the parameters are limited to a fixed set of built-in types. In Resolve/C++, for example, every global operation in a subroutine library would have only parameters of type *Boolean*, *Character*, *Integer*, *Real*, *Text*, *Character_IStream*, or *Character_OStream*.

This might not seem too restrictive at first, but it leads to surprisingly difficult design and maintenance problems. For example, suppose you are dealing with an application program that involves complex numbers. These are numbers of the form $a + b \bullet i$ where a and b are real numbers and $i^2 = -1$; notice that i is not a real number. There is a rich mathematical theory of complex numbers and special notation for dealing with them. How do you make the connection between your application program's complex numbers and the operations available in the subroutine library?

If there are operations in the subroutine library whose abstract descriptions mention complex numbers, each parameter to these operations must be of one of the built-in types (e.g., *Real*). Therefore you, as a client programmer, may not think of a complex number as a single entity. Every time you want to pass a complex number as an argument, you must pass two real numbers.

Just to make matters a bit worse in this case, there are at least two completely different but plausible ways that two *Real* objects can be used to represent a complex number. One is the way suggested above: the two real numbers are a and b , the “real” and “imaginary” parts of the complex number. Another method is to consider the two real numbers as the “polar coordinates” of the complex number. The details of how these representations work and their relative merits are beyond the scope of this discussion—but they are not the issue. The point is that merely saying that a complex number is represented by using two *Real* objects does not pin things down enough. A client programmer must know which representational scheme is being used by which operations, and if different representations are used for different operations in the subroutine library then the client programmer must convert manually between them. In short, this is a mess.

If you could view an object whose value is a complex number as a unit—as a single object with an abstract state described in terms of the mathematical theory of complex numbers—then you wouldn't need to be concerned with such issues.

1.1.2 User-Defined Types

The problem outlined above suggests that a subroutine library might not be the best way of organizing off-the-shelf software components. Even if it works well enough in limited application domains, focusing on individual operations generally is not the right way to think about software components. Experienced software engineers have recognized for decades the need for a richer set of types than what is built-in to most programming languages. No *fixed* set of types, however large or sophisticated, can possibly be just what it takes to develop application programs that no one has ever thought of before. The requisite level of generality can only be obtained if a software engineer can design and implement *new* types and associated operations that fit the situation at hand, when that is appropriate.

Such a new type—it might be called *Complex* in the scenario above—is called a ***user-defined type***, to contrast it with a built-in type. A client programmer developing a new application might not need to create any new types. Often, he or she can just retrieve types that are generally useful from a catalog of off-the-shelf software components developed by others.

Recall from Volume 1 that the type of an object determines both the mathematical model of the object's value (state) and the operations that a client programmer may call in order to observe

and to control the value of that object. Given the central role of types, then, it seems natural to consider a type's mathematical model and its associated operations to be the basis for a software component. This is the view taken in object-oriented programming in general, and in the Resolve/C++ discipline in particular.

1.1.3 Classes

C++ offers a program unit called a *class* in which you can describe either a type's mathematical model and its associated operations (via an interface contract specification), or a method of achieving that behavior (via implementation code). This unit is introduced with the keyword **class**, prefixed in the Resolve/C++ discipline by a qualifying keyword¹ to indicate which category it belongs to in the Resolve/C++ taxonomy of components: **abstract_instance**, **concrete_instance**, **abstract_template**, or **concrete_template**. The chapter discusses only instances; templates are introduced in Chapter 2.

1.1.4 Components and Catalogs

We can now say what a software component actually is. A *Resolve/C++ software component* is a C++ class, of any of the four kinds listed above. A *component catalog* is a collection of abstract and concrete instances and templates. In practice, each component is stored in a file, and the files defining the components in a catalog are organized within a directory for that catalog. The *Resolve/C++ Catalog* is the primary collection of general-purpose Resolve/C++ components. These are not quite as widely useful as the built-in types and operations, but they are still valuable in a variety of client programs. There can be other catalogs of special-purpose components. In fact, you also may set up your own catalogs, but this is not discussed here.

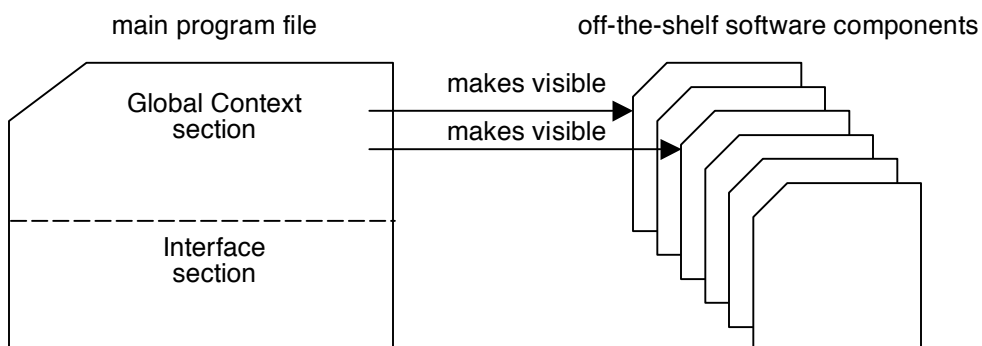


Figure 1-1: A main program file that brings into scope two components

In order to use an off-the-shelf component from one of these catalogs, you first need to bring the component *into scope*, or *make it visible* in your program, so the C++ compiler knows about the component. To do this, you simply write a **#include** statement (by convention, in the section of your program labelled “Global Context”) giving the name of the file containing the component. This is particularly easy for Resolve/C++ Catalog components. In order to avoid the hassle of having to learn and write a complete file name—which might involve a long path, the details of which might vary from system to system—in Resolve/C++ you give the name of a component file relative to the directory where Resolve/C++ Catalog components reside. For special

¹ In C/C++ programs, you would use just the keyword **class**, whether the component being defined is an interface contract specification or an implementation, an instance or a template.

catalogs, you might have to provide the catalog's directory name, too, or follow other catalog-specific instructions for how to include its components. On the other hand, as you develop new components that are not yet ready to be placed in a catalog and to be used by other people in their programs, you should include them by a path to one of your own private directories.

1.1.5 Component File Naming Conventions

As a client programmer using a catalog of off-the-shelf components, then, you need to know the names of the files in which the required components are described. The file names within a catalog directory arise from the following Resolve/C++ conventions, which are based on the taxonomy of component varieties and on the class names used for particular components:

- Within a catalog directory, a component file resides in one of four subdirectories, based on whether it is an abstract or concrete instance or template:
 - The subdirectory called “AI” contains components that are abstract instances.
 - The subdirectory called “CI” contains components that are concrete instances.
 - The subdirectory called “AT” contains components that are abstract templatees.
 - The subdirectory called “CT” contains components that are concrete templatees.
- Within the component-kind directory, a component file resides in a subdirectory whose name is that of a *component family*, i.e., a set of logically related components. For example, for a component defining a new programming type for complex number objects, this subdirectory name probably would be “Complex”. All the components in the directory would deal with such objects and their class names would begin with the prefix *Complex_*.
- Within the component-family directory, an individual component file has a name that is the remainder of the class name (i.e., whatever comes after the component family name and the underscore separator) and a “.h” extension. For example, the abstract instance that defines a programming type for complex number objects along with the most important operations for such objects probably would be called *Complex_Kernel*. As an abstract instance, it would be in the component file AI/Complex/KERNEL.h. One possible concrete instance that implements *Complex_Kernel* might be called *Complex_Kernel_1* (for implementation #1 of *Complex_Kernel*). If so, it would be in the component file CI/Complex/KERNEL_1.h.

1.2 Example: Monte Carlo Estimation of π

Let's look at an example of how you can use an off-the-shelf component to solve an application problem.

Among the many interesting ways to estimate things is to take a random sample. For instance, suppose you want predict what fraction of votes will be cast for candidate X in an election. You don't want to ask everyone how he/she will vote because that would be much too expensive. An appropriate approach is to conduct an “opinion poll”, i.e., to select a small (but not too small) random sample of potential voters, and to use the fraction of the sample respondents who plan to vote for candidate X as an estimate of the fraction of all voters who plan to vote for candidate X.

The same idea can be used to estimate other things, e.g., the value of π . Figure 1-2 illustrates this. Suppose you generate random points within a unit square. Then the fraction of those points that fall within the quarter circle Q can be used to compute an estimate for π . How? The expected number of points that fall within the unit square that also fall within some region inside

it is equal to the area of that inner region (divided by the area of the unit square, which is 1). For example, half the points are expected to lie in the top half of the unit square, half are expected to lie in the left half of the unit square, and so on.

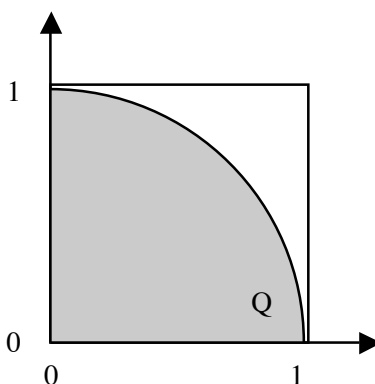


Figure 1-2: Estimating π

How many points are expected to lie within Q ? The area of Q is $\pi/4$, so $\pi/4$ is the expected fraction of points in the unit square that are also in Q . This leads to the conclusion that if you observe that a fraction f of random points lying in the unit square also lie in Q , then $4f$ is a reasonable guess² for the value of π . This is known as a *Monte Carlo estimate* because it is a lot like gambling; only here, you rely on randomness to work for you.

The user's manual for an application program that uses this method might be:

Run the program by typing the command "Monte_Carlo". The program asks for a positive integer which is the number of points to use for a Monte Carlo estimate of π , then reports the resulting estimate of π , and then quits.

There are many ways to estimate π , nearly any of which is a lot more efficient and effective than this one, if not as much fun. If the end user really just wants you to report an approximate value for π then there is no reason to prescribe that it should be done this way. Nonetheless, the Monte Carlo approach is worth knowing (for other applications), and the need for and use of random numbers are the key features this example is intended to illustrate. Estimating π is just a particularly simple use of Monte Carlo techniques.

Faced with designing this program, you would no doubt quickly come to wonder whether there is already a way to generate random numbers. Certainly *someone* must have encountered this problem before, and perhaps you can make use of their solution. What you are looking for is a previous solution that has been captured as a unit, with two separate descriptions:

- an abstract description of behavior that you can consult as a client programmer; and
- a concrete description that you can include in your Resolve/C++ program so the computer can execute the operations you invoke.

Fortunately, there are indeed off-the-shelf software components to make this application program fairly straightforward. The program that follows illustrates this. **Highlighted code** is discussed in the text.

² How reasonable the estimate is depends on how many points you use. Careful treatment of this question is beyond the scope of the current discussion.

```

// /*-----*\
// |   Main Program: Monte Carlo estimate of pi
// | *-----*\
// |   Date:           14 August 1996 (revised 24 November 2006)
// |   Author:         Bruce W. Weide
// |
// |   Brief User's Manual:
// |   Asks for a positive integer, which is the number of
// |   points to use for a Monte Carlo estimate of pi, then
// |   reports the resulting estimate of pi, and then quits.
// |
// \*-----*/

///-----
/// Global Context -----
///-----

#include "RESOLVE_Foundation.h"

#include "CI/Random/Uniform Generator 1.h"

///-----
/// Interface -----
///-----

program_body main ()
{
    object Character_IStream input;
    object Character_OStream output;
    object Random Uniform Generator 1 rn;
    object Integer n, pts_in_U, pts_in_Q;
    object Real pi_estimate;

    // Open input and output streams

    input.Open_External ("");
    output.Open_External ("");

    // Ask user for number of points

    output << "Number of points: ";
    input >> n;

    // Create counts for estimate

    while (pts_in_U < n)
    /*!
        preserves n
        alters rn, pts_in_U, pts_in_Q
        maintains
            0 <= pts_in_Q <= pts_in_U <= n and
            pts_in_Q = [number of points that have been
                       in the quarter-circle] and
            pts_in_U = [number of points that have been
                       in the unit square]
        decreases

```

```

        n = pts_in_U
    */
    {
        object Real x, y;

        // Generate a random point in the unit square U

        rn.Generate_Next ();
        x = rn.Uniform_Real (0.0, 1.0);
        rn.Generate_Next ();
        y = rn.Uniform_Real (0.0, 1.0);
        pts_in_U++;

        // Check if point is also in the quarter circle Q

        if (x * x + y * y < 1.0)
        {
            pts_in_Q++;
        }
    }

    // Estimate pi

    pi_estimate = 4.0 * To_Real (pts_in_Q) / To_Real (n);
    output << "pi is about equal to " << pi_estimate << "\n";

    // Close input and output streams

    input.Close_External ();
    output.Close_External ();
}

```

Figure 1-3: Monte_Carlo.cpp

This program introduces no substantial new features of the Resolve/C++ discipline except how to make visible, and then how to use, types and operations that are not built-in but rather are defined by off-the-shelf software components.

1.2.1 Bringing a Concrete Instance Into Scope

The following line:

```
#include "CI/Random/Uniform_Generator_1.h"
```

brings into scope the code that implements the operations defined by the concrete instance called *Random_Uniform_Generator_1*. As a class name, *Random_Uniform_Generator_1* is also the name of the type defined by this concrete component, so this type name is now in scope. Furthermore, the names of all the operations provided by this concrete component are in scope. This means that you may now declare objects of type *Random_Uniform_Generator_1* and may call the operations provided by concrete instance *Random_Uniform_Generator_1*.

Why do you bring into scope the concrete component, as opposed to the abstract component that it implements? For human consumption, you could include just the abstract component that provides the cover story, i.e., the interface contract specification. But you *always* need to include the concrete component if you plan to declare objects of the provided type and call provided operations because the compiler needs this information in order to generate executable program code.

In principle, a concrete instance could be stored in a “.cpp” (C++ separately compiled) file, which could be compiled once into an *object code library* that is to be *linked* with your separately compiled application program. This would save a little compilation time in client programs that use the component. In practice, however, almost all Resolve/C++ components are templates rather than instances, and “.h” (C++ header) files are required for templates because C++ templates are not separately compiled. For uniformity and simplicity, every component discussed in this book is defined in a “.h” file—whether it is a template or an instance.

When you bring a concrete instance into scope, the abstract instance that contains the interface contract specification comes along for the ride *by transitivity* because the concrete component **#includes** the abstract component it implements. In fact, by virtue of the Resolve/C++ discipline’s rules for component design and coding, explicitly including a component with **#include** automatically brings into scope *all* the other components—abstract and concrete—on which it depends either directly or indirectly.

1.2.2 Using the Concrete Instance

How do you use an off-the-shelf component in a client program? There is a simple rule: once you have brought an off-the-shelf concrete component into scope as described above, you may use the type and operations exactly as though they were built-in. You may declare objects of the type provided by the component, and call the operations provided by the component, exactly as you do with the built-in types and their operations.

To illustrate, Monte_Carlo declares an object *rn* of type *Random_Uniform_Generator_1*, and calls its operations *Generate_Next* (which generates a new random number held in *rn*) and *Uniform_Real* (which transforms the random number held in *rn* into a corresponding real number that lies between this function’s two arguments). If you didn’t know that this type and these operations were not built-in to Resolve/C++, you could not tell by looking at the code that uses them. You reason about the behavior of the client program in precisely the same way, too. Objects of user-defined types are, in every important sense, first-class citizens with the same status as objects of the built-in types. This property means there is uniformity and consistency of client code even in very large applications with many user-defined types.

1.3 Anatomy of a Component Family: *Random*

From a client programmer’s perspective, you are most interested in the abstract instance(es) that explain each concrete instance used in your program: the interface contract that you need in order to use a component. This section focuses on the off-the-shelf component *Random_Uniform_Generator_1* used in Monte_Carlo, as an example to illustrate the structure and meanings of the abstract component(s) that specify the functionality of this concrete component. Other than its name, there is no need for you as a client programmer to know anything more than this about the concrete instance that is **#included**.

The Resolve/C++ discipline advises that you explain the behavior of a possibly complex concrete instance by dividing the overall abstract description into “bite-sized” abstract instances. The main abstract instance specifies a new user-defined *kernel type*, by stating its mathematical model, along with the *standard operations* and *kernel operations* that can be applied to objects of that type. Other operations called *extensions* can be added in other abstract instances, the total behavior being the “sum” or “union” of the behaviors described in these smaller pieces.

1.3.1 The Kernel Type, Standard Operations, and Kernel Operations

Figure 1-4 shows the entire file contents for the abstract component called *Random_Kernel*. All abstract instance files that introduce new kernel types are identical in structure to this one.

```
// /*-----*\
// |   Abstract Instance : Random_Kernel
// \*-----*/

#ifndef AI_RANDOM_KERNEL
#define AI_RANDOM_KERNEL 1

///-----
/// Interface -----
///-----

abstract_instance
class Random_Kernel
{
public:

    /*!
       math definition LIMIT: integer satisfies restriction
       0 < LIMIT <= MAXIMUM_INTEGER

       math subtype RANDOM_MODEL is integer
       exemplar rn
       constraint
       0 <= rn < LIMIT

       math definition NEXT (
           n: RANDOM_MODEL
       ): RANDOM_MODEL satisfies restriction
       [successive application of NEXT starting with n
        (i.e., NEXT(n), NEXT(NEXT(n)), and so forth)
        produces a sequence of pseudo-random numbers]

    !*/

    standard_abstract_operations (Random_Kernel);
    /*!
       Random_Kernel is modeled by RANDOM_MODEL
       initialization ensures
       self = 0

    !*/

    procedure Set_Value (
        preserves Integer n
    ) is_abstract;

    /*!
       ensures
       self = n mod LIMIT

    !*/

    function Integer Value () is_abstract;
    /*!
       ensures
       Value = self
```

```

    /*/

    procedure Generate_Next() is_abstract;
    /*!
        ensures
            self = NEXT (#self)
    /*/

    function Integer Limit () is_abstract;
    /*!
        ensures
            Limit = LIMIT
    /*/

};

#endif // AI_RANDOM_KERNEL

```

Figure 1-4: AI/Random/Kernel.h

The stylized comment block at the beginning says this file defines an abstract instance whose name is *Random_Kernel*. This kind of comment is always the first thing in a Resolve/C++ component. The middle line of the block contains the kind of component, followed by a colon, followed by the abstract instance name. Having a strict convention about even this aspect of what the file contains is helpful in two ways:

- It gives a reader of the file some useful overview information in a form that he or she is familiar with, because all Resolve/C++ components look like this.
- It facilitates building tools that process files containing Resolve/C++ components. (Of course, the C++ compiler doesn't care about these comments; we're talking about other tools that might process component files for purposes such as convenient browsing through a component catalog.)

The two lines that follow the opening comment block (starting with **#ifndef** and **#define**), and the very last line (starting with **#endif**), are part of a standard “idiom” of C++. One problem with C++ is that if you include the same file more than once in a single compilation, then you get errors because the compiler doesn't like to see the same declarations and definitions more than once. A single off-the-shelf component might well be included in two or more different and independent subcomponents of a main program.

To prevent a component from being introduced into your program more than once, you place the main contents—of each file that might ever be included anywhere—inside a special kind of if-statement that is a directive to the compiler itself. It introduces a condition to be checked. Only if that condition holds does the compiler *actually* include the code between the if-directive and the corresponding **#endif**. The if-directive here, **#ifndef**, asks the compiler to check whether the global compile-time constant *AI_RANDOM_KERNEL* is not defined (hence, “ndef”). The name of this constant is, by convention in Resolve/C++, the kind of component—AI, CI, AT, or CT—followed by the class name in all upper-case characters; this makes it unique. The first time the file is included somewhere, this constant is not yet defined, so the compiler includes the rest of the file through the matching **#endif**. The directive just after the **#ifndef**, i.e., **#define**, defines the constant (to have the value 1, but this value is immaterial). So, each subsequent time the file is included during this compilation, the constant *AI_RANDOM_KERNEL* already has a defined value, and the compiler ignores everything from **#ifndef** through the matching **#endif**. This trick

makes sure that each included component is (really) included and compiled exactly once, even if it is (apparently) included in many different places.

The next section of a typical Resolve/C++ file is the global context section, which brings into scope any other components that this one directly depends on. Here, this section is absent, as it is for a typical abstract component that describes a kernel type. The abstract instances discussed later in this section describe additional operations and these always include other components—the ones they directly depend on—in their global context sections.

By the way, you might be wondering why `RESOLVE_Foundation.h` is not included here. You could include it without causing any trouble because it is protected against multiple inclusion by the above C++ idiom. But including it is not necessary because a “.h” file is not intended to be compiled separately. You really only need to include `RESOLVE_Foundation.h` in a program that *can* be compiled—e.g., in `Monte_Carlo.cpp` from Figure 1-3. The simple rule is to include `RESOLVE_Foundation.h` in all “.cpp” files, but nowhere else.

Next comes the interface section, which contains the declaration of **abstract_instance**³ **class** *Random_Kernel*. By convention for an abstract component that introduces a kernel type and its kernel operations, the class name ends in *_Kernel*.

Inside the block that follows is the keyword **public**, which indicates that the declarations that follow are part of the publicly-accessible interface of the class. In other words, any client of this class is permitted to use the mathematical and programming entities declared after **public**. The first of these entities are the mathematical definitions used to specify the interface contract of the programming type and operations. Here the mathematics part makes three declarations:

- Math definition *LIMIT* is a mathematical integer constant whose value is supplied by the particular concrete instance chosen to implement this abstract instance. As a client of *Random_Kernel*, all you know about *LIMIT* is that it satisfies the restriction that *LIMIT* is positive and no larger than the maximum integer representable on this computer. Note that the latter restriction is important because *LIMIT* is a mathematical integer, not a programming *Integer*, and hence could otherwise be arbitrarily large because mathematical integers know no bounds.
- Math subtype *RANDOM_MODEL* is a mathematical integer between 0 and *LIMIT*.
- Math definition *NEXT* is a mathematical function from the domain *RANDOM_MODEL* to the range *RANDOM_MODEL* that captures how a particular implementation of *Random_Kernel* generates pseudo-random numbers. *Pseudo-random numbers* are series of numbers that seem to be random, in the sense that they pass certain statistical tests of randomness, even though they are generated by a deterministic and hence reproducible algorithmic process. It is beyond the scope of this book to describe in a formal way an interface contract for this component. The key point is that a client of *Random_Kernel* needs to know only that *NEXT* describes this process and the properties those numbers will have, not the details of exactly what the series of numbers will be.

Next is a declaration that the new kernel type, like all Resolve/C++ types, has the *standard operations*:⁴

³ In C/C++ programs, you would use no keyword at all in place of the Resolve/C++ keyword **abstract_instance**.

- A **constructor**, which is an operation that is called automatically for each object of this type—without the client having to write an explicit call—when execution reaches the point where that object is declared. The purpose of the constructor is to create the representation of the object and to give it an initial value (in this example, 0). As a client, you don’t need to know anything about the constructor except the initial value it gives each newly declared object of the type, which is specified in the formal comment that follows **standard_-abstract_operations**.
- A **destructor**, which is an operation that is called automatically for each object of this type—without the client writing an explicit call—when execution leaves the block in which that object is declared. The purpose of the destructor operation is to “destroy” (a better description would be “reclaim”) the representation of the object. What does this mean? As a client programmer, you have no need to know this. You do not explicitly call the destructor anyway, and in general the details of what it does, and how, are of interest only to implementers.
- A swap operation, which exchanges the values of any two objects of this type using the operator `&=`.
- A **Clear** operation that resets the receiver object’s value to an initial value for this type.

The formal comment that follows says that this new kernel type is modeled by the mathematical subtype *RANDOM_MODEL* as defined above, and that every new object of this type has the initial value 0 when the object is declared.

Notice a very important point: the name of the type described here is the name of the class. This is potentially confusing but, alas, it is something you have to learn to live with when using C++ and other object-oriented languages.

Next is the header declaration for a procedure operation, *Set_Value*, introduced with the keyword **procedure**.⁵ At the end of the procedure declaration and before the formal comment containing its specification is the keyword **is_abstract**.⁶ This appears in the declaration of every operation in an abstract instance or an abstract template. It tells the compiler that there is no body for the procedure in this abstract component; the body will appear in some concrete component that implements this abstract component.

⁴ The **standard_abstract_operations** declaration replaces what you would otherwise write in C/C++ at this point: individual declarations of the constructor, destructor, and (if you designed the class to have them) swap operator and *Clear* operation. It also inhibits the creation of two operations that C++ would otherwise give you by default: assignment (operator `=`) and copy constructor. You could do the same thing in C/C++ by declaring the assignment operator and copy constructor to be private. This means that if you try to assign something to an object of the new type, or try to pass an object of the new type as the actual parameter where the associated formal parameter is declared without an “&”, the compiler will report an error. The default versions of assignment and copy constructor as created by C++ generally would result in incorrect behavior, so it is helpful that the C++ compiler can be made to detect these violations of the Resolve/C++ discipline.

⁵ In C/C++ programs, you would use the keywords **virtual void** to introduce a method returning no value, in place of the Resolve/C++ keyword **procedure**.

⁶ In C/C++ programs, you would use “= 0” in place of the Resolve/C++ keyword **is_abstract**.

Another interesting thing here is *self* in the ensures clause. Recall that the way you invoke an operation that is associated with a type (also known in object-oriented programming parlance as a *method*) is by saying something like this:

```
rand.Set_Value (k);
```

Here, *k* must be an object of type *Integer*, and *rand* must be an object of a type that implements *Random_Kernel*, in order for the compiler to consider the call to be valid. By way of contrast, if *Set_Value* were a global operation then you would invoke it like this:

```
Set_Value (rand, k);
```

In this situation, two formal parameters would be declared in the header of *Set_Value* so there would be two names to use in the ensures clause. But with *Set_Value* being a method for type *Random_Kernel* rather than a global operation, there is only one formal parameter and therefore only one name to use in the ensures clause. In either case, the formal parameter to which *k* is bound at the time of the call has a name: *n*. But what is *rand* bound to in the interface contract specification? For this—the implicit name of the formal parameter to which the receiver object is bound—we use *self*. What is the mode of *self*? By default, for any procedure operation, *self* is an alters-mode parameter; for any function operation, *self* is a preserves-mode parameter.

LIMIT is the number of different values that an object of type of *Random_Kernel* might have; see the math subtype *RANDOM_MODEL*. Suppose *LIMIT* = 19—which is a horribly small value for a useful random-number generator, but it is technically possible to have a concrete component that implements *Random_Kernel* and stipulates that *LIMIT* = 19. Then a tracing table for the statement above might look like this:

Statement	Object Values
	rand = 13 k = 27
rand.Set_Value (k);	
	rand = 8 k = 27

The interface contract specification says that, upon return from *Set_Value*, we know that *self* = *n mod LIMIT*. It also says that *n* is preserved. The parameter binding for this call matches *rand* with *self* and *k* with *n*. Therefore, we must see the result shown above: *rand* = 27 mod 19 = 8. So, suppose you use a concrete component that provides a type whose behavior can be explained using the abstract instance *Random_Kernel*. Then any object of that type is considered a unit whose state space is modeled as a *RANDOM_MODEL*, and whose associated operations include the standard operations as well as the kernel operations specified in *Random_Kernel*.

1.3.2 The *Extends* Relation (Abstract-to-Abstract)

In addition to a new kernel type and its standard and kernel operations, some clients might want additional operations that are not essential but rather are merely useful. Figure 1-5 shows an example. The declaration of *Random_Uniform* begins:

```
abstract_instance
class Random_Uniform :
    extends
        abstract_instance Random_Kernel
...
```

which states the dependence relationship between the new class *Random_Uniform* and the existing class *Random_Kernel*, namely, *Random_Uniform extends Random_Kernel*.

```
// /*-----*\
// |   Abstract Instance : Random_Uniform
// \*-----*/

#ifndef AI_RANDOM_UNIFORM
#define AI_RANDOM_UNIFORM 1

///-----
/// Global Context -----
///-----

#include "AI/Random/Kernel.h"

///-----
/// Interface -----
///-----

abstract_instance
class Random_Uniform :
    extends
        abstract_instance Random_Kernel
{
public:

    function Real Uniform_Real (
        preserves Real a,
        preserves Real b
    ) is_abstract;

    /*!
    ensures
        Uniform_Real = a + (b - a) * TO_REAL(self) / TO_REAL(LIMIT)
    !*/

    function Integer Uniform_Integer (
        preserves Integer j,
        preserves Integer k
    ) is_abstract;

    /*!
    ensures
        Uniform_Integer = j + TO_INTEGER (TO_REAL((k - j + 1) *
                                                    TO_REAL(self) / TO_REAL(LIMIT)))
    !*/

};

#endif // AI_RANDOM_UNIFORM
```

Figure 1-5: AI/Random/Uniform.h

The meaning of the keyword **extends**⁷ here (where it relates two abstract components) is that any concrete component that claims to implement *Random_Uniform* must implement the interface contract specified in *Random_Kernel*, and in addition it must implement the extra interface contracts for the new operations specified in *Random_Uniform*. Phrased another way, the behavior of any concrete component that can be explained using *Random_Uniform* can be (partially) explained using *Random_Kernel*; *Random_Uniform* itself contains only the explanations of two extra operations called *Uniform_Real* and *Uniform_Integer*. This style of organizing interface contract specifications is known as **specification by difference**, or **incremental specification**. Rather than putting all operation specifications in one place, you separate them into logically cohesive pieces and “assemble” them in mix-and-match fashion.

Figure 1-6 shows how to depict in a component coupling diagram, or CCD, the fact that the *extends* relation holds between these two abstract instances. Recall that the only relations captured in CCDs in Volume 1 were *uses* and *implements*. This is a new relation.

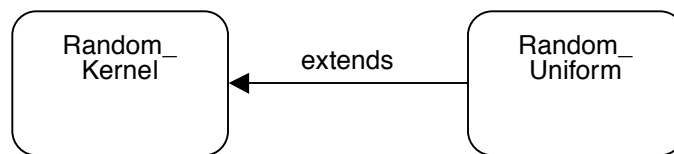


Figure 1-6: The extends relation may hold between two abstract instances

The same approach can be used to extend any abstract description with any other, so long as the functionalities being added together do not conflict. This is always the case if you follow the Resolve/C++ discipline, although it is not the case if you aren’t careful. For example, C++ will let you extend abstract instance *Random_Kernel* by adding an operation called *Set_Value* with a single *Integer* parameter and almost the same specification as above, except that the ensures clause is different. Using the rules of C++, the new definition would **override**, or replace, the original one rather than adding a new operation. There is no reason to do this when extending a component using the Resolve/C++ discipline, and you must avoid it.

1.3.3 Using a Concrete Component

As a client of the off-the-shelf concrete component used in the Monte_Carlo program, all you need to know about it is summarized in the CCD in Figure 1-7. The name of the concrete instance is *Random_Uniform_Generator_1*, and it implements *Random_Uniform* and hence also *Random_Kernel*. The interface contract specifications in the latter two components explain how *Random_Uniform_Generator_1* objects will behave in the client program: the possible values those objects can have and the operations that you can call on them. That’s it. As a client, you do not need to see the code for *Random_Uniform_Generator_1*.

⁷ In C/C++ programs, you would use the keywords **virtual public** in place of the Resolve/C++ keyword **extends**.

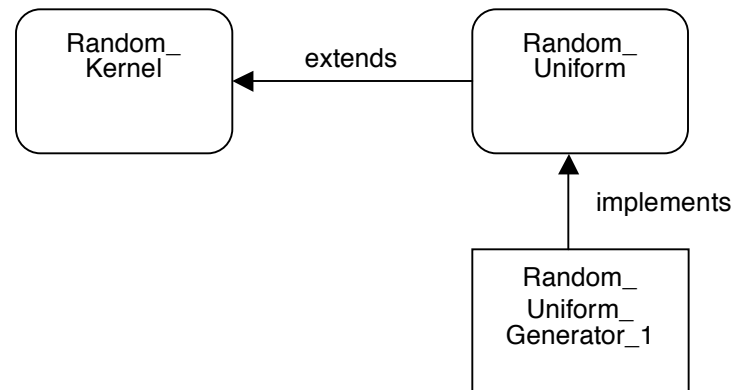


Figure 1-7: The implements relation may hold between a concrete and abstract instance

1.4 Summary

There are four kinds of Resolve/C++ software components, each of which is a class in C++: abstract and concrete instances, and abstract and concrete templates. An abstract instance defines an interface contract specification, while a concrete instance comprises code that implements such a contract. (Templates, which are effectively generators for entire families of instances, are explained in Chapters 2 and 3.)

A kernel abstract instance acts as a centerpiece for a family of related components. It defines the mathematical model of a new user-defined kernel type—which automatically has the standard operations: constructor, destructor, swap (operator &=), and *Clear*—along with the interface contracts of its kernel operations, or methods. The latter are operations that together are deemed by the designer to be both necessary and sufficient to perform all useful manipulations of objects of the new type. Other abstract instances can then be added incrementally to specify interface contracts for more operations (methods). These abstract instances are said to extend the kernel abstract instance.

Generally, a kernel abstract instance is self-sufficient, in the sense that it has no dependencies on other components. An abstract instance that specifies an extension, of course, depends (at least) on the abstract instance it extends. This dependency is recorded in the code in two places: in a **#include** statement that brings the other component into scope so it is visible to the compiler, and in an **abstract_instance class** declaration using the keyword **extends** to indicate the explicit dependence relation between the new component and the kernel abstract instance. A component coupling diagram (CCD) depicts this relationship schematically to ease human understanding.

A client program that needs to use some implementation of an abstract instance must bring into scope the particular concrete instance it wishes to use, not just the abstract instance that explains its behavior for humans. This allows the compiler to generate executable code for the client program. However, all the client programmer needs to know in order to do this is the location in the file system of the appropriate concrete instance; there is no reason to look at the Resolve/C++ code of that instance. The component's location is determined by a set of Resolve/C++ conventions: it is in a component catalog directory, then down in a component-kind directory (in this case, "CI" since the component is a concrete instance), then down in a component family directory (determined by the kernel abstract instance that the concrete instance implements), and finally in the file in that directory that contains the concrete instance code itself. General-purpose Resolve/C++ components are located in the Resolve/C++ Catalog—a default starting point as the compiler searches for components—so **#include** statements in application programs

do not include long absolute paths but rather relative paths from this fixed location. Again, a CCD concisely depicts the fact that a concrete instance implements a particular abstract instance (and, if that abstract instance specifies an extension, the kernel abstract instance that it extends). All other components that a concrete instance depends on are automatically brought into scope by transitivity, making component inclusion in a client program easy to manage.

In a client program that uses a typical concrete instance, the name of that concrete instance (a class in C++) is also the name of a type. As a client programmer, you may declare objects of that type and may invoke on them any operations in the abstract instance(s) that are implemented—subject, of course, to the interface contracts specified for those operations. Once a concrete instance is brought into scope with its **#include** statement, the new type is available to be used just like any built-in type of Resolve/C++.

2 Abstract and Concrete Templates: Generalization

Most existing software has been designed with software components that are, in the Resolve/C++ taxonomy, abstract or concrete instances. Yet there are significant advantages to designing software components that are abstract and concrete templates. Templates serve two distinct purposes:

- to generalize components that otherwise would be too special-purpose for widespread use in different application programs; and
- to decouple dependencies between components.

This chapter and the next build upon Chapter 1 by showing you how to use these two slightly more advanced but very powerful software component engineering techniques. Consider, for example, the built-in type *Text*. A *Text* object is modeled by a mathematical string of characters. With the kernel operations, you can add, remove, or access a character by its position in the string, and you can get the length of the string. But what's so special about characters? Why can't you have objects modeled by strings of, say, booleans or integers or reals or anything else, and do *Text*-like operations on those objects? You can—and luckily, it's easy. This chapter shows you how to do it.

2.1 Parameterized Software Components

Mathematical string theory is a *generic*, or *parameterized*, theory: “stringness” is independent of what kind of entries are in a string. Some other mathematical theories are similar. You can have a mathematical set of any type of entry, a mathematical function from any type to any type, tuples of any types, and so on. Objects whose values are modeled as, say, mathematical strings of booleans are perfectly reasonable. How would you define a new programming object type with this model? Following the approach in Chapter 1, you would define a new abstract instance providing a type called, say, *Sequence_Of_Boolean* whose model would be a string of booleans, with essentially the same kernel operations as *Text*. Here's how you might specify it:

```
abstract_instance
class Sequence_Of_Boolean
{
public:

    standard_abstract_operations (Sequence_Of_Boolean);
    /*!
        Sequence_Of_Boolean is modeled by string of boolean
        initialization ensures
            self = empty_string
    !*/

    ...

    procedure Remove (
        preserves Integer pos,
        produces Boolean& x
    ) is_abstract;
    /*!
        requires
            0 <= pos < |self|
        ensures
```

```

        there exists a, b: string of boolean
        (|a| = pos and
         #self = a * <x> * b and
         self = a * b)
    !*/

    ...
};

```

You could specify the other operations in a similar way, by copying the interface contract specifications almost directly from the operations of *Text*.

Then you could do the same thing for a new abstract instance defining a type called, say, *Sequence_Of_Integer* whose model would be a string of integers. *Sequence_Of_Integer* would have the same kernel operations as *Sequence_Of_Boolean* except that the entries in an object's string model would be of type *Integer*.

Now if you're paying attention, you'll see it would be silly to key in *Sequence_Of_Integer* from scratch. Instead, you'd just copy the file for *Sequence_Of_Boolean* and replace all occurrences of "Boolean" by "Integer" using a text editor. The places to change are double-underlined below (shown after the substitutions have been made):

```

abstract_instance
class Sequence_Of Integer
{
public:

    standard_abstract_operations (Sequence_Of Integer);
    /*!
       Sequence_Of Integer is modeled by string of integer
       initialization ensures
           self = empty_string
    !*/

    ...

    procedure Remove (
        preserves Integer pos,
        produces Integer& x
    ) is_abstract;
    /*!
       requires
           0 <= pos < |self|
       ensures
           there exists a, b: string of integer
           (|a| = pos and
            #self = a * <x> * b and
            self = a * b)
    !*/

    ...
};

```

Copying and pasting code like this is generally bad, for three reasons:

- It is tedious.

- It is easy to make a mistake with the editor and accidentally mess up parts of the code that you didn't intend to change—even parts that you didn't think you touched. Taking a text editor to code is a lot like taking a blowtorch to a steel I-beam: once you've modified it, all the good properties you've established about it (e.g., the correctness of the code or the strength of the I-beam) are suddenly in doubt and have to be established again.
- Even if you do the modifications correctly, the result is that you have multiple copies of essentially the same code. If you need to change the common code later for any reason, you have to make these changes in multiple places. Software engineers know from experience that it is far better to have a *single point of control over change*, a single place where each non-trivial piece of code resides. This way, if you have to change it, you know the change needs to occur only there. In fact, if there is only one fundamental principle of software engineering that you must know, this is the one!

Fortunately, you don't have to do all this manual copying and pasting. Software components can have the same flexibility as mathematical theories in the sense of being parameterized. A parameterized software component is called a *template*. By using a template instead of an editor to generalize from *Text* to *Sequence*, you avoid the three problems noted above.

2.1.1 Resolve/C++ Language Support for Templates

In Resolve/C++, you declare one formal (template) parameter for each different type name that needs to be replaced; hence the term “parameterized component”. This formal parameter is analogous to a formal parameter to an operation. That is, a formal parameter is an arbitrary name that serves as a placeholder for information to be filled in later. The main differences between operation parameters and template parameters are summarized in Figure 2-1.

	Parameter to an operation	Parameter to a template
What the formal parameter stands for	A value of the (fixed) type of the parameter	A type
When the actual parameter value is supplied	Execution time	Compile time

Figure 2-1: Operation parameters vs. template parameters

In Resolve/C++ syntax, here's how you might use a template parameter to write an *abstract template* that serves as an interface contract specification for the generalized *Sequence_Kernel* (Section 2.3 gives the details):

```

abstract_template <
    concrete_instance class Item
>
class Sequence_Kernel
{
public:

    standard_abstract_operations (Sequence_Kernel);
    /*!
        Sequence_Kernel is modeled by string of Item
        initialization ensures
            self = empty_string
    */
}

```

```

    !*/

    ...

    procedure Remove (
        preserves Integer pos,
        produces Item& x
    ) is_abstract;
    /*!
        requires
            0 <= pos < |self|
        ensures
            there exists a, b: string of Item
                (|a| = pos and
                 #self = a * <x> * b and
                 self = a * b)
    !*/

    ...
};

```

The list of formal template parameters appears within angle-bracket delimiters <...> wedged between the keywords **abstract_template** and **class**. In this, case there is only one formal parameter, declared here as **concrete_instance class** *Item*, which denotes the type of the entries in the objects being specified. Other template components might have multiple template parameters.

2.1.2 Collection Components

In the *Sequence* family of components, we parameterize each component by the type of entries in the string to generalize code that otherwise would be far more special-purpose. This use of templates is therefore called **generalization**. By far the most common scenario of this kind arises when designing components whose mathematical models involve parameterized mathematical theories such as those mentioned at the beginning of this chapter: strings, sets, functions, and tuples. Such components usually are called **collection components** because objects of their types are collections of objects of some other type(s), which is/are the template parameter(s).

The appendices document several families of collection components. Among these are the *List*, *Queue*, *Sequence*, and *Stack* families (whose models involve mathematical strings); the *Array*, *Partial_Map*, *Set*, and *Static_Array* families (whose models involve mathematical sets); and the *Binary_Tree* family (whose model involves mathematical trees).

2.1.3 Template Instantiation

Suppose you are a client programmer writing an application program and you want the formal parameter *Item* in *Sequence_Kernel* to be replaced by, say, *Boolean*, creating an instance *Sequence_Of_Boolean* that is logically identical to the first piece of code in Section 2.1. When you ask the compiler to **instantiate** the template by providing this actual template parameter to bind to the formal template parameter, the C++ compiler automatically replaces all occurrences of *Item* by *Boolean* and treats the resulting code as an abstract instance. Instantiation is tantamount to “filling in the blanks” in a form. The compiler already knows how to compile the resulting abstract instance, and the net effect is exactly the same as if the original abstract instance *Sequence_Of_Boolean* had been used.

As a client programmer, however, you never explicitly instantiate an abstract template. The reason is that an application program needs to use concrete instances (which contain code that can be executed), not abstract instances (which contain interface contract specifications for code that can be executed). Parameterized concrete components—**concrete templates**—are entirely parallel to abstract templates in structure and purpose. The difference is that they are concrete, i.e., they consist of code that describes not what behavior is available but how that behavior is achieved. Once instantiated, this code can be compiled and executed. As with concrete instances for kernel components, a client programmer need not see any of the code in the concrete templates for kernel components being used, but rather only the abstract templates that specify their interface contracts. The next section illustrates how to develop an application program that uses a concrete template.

2.2 Example: Polynomial Evaluation

A problem that arises in numerical software is evaluation of a polynomial. For instance, suppose you have a polynomial:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

that you need to evaluate at a point x . How can you do this? The obvious method is to repeatedly multiply x by itself to compute x^n and multiply that by a_n to give the first term; then to compute x^{n-1} and multiply it by a_{n-1} to give the second term, and add it to the first term; and so on.

Horner's rule is a more efficient way to compute the same thing. If you rewrite the polynomial as follows, this alternative computation almost pops out at you:

$$p(x) = (\dots((a_n x + a_{n-1})x + a_{n-2})x + \dots + a_1)x + a_0$$

That is, start with a_n ; multiply it by x and add a_{n-1} ; then multiply that by x and add a_{n-2} ; and so on. With a little thought you can deduce that the original approach to the computation uses about $n^2/2$ multiplications and n additions, where Horner's rule uses only n multiplications and n additions. The problem we wish to explore is how much execution speed difference there is between these two methods, and also between a recursive and an iterative implementation of Horner's rule.

The user's manual for an application program to answer these questions might be:

Run the program by typing the command "Poly_Eval". It reads from *stdin* a non-negative integer which is the degree of the polynomial to be evaluated (i.e., n), followed by the $n+1$ real-valued coefficients in the order a_0 through a_n , followed by the real-valued point at which the polynomial is to be evaluated (i.e., x). Each number is on its own line of input. The program then prints out the polynomial followed by the evaluation of the polynomial at the point x and the execution time, for the obvious method and for Horner's rule (both iterative and recursive), and then quits.

The program that follows illustrates a solution, along with many other things.

```
// /*-----*\
// |   Main Program: Polynomial evaluation timing comparison
// | *-----*|
// |   Date:           9 August 1999 (revised 24 November 2006)
// |   Author:         Bruce W. Weide
// |
// |   Brief User's Manual:
```

```

// | Reads from stdin a non-negative integer which is the
// | degree of the polynomial to be evaluated (i.e., n),
// | followed by the n+1 real-valued coefficients in the
// | order a0 through an, followed by the real-valued point
// | at which the polynomial is to be evaluated (i.e., x).
// | Each number is expected to be on its own line of input.
// | The program then prints out the polynomial followed by
// | the evaluation of the polynomial at the point x and the
// | execution time, for the obvious method and for Horner's
// | rule (both iterative and recursive), and then quits.
// |
// \*-----*/

///-----
/// Global Context -----
///-----

#include "RESOLVE_Foundation.h"

#include "CT/Sequence/Kernel_1a_C.h"
#include "CI/Timer/1.h"

//-----

concrete_instance
class Polynomial_Coefficients :
    instantiates
        Sequence_Kernel_1a_C <Real>
{};

///-----
/// Interface -----
///-----

/*!
    math definition INPUT_ENCODING (
        s: string of real
    ): string of character satisfies
    if s = empty_string
    then INPUT_ENCODING (s) = empty_string
    else there exists y: real, r: string of real
        (s = <y> * r and
        INPUT_ENCODING (s) = TO_TEXT (y) * "\n" *
        INPUT_ENCODING (r))

    math definition EVAL (
        s: string of real,
        x: real
    ): real satisfies
    if s = empty_string
    then EVAL (s, x) = 0.0
    else there exists y: real, r: string of real
        (s = <y> * r and
        EVAL (s, x) = x * EVAL (r, x) + y)

!*/

```

```

//-----
//-----

global_procedure Read_Polynomial_And_X (
    alters Character_IStream& input,
    produces Polynomial_Coefficients& a,
    produces Real& x
);
/*!
    requires
        input.is_open = TRUE and
        there exists s: string of real, z: real
            (TO_TEXT (|s|) * "\n" * INPUT_ENCODING (s) *
              TO_TEXT (z) * "\n"
              is prefix of input.content)
    ensures
        input.is_open = TRUE and
        input.ext_name = #input.ext_name and
        #input.content = TO_TEXT (|a|) * "\n" *
                        INPUT_ENCODING (a) *
                        TO_TEXT (x) * "\n" *
                        input.content
!*/

//-----

global_procedure Write_Polynomial (
    alters Character_OStream& output,
    preserves Polynomial_Coefficients& a
);
/*!
    requires
        output.is_open = TRUE
    ensures
        output.is_open = TRUE and
        output.ext_name = #output.ext_name and
        output.content = #output.content * [display of
                                polynomial with coefficients
                                that are in a]
!*/

//-----

global_function Real Original_Evaluation (
    preserves Polynomial_Coefficients& a,
    preserves Real x
);
/*!
    ensures
        Original_Evaluation = EVAL (a, x)
!*/

//-----

global_function Real Horner_Iterative_Evaluation (
    preserves Polynomial_Coefficients& a,
    preserves Real x

```

```

    );
    /*!
        ensures
            Horner_Iterative_Evaluation = EVAL (a, x)
    !*/

    //-----

global_function Real Horner_Recursive_Evaluation (
    preserves Polynomial_Coefficients& a,
    preserves Real x
    );
    /*!
        ensures
            Horner_Recursive_Evaluation = EVAL (a, x)
    !*/

    //-----
    //-----

global_procedure_body Read_Polynomial_And_X (
    alters Character_IStream& input,
    produces Polynomial_Coefficients& a,
    produces Real& x
    )
{
    object Integer n, k;

    input >> n;
    a.Clear ();
    while (k <= n)
    {
        object Real coeff;

        input >> coeff;
        a.Add (k, coeff);
        k++;
    }
    input >> x;
}

//-----

global_procedure_body Write_Polynomial (
    alters Character_OStream& output,
    preserves Polynomial_Coefficients& a
    )
{
    object Integer k;

    output << "p(x) = \n";
    k = a.Length () - 1;
    while (k > 1)
    {
        output << " " << a[k] << " * x^(" << k << ") + \n";
        k--;
    }
}

```

```

    }
    if (k == 1)
    {
        output << "  " << a[1] << " * x + \n";
    }
    output << "  " << a[0] << "\n";
}

//-----

global_function_body Real Original_Evaluation (
    preserves Polynomial_Coefficients& a,
    preserves Real x
)
{
    object Real result;
    object Integer k;

    while (k <= a.Length () - 1)
    /*!
        preserves a, x
        alters result, k
        maintains
            there exists s: string of real
                (|s| = k and
                 s is prefix of a and
                 result = EVAL (s, x))
        decreases
            |a| - k
    !*/
    {
        object Real term = a[k];
        object Integer i;

        // Compute the k-th term = a[k] * x ^ (k)

        while (i < k)
        /*!
            preserves x, k
            alters term, i
            maintains
                i <= k and
                term = #term * x ^ (i)
            decreases
                k - i
        !*/
        {
            term *= x;
            i++;
        }

        // Add this term to total so far

        result += term;
        k++;
    }
}

```

```

    return result;
}

//-----

global_function_body Real Horner_Iterative_Evaluation (
    preserves Polynomial_Coefficients& a,
    preserves Real x
)
{
    object Real result;
    object Integer k = a.Length () - 1;

    while (k >= 0)
    /*!
        preserves a, x
        alters result, k
        maintains
            there exists s: string of real
                (|s| = |a| - (k + 1) and
                 s is suffix of a and
                 result = EVAL (s, x))
        decreases
            k + 1
    !*/
    {
        result = result * x + a[k];
        k--;
    }

    return result;
}

//-----

global_function_body Real Horner_Recursive_Evaluation (
    preserves Polynomial_Coefficients& a,
    preserves Real x
)
{
    object Real result;

    if (a.Length () == 0)
    {
        return 0.0;
    }
    else
    {
        object Real y;

        a.Remove (0, y);
        result = x * Horner_Recursive_Evaluation (a, x) + y;
        a.Add (0, y);
        return result;
    }
}

```



```

//-----
//-----

program_body main ()
{
    object Character_IStream input;
    object Character_OStream output;
    object Polynomial_Coefficients a;
    object Real x, evaluation;
    object Integer k, elapsed_time;
    object Timer_1 t;

    // Open input and output streams

    input.Open_External ("");
    output.Open_External ("");

    // Read number of coefficients, coefficients a, and point x

    Read_Polynomial_And_X (input, a, x);

    // Print polynomial, evaluate it, and report timing for each
    // evaluation method

    // ---- Original method ----

    output << "Original method:\n";
    Write_Polynomial (output, a);
    k = 0;
    t.Restart ();
    while (k < 1000)
    {
        evaluation = Original_Evaluation (a, x);
        k++;
    }
    elapsed_time = t.Reading ();
    output << "p(" << x << ") = " << evaluation << "\n"
        << "execution time = " << elapsed_time
        << " usec\n\n";

    // ---- Horner's rule, done iteratively ----

    output << "Horner's rule (iterative):\n";
    Write_Polynomial (output, a);
    k = 0;
    t.Restart ();
    while (k < 1000)
    {
        evaluation = Horner_Iterative_Evaluation (a, x);
        k++;
    }
    elapsed_time = t.Reading ();
    output << "p(" << x << ") = " << evaluation << "\n"
        << "execution time = " << elapsed_time
        << " usec\n\n";
}

```

```

// ---- Horner's rule, done recursively ----

output << "Horner's rule (recursive):\n";
Write_Polynomial (output, a);
k = 0;
t.Restart ();
while (k < 1000)
{
    evaluation = Horner_Recursive_Evaluation (a, x);
    k++;
}
elapsed_time = t.Reading ();
output << "p(" << x << ") = " << evaluation << "\n"
    << "execution time = " << elapsed_time
    << " usec\n\n";

// Close input and output streams

input.Close_External ();
output.Close_External ();
}

```

Figure 2-2: Poly_Eval.cpp

This program introduces only one substantially new feature of Resolve/C++: the use of a concrete template, *Sequence_Kernel_1a_C*. Like the abstract template it implements, a concrete template has formal parameters that must be bound to actual parameters in a client program uses it. There can be many alternative concrete templates that might implement a given abstract template. After deciding which of these concrete templates to use in a given situation—a choice that depends on matching the performance (time and space) characteristics of the candidate implementations with the application's performance requirements—the client programmer needs to do two things in order to use the selected concrete template:

- Bring the desired concrete template into scope in the application program.
- Instantiate it to create a concrete instance.

2.2.1 Bringing a Concrete Template Into Scope

Let's continue with the Poly_Eval example to see exactly how this works. The appendix for the *Sequence* family lists just one implementation of *Sequence_Kernel*, which is called *Sequence_Kernel_1a_C*; so there isn't much choice here.¹ The appendix explains:

To bring this component into scope you write:

```
#include "CT/Sequence/Kernel_1a_C.h"
```

Hence, you see this exact statement in the global context section near the beginning of Figure 2-2.

¹ In fact, there are other implementations of *Sequence_Kernel* in the Resolve/C++ Catalog, but we ignore them here and in the appendix in order to concentrate on how to use a chosen concrete template.

The other component brought into scope at this point in the code is a *Timer* component (a concrete instance) that lets you time the execution of any code fragment. Its use is illustrated by this program but is not further discussed here.

The name “CT/Sequence/Kernel_1a_C.h” derives from the following features, as introduced in Chapter 1:

- “CT/” means this component is a concrete template;
- “Sequence/” means it is in the *Sequence* component family;
- “Kernel_1a_C” means it is implementation “1a_C” of *Sequence_Kernel* (remember there might be others, and they have different names).
- “.h” means the C++ code is not compiled in advance, but rather will be compiled with the application program that is including it.

As with a concrete instance, bringing a concrete template into scope automatically brings into scope the abstract template (in this case, *Sequence_Kernel*) that specifies the interface contract it implements.

2.2.2 Instantiating the Concrete Template to Create a Concrete Instance

After a concrete template component is in scope, you need to instantiate it in order to use it. This step is the only extra one required when using templates as opposed to instances, and fortunately it is pretty easy. The simple declaration that instantiates *Sequence_Kernel_1a_C* in *Poly_Eval.cpp* comes immediately after all the components to be used are brought into scope. It uses the keywords **concrete_instance class** and **instantiates**² as follows:

```
concrete_instance
class Polynomial_Coefficients :
    instantiates
        Sequence_Kernel_1a_C <Real>
{};
```

In *Poly_Eval.cpp*, the application programmer has decided that (mathematically speaking) the coefficients of a polynomial can be treated as a string of real numbers. He/she has consulted the interface contract specification *Sequence_Kernel* and has noted that the mathematical model of a *Sequence_Kernel_1a_C* object is a string, the type of entries in the string being determined by a single template parameter. Therefore, it is appropriate to bind the formal parameter *Item* of *Sequence_Kernel_1a_C* to the type *Real*. The resulting concrete instance is named something that suggests its intended use in this application program: *Polynomial_Coefficients*. You don’t have to name an instance of *Sequence_Kernel_1a_C* something general like *Sequence_Of_Real*.

2.2.3 Using the Resulting Concrete Instance

Once an application program instantiates the concrete templates it needs, it uses the resulting concrete instances just as though they were concrete instances from a component catalog, i.e., just as though they were built-in types. *Poly_Eval.cpp* involves no other new features of *Resolve/C++*. Nonetheless, it is worth studying to see how it uses previously-discussed features.

² In C/C++ programs, you would use the keywords **class** and **virtual public** in place of the *Resolve/C++* keywords **concrete_instance class** and **instantiates**, respectively.

For example, the interface section begins with a mathematical definition that is used (in the specification of *Read_Polynomial_And_X*) to describe precisely the input format for the program. The other mathematical definition is used (in the specifications of the three polynomial evaluation operations) to describe precisely what “polynomial evaluation” means. Both these definitions are inductive.

In contrast to the formal specifications for the input and computation operations, the specification of the operation *Write_Polynomial* is partly informal (see the part of the postcondition inside square-bracket delimiters [...]). It is unambiguous that the output stream must be open at the time of the call, and that it remains open and attached to the same “sink” as when the call was made. But the exact format of the output produced is not completely specified here. In principle, it could be formally specified just as the input format is formally specified.

Poly_Eval.cpp is the first sample program we’ve seen that illustrates how it is possible to specify the format of inputs and/or outputs. It also is the first to illustrate that the exhortation to use formal specifications is not meant to be taken as dogma, but as a practical technique for making your programs more understandable and more maintainable by others (and by you, in the future, after you’ve forgotten the details). Indeed there are two other places where the commitment to formality hasn’t been observed here: the loops in the input and output operations don’t have loop invariants. At least the input loop probably should have this, because the behavior of *Read_Polynomial_And_X* is formally specified.

By the way, we ran this code on one computer with the following input:

```
13
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
11.0
12.0
13.0
14.0
10.0
```

The program produced the following output:

```
Original method:
p(x) =
14.000000 * x^(13) +
13.000000 * x^(12) +
12.000000 * x^(11) +
11.000000 * x^(10) +
10.000000 * x^(9) +
9.000000 * x^(8) +
8.000000 * x^(7) +
7.000000 * x^(6) +
6.000000 * x^(5) +
5.000000 * x^(4) +
4.000000 * x^(3) +
```

```

3.000000 * x^(2) +
2.000000 * x +
1.000000
p(10.000000) = 154320987654321.000000
execution time = 350 usec

```

```

Horner's rule (iterative):
p(x) =
14.000000 * x^(13) +
13.000000 * x^(12) +
12.000000 * x^(11) +
11.000000 * x^(10) +
10.000000 * x^(9) +
9.000000 * x^(8) +
8.000000 * x^(7) +
7.000000 * x^(6) +
6.000000 * x^(5) +
5.000000 * x^(4) +
4.000000 * x^(3) +
3.000000 * x^(2) +
2.000000 * x +
1.000000
p(10.000000) = 154320987654321.000000
execution time = 220 usec

```

```

Horner's rule (recursive):
p(x) =
14.000000 * x^(13) +
13.000000 * x^(12) +
12.000000 * x^(11) +
11.000000 * x^(10) +
10.000000 * x^(9) +
9.000000 * x^(8) +
8.000000 * x^(7) +
7.000000 * x^(6) +
6.000000 * x^(5) +
5.000000 * x^(4) +
4.000000 * x^(3) +
3.000000 * x^(2) +
2.000000 * x +
1.000000
p(10.000000) = 154320987654321.000000
execution time = 230 usec

```

Here are a few simple exercises to test your understanding of the code, the problem it addresses, and this sample test run:

- In the body of *Read_Polynomial_And_X*, is there really a need to clear object *a* before starting the loop? How does the specification of the operation help you decide the answer to this question?
- In the body of *Write_Polynomial*, is there really a need for “**if** (*k* == 1)” before the output of the $a_k x$ term? How does the specification of the operation help you decide the answer to this question?

- Why does the program evaluate the polynomial 1000 times using each method? Is each reported execution time (given in “usec”, which is supposed to be “μsec”, i.e., microseconds) the time for one execution of the evaluation operation, or for all 1000 evaluations?
- The times reported include some time for incrementing k and for loop overhead. How would you determine whether this error is significant, and how would you adjust the times reported (if necessary)?
- Why does the program output the polynomial before using each evaluation method? (Rather, why does the requirement for the application state that it should do this?)
- On how many different polynomials would you test the program before you could state confidently under what conditions Horner’s rule is faster than the obvious method of evaluation, and by how much? Similarly, what test runs would enable you to assert confidently whether there is a performance penalty for using the recursive implementation of Horner’s rule, and if so how much of a penalty?

2.3 Anatomy of a Component Family: *Sequence*

There is one essential difference between the code for an abstract instance and that for an abstract template: the latter is parameterized, so there is a list of formal template parameters. Other than this, the basic structure of a component family is just as explained in Section 1.3.

2.3.1 The Kernel Type, Standard Operations, and Kernel Operations

Figure 2-3 shows the entire file contents for the abstract component called *Sequence_Kernel*. Some of this file is so cut-and-dried that all abstract template files that introduce new kernel types are identical in structure to this one.

```
// /*-----*\
// |   Abstract Template : Sequence_Kernel
// \|-----*/

#ifndef AT_SEQUENCE_KERNEL
#define AT_SEQUENCE_KERNEL 1

///-----
/// Interface -----
///-----

abstract_template <
    concrete_instance class Item
>
class Sequence_Kernel
{
public:

    standard_abstract_operations (Sequence_Kernel);
    /*!
        Sequence_Kernel is modeled by string of Item
        initialization ensures
            self = empty_string
    !*/

    procedure Add (
```

```

        preserves Integer pos,
        consumes Item& x
    ) is_abstract;
    /*!
    requires
        0 <= pos <= |self|
    ensures
        there exists a, b: string of Item
            (|a| = pos and
             #self = a * b and
             self = a * <#x> * b)
    !*/

    procedure Remove (
        preserves Integer pos,
        produces Item& x
    ) is_abstract;
    /*!
    requires
        0 <= pos < |self|
    ensures
        there exists a, b: string of Item
            (|a| = pos and
             #self = a * <x> * b and
             self = a * b)
    !*/

    function Item& operator [] (
        preserves Integer pos
    ) is_abstract;
    /*!
    requires
        0 <= pos < |self|
    ensures
        there exists a, b: string of Item
            (|a| = pos and
             #self = a * <self[pos]> * b)
    !*/

    function Integer Length () is_abstract;
    /*!
    ensures
        Length = |self|
    !*/

};

#endif // AT_SEQUENCE_KERNEL

```

Figure 2-3: AT/Sequence/Kernel.h

The fact that *Sequence_Kernel* is a template rather than an instance is evident, first, in the stylized comment at the beginning of this file. The keyword **abstract_template**³ tells the

³ In C/C++ programs, you would use the keyword **template** in place of the Resolve/C++ keyword **abstract_template**.

compiler to expect next the angle brackets <...> that surround the list of the formal template parameters; recall Section 2.1.1. Here, there is one template parameter: a concrete instance class called *Item*. A formal parameter name is used only within the template where it is declared. It has no special status, and may be given any name not already in scope.

As you can readily see, this interface contract specification includes the standard operations and the other kernel operations, just like the abstract instance *Random_Kernel* in Chapter 1.

2.3.2 The *Extends* Relation (Abstract-to-Abstract)

Chapter 1 discusses a second component in the *Random* family: an abstract instance that adds interface contract specifications for additional operations to extend *Random_Kernel*. An analogous approach applies to templates. Suppose you want to add to *Sequence_Kernel* an interface contract for an operation to reverse a sequence. Figure 2-4 shows how to do this.

```
// /*-----*\
// |   Abstract Template : Sequence_Reverse
// \*-----*/

#ifndef AT_SEQUENCE_REVERSE
#define AT_SEQUENCE_REVERSE 1

///-----
/// Global Context -----
///-----

#include "AT/Sequence/Kernel.h"

///-----
/// Interface -----
///-----

abstract_template <
    concrete_instance class Item
>
class Sequence_Reverse :
    extends
        abstract_instance Sequence_Kernel <Item>
{
public:

    procedure Reverse () is_abstract;
    /*!
        ensures
            self = reverse (#self)
    !*/

};

#endif // AT_SEQUENCE_REVERSE
```

Figure 2-4: AT/Sequence/Reverse.h

The formal template parameter of *Sequence_Reverse* happens to be called *Item*, as in *Sequence_Kernel*. However, these two templates are not connected by this name! The formal

parameter names in these two templates could be different from each other. The connection is made not by the parameter names but rather by this code:

```
abstract_template <
    concrete_instance class Item
>
class Sequence_Reverse :
    extends
        abstract_instance Sequence_Kernel <Item>
...

```

This declaration of *Sequence_Reverse* means that every instance of the *Sequence_Reverse* template extends some (corresponding) instance of the *Sequence_Kernel* template. In particular, the abstract instance *Sequence_Reverse*<*Item*> extends the abstract instance *Sequence_Kernel*<*Item*>, for any type that a client might choose to provide for *Item*. For example, *Sequence_Reverse*<*Integer*> extends *Sequence_Kernel*<*Integer*>, *Sequence_Reverse*<*Random_Uniform_1*> extends *Sequence_Kernel*<*Random_Uniform_1*>, and so on.

In the postcondition for program operation *Reverse*, the built-in mathematical function *reverse* denotes the string obtained by considering the entries in the string to be in the opposite order. For example, with strings of integers, the following is true:

```
reverse (<1, 2, 3>) = <3, 2, 1>
```

Figure 2-5 shows how to depict in a CCD the fact that the *extends* relation holds between these two abstract templates. The difference between this CCD and its counterpart in Chapter 1 is that the rounded-corner rectangles standing for the two abstract templates have heavy borders. This is a visual cue used in CCD's to indicate that components are templates rather than instances.

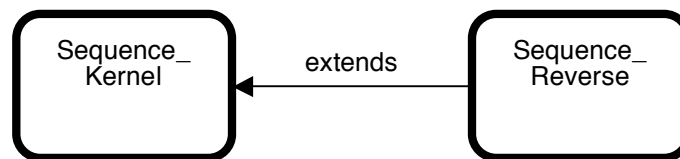


Figure 2-5: The extends relation may hold between two abstract templates

This CCD raises a second subtle issue. Both *Sequence_Kernel* and *Sequence_Reverse* have template parameters. If a component uses another component that is not built-in, then this relationship should be shown in the CCD. Why, then, is there nothing in the CCD to indicate that each of these components *uses* this other type called *Item*? Shouldn't there be another box in the diagram to stand for *Item*—because the type a client binds to *Item* might not be built-in—along with a *uses* arrow to it from each of the templates shown above?

No, and here's why. It is certainly true that the *uses* relation holds between any template and the components bound by the client to its template parameter(s). However, the purpose of a CCD is to show dependencies between components at **component design time**, i.e., as those components exist in a component catalog, out of the context of any particular client program in which they might be used. The bindings of particular values to template parameters arise at **instantiation time**, i.e., as components are integrated into a specific client program by template instantiation. At design time, we know literally nothing about which component *Item* (in this example) might be bound to later: it might be any type whatsoever. The template parameter *Item* therefore introduces no dependency between *Sequence_Kernel* or *Sequence_Reverse* and any other

component that is known at component design time. So, no box (for *Item*) or arrow (for *uses*) is depicted in the above CCD.

This conclusion is consistent with the intended interpretation of a CCD: in order to understand a component, you need to understand all—and only—the other components that can be reached from that component by following arrows out of it in the CCD. To understand *Sequence_Kernel*, for example, you don't need to understand any other component. To understand *Sequence_Reverse*, you need to understand *Sequence_Kernel*. Why? The specification of the mathematical model for sequence objects is there. The interface contract for the *Reverse* operation in Figure 2-4 makes no sense unless you know that *self* is a string of *Items*.

2.3.3 Using a Concrete Component

As a client of the off-the-shelf concrete component used in the Poly_Eval program, all you need to know about it is summarized in the CCD in Figure 2-6. The name of the concrete instance is *Sequence_Kernel_1a_C*, and it implements *Sequence_Kernel*. The interface contract specification in the latter component explains how *Sequence_Kernel_1a_C* objects will behave in the client program: the possible values those objects can have and the operations that you can call on them. That's it. As a client, you do not need to see the code for *Sequence_Kernel_1a_C*.

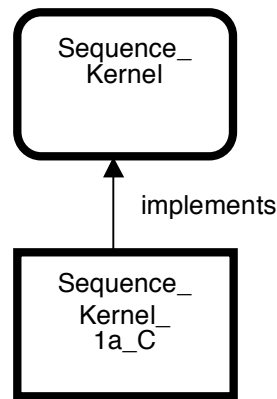


Figure 2-6: The implements relation may hold between a concrete and abstract template

2.4 Summary

Abstract and concrete templates are parameterized software components that you can instantiate to create abstract and concrete instances, respectively. One important use of templates is to generalize component designs. This is typically done with a collection component, i.e., one that introduces a user-defined type whose mathematical model involves strings, sets, functions, and/or tuples. In each of these mathematical theories, there is at least one type parameter: the type of entries for a string or set, the domain and range types for a function, the types of the fields of a tuple.

Templates in this case address the following problem. Suppose you have a component that dictates a specific type for a collection's entries, and you want to make it more general so it allows the collection's entries to be any type. If all you have are abstract and concrete instances, then there is an obvious way to do this: copy and paste code from the instance you have and make routine replacements in it with a text editor. Unfortunately, copying and pasting code has severe disadvantages. The most important is the loss of single point of control over change. If

you ever need to change a piece of code that has been copied and pasted, you have to find all the places where it has been propagated and change it there, too.

The advantage of using templates rather than copy/paste is that the template becomes that single point of control over change. Instances generated by instantiating the template (i.e., by fixing the template's parameters) are not created manually by a software engineer wielding a text editor but rather by the C++ compiler. Instantiation of a template by a client programmer involves a simple statement that looks similar to an operation call. However, fixing a template's parameters happens at compile-time (not execution-time, as with an operation call) and denotes a textual substitution that converts the template to an instance. An instance is something that the compiler already knows how to process.

3 Abstract and Concrete Templates: Decoupling

In addition to their most obvious use (for generalization), templates can help you design more elegant and reusable software components in another way. This chapter starts by considering the problems caused by component-to-component dependence, and considers why concrete-to-concrete component dependence in particular is a problem—both for human understandability and for maintainability of software. It then explains how to use templates to avoid concrete-to-concrete component dependence.

3.1 The Problem of Component Dependence

Component *X* **depends** on component *Y* if you need to know something about *Y* in order to understand *X*. When the code for component *X* mentions component *Y* by name, then *X* directly depends on *Y*. Notice that dependence is **transitive**, i.e., if *X* depends on *Y* and *Y* depends on *Z*, then *X* depends on *Z*. Every specific dependence relation between components discussed so far, i.e., *extends*, *implements*, and *uses*, is transitive; but even if a specific dependence relation were not transitive, the general dependence it introduces would be transitive.

Looking the other direction, if you change *Y* even slightly then you certainly have to understand the impact of this change on *X*, and you might have to change *X*, too, and then anything that depends on *X*. So more dependencies reduce not only the understandability but the maintainability (ease of change) of software components.

3.1.1 Avoidable Dependence Is Bad

Some dependencies are simply unavoidable. For example, consider the example from Chapter 1 whose CCD is reproduced in Figure 3-1. There is just no way to explain the operation *Uniform_Real* (which is in the component *Random_Uniform*) without talking about the mathematical model of *self* (which is in the component *Random_Kernel*). Of course, you could try to eliminate the dependence *between* the two abstract components by combining the dependent specifications *within* a single abstract component. But this wouldn't solve the problem of limiting what you need to know and understand. It would merely make internal—and therefore not apparent from a CCD—the inherent dependence of the specification of each operation on the kernel type's mathematical model. Since CCDs are intended to show dependencies, this attempt to mask the problem would be self-defeating.

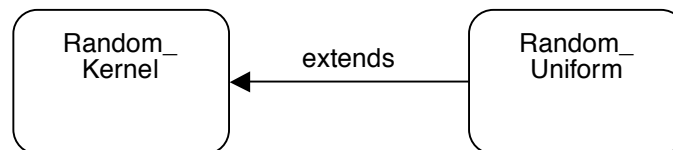


Figure 3-1: Unavoidable dependence between abstract instances

Figure 3-2 illustrates the kind of situation you really want to avoid: a long **dependence chain**, or a long path of arrows in a CCD. Remember, in order to understand a component you need to understand all the components it depends on, either directly or (by transitivity) indirectly. These are all the components that are reachable from a component by following arrows out of it in the CCD. Similarly, if you change a component, you might have to change any or all of the components from which the changed component is reachable along a dependence chain.

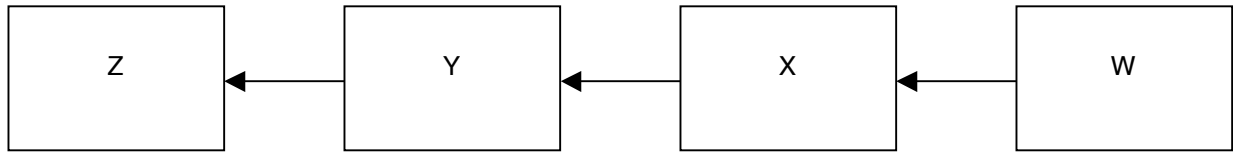


Figure 3-2: A dependence chain

The intuitive objective that you should minimize such dependence chains suggests a general rule-of-thumb for component design:

Limited Dependence Rule: Design each component so it depends on as few other components as possible.

Here is an important general consequence of the limited dependence rule that says something about how to achieve it, i.e., about how to partition information among components:

Least Information Rule: Design each component so it depends only on components that contain information that is required to understand the new component.

This design rule is the basis for distinguishing abstract and concrete components: a client program needs to know only the interface contracts for the components it uses, not the details of their implementations. In fact, as we will see next, the division of component information into “what” vs. “how” is arguably the most important idea in component-based software.

3.1.2 Avoidable Concrete-to-Concrete Dependence Is Worse

The ill effects of long dependence chains arise whether the components involved are abstract components or concrete components. But the effects of dependence chains are more serious when they involve concrete components because—in the absence of abstract components and the knowledge of how to take advantage of them—most programmers routinely introduce concrete-to-concrete dependencies. Without disciplined design, a concrete component depends on another concrete component, and that depends on yet another concrete component, that on yet another, and so on: you have a long dependence chain.

In fact, it gets worse. If you don’t consciously limit dependencies between concrete components, then a typical concrete component depends on more than one other component, so the dependencies “branch out”. Every time there is a dependence from a concrete component to two others, there is a doubling of the number of dependence chains. The resulting chains together form a *dependence graph*. A component at the root of this graph depends on all the components in all the dependence chains in the entire graph—and the number of chains in the graph typically grows very quickly with the lengths of the chains because of the typical branching factor for a concrete component. Figure 3-3 illustrates how a single concrete component like the one on the right can depend (directly and indirectly) on a very large number of other components. Imagine that the same kinds of dependencies continue for another few “levels” of arrows and you can see that trouble is brewing. Something like Figure 3-2 appears as a single (highlighted) dependence chain in Figure 3-3—but as you can see, it’s only the tip of the iceberg.

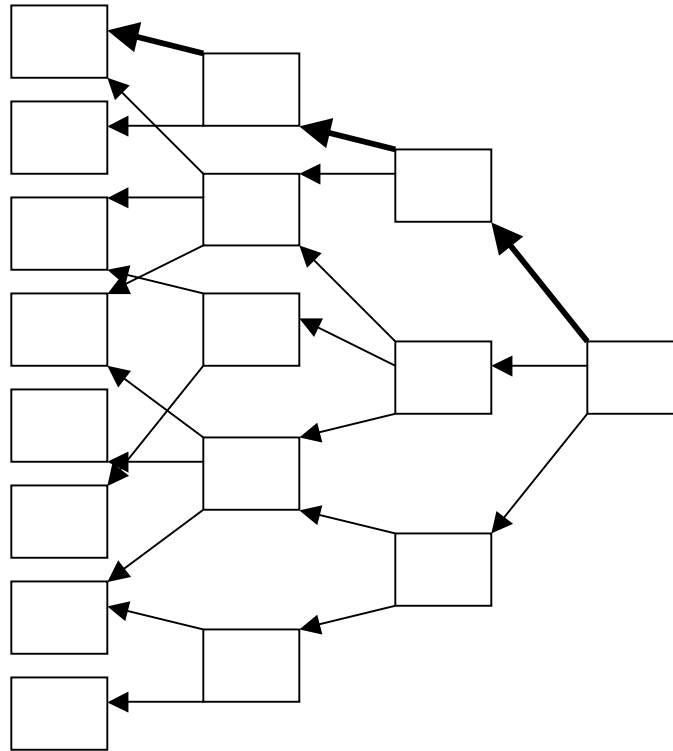


Figure 3-3: Dependence chains are part of dependence graphs

How can you avoid this kind of situation? A design-by-contract discipline such as Resolve/C++ makes it possible for a component designer to have each concrete component depend only on the interface contracts needed to build it, and not on particular concrete components that implement those contracts. Where there is a dependence on an abstract component, in general this dependence is to a kernel abstract component or to an extension of it. This dependence chain from the concrete component can, therefore, go through perhaps two other components; there, it (usually) stops. This means you can think of a dependence on an abstract component as “killing” a dependence chain—or at least, as declaring it terminally ill.

This observation leads to another specific consequence of the limited dependence rule:

Concrete Component Dependence Rule: Avoid introducing a dependence on a concrete component. If the dependence of a concrete component on another component is required, create that dependence to an abstract component.

It is easy to avoid dependencies from abstract components to concrete components because it is never necessary to make what something does depend on how something else achieves its behavior. The problem is in how to avoid dependence of a concrete component on other concrete components. Specifically, the *checks* and (concrete-to-concrete) *extends* relations are defined later in this chapter as concrete-to-concrete dependencies. The remaining sections of this chapter explain these dependence relationships, and then describe how to use templates to *decouple*, or break, the dependencies in what otherwise could become very long dependence chains among concrete components. The result of carefully applying these rules is that there are only two categories of dependencies in well-engineered software component catalogs: abstract-to-abstract dependencies and concrete-to-abstract dependencies. There are no long dependence chains or large dependence graphs.

The design rules above inform the recommended solutions to several problems that arise in software component engineering—even ones that, on the surface, do not seem directly related to dependencies. Let’s look at two of them now, along with their template-based solutions.

3.2 Decoupling the Checks Relation

Traditionally, most people have agreed that one of the key features of good software is that it is “bulletproof”. A proper interpretation of this advice is that the end user of an application program should not be able to do anything that causes it to crash. If you’ve studied them carefully (as you should), then you might notice that some of the application program examples in this book seem to be bulletproof. For example, consider the Euclid program discussed in Volume 1, and reproduced here for convenience.

```
// /*-----*\
// |   Main Program: Euclid's algorithm for GCD
// |*-----*|
// |   Date:           13 August 1996 (revised 24 November 2006)
// |   Author:         Bruce W. Weide
// |
// |   Brief User's Manual:
// |   Asks for two integers, and if both are non-negative
// |   and one is positive, reports the GCD of the two, and
// |   then quits.
// |
// |*-----*/

///-----
/// Global Context -----
///-----

#include "RESOLVE_Foundation.h"

///-----
/// Interface -----
///-----

/*!
  math definition DIVIDES (
    d: integer,
    n: integer
  ): boolean is
    there exists k: integer (n = k * d)

  math definition IS_GCD (
    gcd: integer,
    m: integer,
    n: integer
  ): boolean is
    (m /= 0 or n /= 0) and
    DIVIDES (gcd, m) and
    DIVIDES (gcd, n) and
    for all k: integer where (DIVIDES (k, m) and DIVIDES (k, n))
      (k <= gcd)
!*/
```



```

//-----
global_function Integer Greatest_Common_Divisor (
    preserves Integer j,
    preserves Integer k
);
/*!
    requires
        0 <= j <= k  and
        0 < k
    ensures
        IS_GCD (Greatest_Common_Divisor, j, k)
!*/
//-----

global_function_body Integer Greatest_Common_Divisor (
    preserves Integer j,
    preserves Integer k
)
{
    if (j == 0)
    {
        return k;
    }
    else
    {
        return Greatest_Common_Divisor (k mod j,  j);
    }
}
//-----

program_body main ()
{
    object Character_IStream input;
    object Character_OStream output;
    object Integer small, large;

    // Open input and output streams

    input.Open_External ("");
    output.Open_External ("");

    // Get two integers from user; put them in proper order

    output << "Please input an integer: ";
    input >> small;
    output << "Please input another integer: ";
    input >> large;
    if (small > large)
    {
        small &= large;
    }

    // Compute GCD of small and large

```

```

if ((small < 0) or (large <= 0)) // Can't find GCD
{
    output << "\nSorry, both integers must be non-negative, "
        << "and one must be positive";
}
else // Compute and report GCD
{
    object Integer gcd;

    gcd = Greatest_Common_Divisor (small, large);
    output << "\nGCD = " << gcd << "\n";
}

// Close input and output streams

input.Close_External ();
output.Close_External ();
}

```

Figure 3-4: Euclid.cpp

This program asks the end user to input two integers. It expects both numbers to be non-negative and one to be positive, and if this isn't the case then the program gently reports the error:

Sorry, both integers must be non-negative, and one must be positive.

On the other hand, watch what happens if you respond to the program's first prompt for an integer by typing:

an integer?

The program squawks (something like):

```

=====

0: main
1: operator>>(Character_IStream&, Integer&)
2: Integer::Get_From(Character_IStream&)
3: To_Integer(Text)

=====

Violated assertion at:

File: /class/sce/rcpp/RESOLVE_Foundation/Conversion_Operations/-
Conversion_Operations.cpp
Line #: 552
Operation: To_Integer

Violated assertion is:
=====
<OK_Text>      ::= <White_Space> ['+'|'-'] <Digits> <White_Space>
<White_Space> ::= {' ', '\t', '\n'}*
<Digits>       ::= {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}+
-----
t is in language of OK_Text and

```

```
-2147483648 <= TO_INTEGER(t) <= 2147483647
=====

=====

Abort
```

The relatively cryptic “violated assertion” message comes from the operation *To_Integer*, which is invoked by the operation *Get_From* for the built-in type *Integer*, which is invoked by the `>>` operator in the main program of Euclid. The *Get_From* operation for *Integer*, it turns out, has a non-trivial precondition: the next line of characters to be read from the source of characters that is attached to the *Character_IStream* object *input* at this point—normally the keyboard—must be interpretable as an *Integer* value. Specifically, the next line must begin with zero or more whitespace characters followed by at least one decimal digit character followed by zero or more whitespace characters. The characters you typed in response to the prompt, i.e., “an integer?”, do not meet this condition and cannot be interpreted as an *Integer* value.

The second error message is a lot less end-user friendly than the first one. But at least you know something went haywire at the point you typed a response to the prompt. Agreed, it would be nicer for the program to inform you:

```
Sorry, you have not typed an integer. Please try again.
```

It is possible to change the program to do this: rather than reading into an *Integer* object, you could read the user’s response into a *Text* object (which is always legal), then check whether it can be converted to an *Integer*. But there are plenty of even less-friendly things the program could do! For instance, it could forge blithely ahead and keep on computing, apparently happy as a clam, only to surprise you later with a bizarre result or a terrible death-by-segmentation-fault or death-by-bus-error¹ for which the root cause would be much harder to find.

3.2.1 The Defensiveness Dilemma

You might be tempted to conclude that, because programs should be bulletproof for end users, the same maxim applies to a software component’s operations whose “users” are client programs that invoke the services of that component. That is, you might expect that every operation with a non-trivial precondition—like *Get_From* for *Integer*—should always check that precondition to make sure the client program does not crash. In fact, if you jumped to this conclusion then you would be in good company. Many programming texts advocate this. But the Euclid program violates this advice. For example, its *Greatest_Common_Divisor* operation has a non-trivial precondition, yet the implementation of the operation does not check that this condition holds.

What are the differences among the various situations in Euclid? Why does one kind of error lead to an easy-to-understand application-specific error message while the other leads to a cryptic general-purpose (but still arguably helpful, to the programmer) error message? Why do some operation implementations check their preconditions while others do not?

The *defensiveness dilemma* is an important issue faced by an operation’s implementer, including a client programmer who needs to implement a new operation:

To check or not to check preconditions, that is the question.

¹ These are two of the tragedies that can befall a defective program on one system. The catastrophes on your system may vary.

The term “defensiveness” comes from the traditional, party line advice—always defend against client errors in calling your operation by checking its precondition in the operation’s implementation. A closer analysis of the situation reveals why there seems to be a “dilemma”, and why the party line is not the right approach.

For any particular call of any particular operation, there are four possible answers to the question of which person—client programmer or operation implementer—is responsible for establishing that the precondition of the operation holds at the time of the call:

1. Neither is responsible.
2. Both are responsible.
3. Operation implementer is responsible.
4. Client programmer is responsible.

It is easy to dispense with answers 1 and 2. If an operation has a non-trivial precondition that is not satisfied at the time of the call, then by definition of the interface contract, the operation is permitted to do *anything*. If no one is responsible for seeing to it that this condition holds, then the operation can, among other things, legitimately:

- Do nothing more, but quietly return to the calling program without doing anything else.
- Continue computing with “garbage” data, and produce any answers it likes.
- Halt the program without further report or further incident.
- Emit an error message.
- Fire a missile toward Michigan.

As the caller of the operation you cannot predict which, if any, of these or many other possibilities will ensue. So the only reason a rational client programmer would not worry about whether the precondition holds is that he/she thinks the operation implementer has written code to check it and to do something tame in response to a violation. And the only reason a rational operation implementer would not check the precondition is that he/she thinks the client programmer will check it. If there is no policy that establishes who is responsible then trouble is inevitable. Answer 1 is, therefore, not a good policy.

So consider answer 2: both parties are responsible for writing code that checks the precondition of the call. Now there is no problem in principle, but there is a problem in practice because this approach is needlessly inefficient. Why do the same computation twice when once is always enough, if only there were some policy to determine who is responsible for performing it?

This leaves two apparently reasonable choices. To embrace answer 3 is to adopt the party line and make the operation implementer responsible. There are two problems with this approach. The first problem is that a check inside the operation body is often redundant, even when the client programmer does not “consciously” check the precondition before making the call. Consider, for example, the operation *Greatest_Common_Divisor* in the Euclid example:

```
global_function Integer Greatest_Common_Divisor (
    preserves Integer j,
    preserves Integer k
);
/*!
    requires
        0 <= j <= k and
```

```

    0 < k
  ensures
    IS_GCD (Greatest_Common_Divisor, j, k)
!*/

global_function_body Integer Greatest_Common_Divisor (
    preserves Integer j,
    preserves Integer k
)
{
    if (j == 0)
    {
        return k;
    }
    else
    {
        return Greatest_Common_Divisor (k mod j, j);
    }
}

```

The (recursive) call in the body is never made if $j = 0$, since it occurs in the **else** block of an **if** statement whose condition is “ $j == 0$ ”. So the calling program does check this part of the precondition—not because it’s “afraid” to call *Greatest_Common_Divisor* in that case, but because it doesn’t have to; this is the base case in the recursion.

The subtle difference in intent for the above case might seem pedantic. So consider the other part of the *requires* clause, which says (in part) that the first argument to the operation is no larger than the second. There is no explicit check for this before the recursive call because the client programmer can argue that the first actual parameter, $k \bmod j$, is no larger than j , even without checking. This conclusion is based purely on *reasoning* about the program behavior — about what **mod** can return. In other words, if you can establish by a valid argument that a precondition holds at the time of a call, then there is no reason to have *any* code to check that the precondition holds.²

From this example you can see how adding a test at the beginning of the operation body would only make this program less efficient than it is now, and no safer in terms of precondition violations. Well, this is not entirely true, because there is one call—from **main** in this case—that is not recursive. This is precisely the reason the main program does check the precondition of *Greatest_Common_Divisor* before calling it. The manifestation of this is that the end user who enters two integers that do not satisfy the precondition is informed:

Sorry, both integers must be non-negative, and one must be positive.

And this brings us to the second problem with the party line approach. Suppose *Greatest_Common_Divisor* were an operation provided by an off-the-shelf software component. If the precondition were checked in the body of the operation, the error message could not be expected to make nearly as much sense to the end user as the one above. Perhaps this is not

² This example illustrates (albeit unwittingly, as it is tangential to the real point here) something about recursion. The programmer who implements an operation by writing a recursive body for it is simultaneously the operation’s implementer *and* a client programmer calling that operation. This suggests that you should think of “client programmer” and “operation implementer” not as two different individuals, but as two different roles. Sometimes, one person plays both roles at the same time; sometimes different individuals are involved.

obvious from the present application example because the arguments to *Greatest_Common_Divisor* are just numbers and have no other application-specific meaning. But in another situation you might like the end user to see a message like this:

Sorry, the number of apples must be non-negative, and the number of oranges must be positive.

Obviously, the off-the-shelf component has no notion of apples or oranges, only of numbers, and cannot be expected to emit such a message if its precondition is violated. You can see the effect of this problem in the second error message from Euclid, which comes from the *Get_From* operation for *Integer* and which is, therefore, a lot less end-user friendly than the first error message.

This leaves us to examine case 4, which is to make the client programmer responsible for establishing that the precondition holds before making any call to any operation. The very presence of a non-trivial precondition in the specification is a direct statement of this responsibility. It means that the operation is allowed to do *anything* if that condition is not satisfied: *caveat emptor*. Sometimes, the client programmer is able to show by careful reasoning that the precondition for a call holds and therefore does not need to write code to test it. Other times, the client programmer is unable to prove that the precondition holds and must write code to test it, unless he/she is willing to settle for a program that is not really bulletproof. But in any event it is the client programmer who is responsible for establishing preconditions.

Bertrand Meyer calls this decision **design-by-contract**—we also use the minor variant **programming-by-contract**—because there is an official agreement between the client and the implementer of a component that lays down precisely which party is responsible for what. The client is responsible for making sure that the precondition of an operation holds at the time it is called, and the implementer is responsible for making sure that the postcondition holds at the time it returns. We state the client programmer's obligation under this agreement as follows:

Design-by-Contract Rule: Do not call an operation when its precondition is not satisfied, unless you are willing to accept *any behavior whatsoever* from that point onward during execution of the program.

The client programmer of Euclid has used this rule in two ways. He apparently has decided that it is unacceptable to call *Greatest_Common_Divisor* with bogus arguments, but that it is acceptable to call *Get_From* with an input stream that might not contain characters that can be interpreted as an *Integer* value. The program's reaction to a violation of this second precondition is whatever the built-in *Get_From* happens to do, which is to emit the less-friendly error message. To summarize, Euclid is not bulletproof because there is no guarantee that it won't simply crash (or do anything else) if the end user types in the wrong thing in response to a prompt for an integer value.³

³ All operations for Resolve/C++ built-in types check their preconditions in the instructional version, since efficiency is not nearly as important for student programs as the ability to isolate bugs. The Resolve/C++ design-by-contract rule is that client programmers are responsible for establishing preconditions, but students occasionally forget to do so, and automatically catching violated preconditions makes it easier to debug programs. Professional programmers also can benefit from using this approach during early software development stages at least.

There remains one plausible objection to the design-by-contract rule. It would be nice to be able to catch a client programmer's mistakes early in the development process—not after delivery of the program but rather during its development, when some errors are almost inevitable. Can this be done without proliferating throughout the client program precondition-checking code that later, according to design-by-contract, should be removed before delivery? Yes, by using templates.

3.2.2 Example: The *Natural* Family

The abstract instance *Random_Kernel* used in the Monte_Carlo example in Chapter 1 is somewhat unusual in that a client program may invoke any of its operations at any time. In other words, none of the operations has a *requires* clause; there is no way for the client of an implementation of *Random_Kernel* to violate the interface contract by failing to respect a precondition! So, let's look at a different abstract instance to illustrate how to address the problem of an explosion of precondition-checking code in client programs. This component allows you to manipulate arbitrarily large natural numbers, and (as is more typical of interface contracts) one of its operations has a ***non-trivial precondition***, i.e., its interface contract specification has a *requires* clause that is not necessarily true at the time of a call.

Objects of the built-in type *Integer* are limited to a fairly large but, practically-speaking, far from unlimited range of values: about -2 billion to $+2$ billion. There are many situations where you need only non-negative integer-valued objects but where two billion or so is not the largest value ever encountered. For example, the amount of an electronic bank transaction in U.S. currency might be recorded in an *Integer* object—the number of cents being transferred. But then a figure like the U.S. national debt or even a plausible inter-bank electronic fund transfer is too large to handle with an *Integer* object. A family of components in the Resolve/C++ Catalog is available to help you in this situation. The central abstract component for this family is *Natural_Kernel*. There are four kernel operations to manipulate a *Natural_Kernel* object through the digits of its ***radix representation***, as explained in the client-view appendix for the *Natural* family. Several additional operations are offered through extensions to provide more directly useful operations to do addition, subtraction, multiplication, etc. Figure 3-5 shows the interface contract specification of *Natural_Kernel*.

```
// /*-----*\
// |   Abstract Instance : Natural_Kernel
// \*-----*/

#ifndef AI_NATURAL_KERNEL
#define AI_NATURAL_KERNEL 1

///-----
/// Interface -----
///-----

abstract_instance
class Natural_Kernel
{
public:

    /*!
       math subtype NATURAL_MODEL is integer
       exemplar n
       constraint
       n >= 0
```

```

    math definition RADIX: integer satisfies restriction
        RADIX >= 2
    !*/

    standard_abstract_operations (Natural_Kernel);
    /*!
        Natural_Kernel is modeled by NATURAL_MODEL
        initialization ensures
            self = 0
    !*/

    procedure Multiply_By_Radix (
        preserves Integer k
    ) is_abstract;
    /*!
        requires
            0 <= k < RADIX
        ensures
            self = #self * RADIX + k
    !*/

    procedure Divide_By_Radix (
        produces Integer& k
    ) is_abstract;
    /*!
        ensures
            #self = self * RADIX + k and
            0 <= k < RADIX
    !*/

    function Integer Discrete_Log () is_abstract;
    /*!
        ensures
            if self = 0
            then Discrete_Log = 0
            else RADIX^(Discrete_Log - 1) <= self < RADIX^(Discrete_Log)
    !*/

    function Integer Radix () is_abstract;
    /*!
        ensures
            Radix = RADIX
    !*/

};

#endif // AI_NATURAL_KERNEL

```

Figure 3-5: AI/Natural/Kernel.h

Note that *Multiply_By_Radix* has a non-trivial precondition. Intuitively, the following call:

```
n.Multiply_By_Radix (k);
```

alters *n* by tacking the digit *k* onto the right end of the digits of the old *n*. The precondition is that *k* must actually be a “digit” in the radix used for *n*, i.e., it must be between 0 and *RADIX*–1,

inclusive.⁴ Design-by-contract says it is ultimately the client’s responsibility to make sure k has a legal value, and that an implementation of *Natural_Kernel* (say, *Natural_Kernel_1*) should not check this condition in the body of *Multiply_By_Radix*. *Natural_Kernel* can, therefore, help to illustrate how to write precondition-checking code for a component operation just once, and package it into a component so it works for all of the operation’s call sites in all client programs.

3.2.3 Checking Components and the Checks Relation

The recommended approach to catching erroneous calls to component operations during client program development is to create and use a **checking component**. A checking component for *Natural_Kernel* is a concrete component that “wraps” an implementation of *Natural_Kernel* (e.g., *Natural_Kernel_1*) inside a protective wall and thereby defends that implementation against a client programmer’s failure to observe the design-by-contract rule. As illustrated in Figure 3-6, the checking component **checks** the wrapped implementation by using the following strategy in the body of *Multiply_By_Radix*:

- Check the precondition of *Multiply_By_Radix*.
- Report a violated assertion to the client and halt execution immediately if the precondition is not satisfied.
- Continue the computation normally by calling *Multiply_By_Radix* from the wrapped implementation if the precondition is satisfied.

Since none of the other *Natural_Kernel* operations has a non-trivial precondition, the checking component need do nothing special to protect against violations of their requires clauses because there can’t be any such violations: every call to one of these operations is a good call that goes through the protective wall without difficulty.

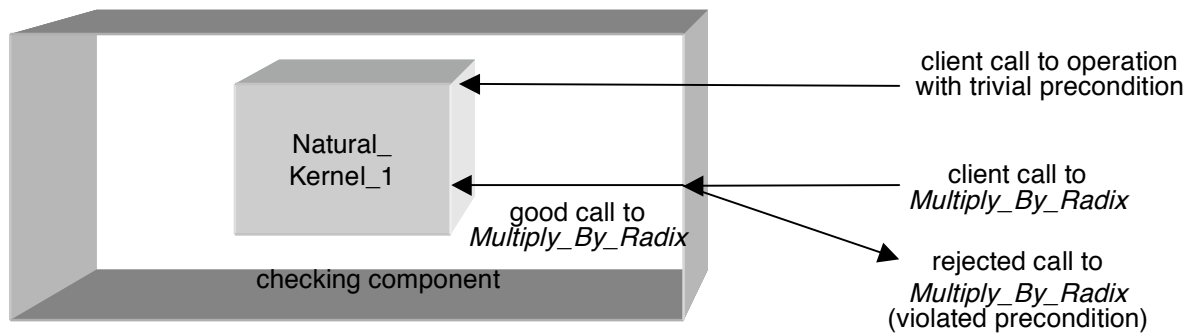


Figure 3-6: Intuitive operation of a checking component

A checking component is a concrete component: it contains code that is executed, not an interface contract. Specifically, the code for *Multiply_By_Radix* in the checking component depicted above might look like this:

⁴ You will not go wrong by assuming $RADIX = 10$, in which case you’re dealing with numbers as represented using the normal decimal digits. With this assumption, if $n = 123$ and $k = 4$ at the time of the call, then $n = 1234$ and $k = 4$ after the call.

```

procedure_body Multiply_By_Radix (
    preserves Integer k
)
{
    assert (0 <= k,
           "0 <= k");
    assert (k < self.Radix (),
           "k < RADIX");
    self.Natural_Kernel_1::Multiply_By_Radix (k);
}

```

A new Resolve/C++ keyword in this code is **assert**,⁵ which you can think of as the name of a built-in global procedure with two arguments. The first argument is a condition that you expect to be true. If the first argument is true, then the **assert** statement does nothing and program execution continues with the next statement. If the first argument is false, then the second argument (a *Text* value) is used in an error message that is output to the screen (where the form of the report is similar to that shown for the violated precondition of *To_Integer* in Euclid in Section 3.2.1), after which **assert** aborts program execution. Since the second argument is reported as the assertion that has been “violated”, it is phrased as something that should have been true but surprisingly was not. In the example code, note the subtle difference between the two arguments: the first is a computation of the value of a mathematical assertion, and the second is the mathematical assertion itself as a *Text* value.

You should use **assert** only where you expect the tested condition to be true under normal circumstances. It should be used only to signal truly abnormal situations from which the program cannot recover and for which immediately stopping execution with an error message is an appropriate response. It normally should be used only to give information to a client programmer who is still developing and/or testing a program, not to give an error message to the end user.

The call to the underlying *Multiply_By_Radix* in the component being checked uses the following long name:

```
Natural_Kernel_1::Multiply_By_Radix
```

This is the *fully qualified name* for the *Multiply_By_Radix* operation in the class *Natural_Kernel_1*; you may leave whitespace around “::” or omit it. You must use the qualified name here because the C++ compiler would interpret a call to an unqualified *Multiply_By_Radix* as a recursive call to the checking component’s code!

Finally, notice that the receiver object in this call is **self**⁶. As you know, C++ uses traditional object-oriented programming syntax in method calls (not global operation calls). Specifically, you put the name of the receiver object, or distinguished argument, before a “.” and the name of the method after it. Recall from Chapter 1 that this syntax causes a problem when writing specifications: there is no formal parameter associated with the receiver object, hence no

⁵ C++ compilers support a version of **assert** with slightly different syntax, with the exact format of the arguments and the effect when the condition evaluates to false also being somewhat different.

⁶ In C/C++ programs, you might use “(*this)” in place of the Resolve/C++ keyword **self**. Alternatively, in C/C++ you would be allowed to write “this->” in place of “**self**.”, or simply leave out the receiver object altogether. In Resolve/C++, the convention is never to call a method without an explicit receiver object.

programmer-declared name you can use to refer to it in the operation's specification. For specifications we therefore reserve the special name *self* for this purpose.

Similarly, there is no programmer-declared name to refer to the receiver object in an operation's body. So in Resolve/C++, **self** is the name reserved for the implicit formal parameter that is bound to the receiver object. If *n* is an object of type *Natural_Kernel_1*, then the statement:

```
n.Multiply_By_Radix (k);
```

results in **self** being bound to *n* in the operation body. Any changes to **self** as a result of executing the body of *Multiply_By_Radix* are reflected in object *n* back in the calling program after the call returns. In other words, **self** in the checking code above is the direct counterpart of *self* in the specification of *Multiply_By_Radix*.

There is a problem with the above code, though: the checking component in which it appears introduces a dependence on *Natural_Kernel_1*, another concrete component. This violates the concrete component dependence rule. To complete the discussion of checking components, let's see how to decouple this dependence using templates.

3.2.4 Implementing a Checking Component

The checking component code actually does not depend in any logical way on *Natural_Kernel_1*; the same code would work with any implementation of *Natural_Kernel* if that concrete component's name could be substituted for *Natural_Kernel_1* in the fully qualified name used in the code. No problem! You can parameterize the checking component by the implementation of *Natural_Kernel* that it checks. The code for this is shown in Figure 3-7.

```
// /*-----*\
// | Concrete Template : Natural_Kernel_C
// \*-----*/

#ifndef CT_NATURAL_KERNEL_C
#define CT_NATURAL_KERNEL_C 1

///-----
/// Global Context -----
///-----

/*!
 *include "AI/Natural/Kernel.h"
!*/

///-----
/// Interface -----
///-----

concrete_template <
    concrete_instance class Natural_Base
    /*!
        implements
        abstract_instance Natural_Kernel
    !*/
>
class Natural_Kernel_C :
    checks
    concrete_instance Natural_Base
```

```

{
public:

    procedure_body Multiply_By_Radix (
        preserves Integer k
    )
    {
        assert (0 <= k,
            "0 <= k");
        assert (k < self.Radix (),
            "k < RADIX");
        self.Natural_Base::Multiply_By_Radix (k);
    }
};

#endif // CT_NATURAL_KERNEL_C

```

Figure 3-7: CT/Natural/Kernel_C.h

The suffix “_C” in the name of the component is a Resolve/C++ convention indicating that the component checks the preconditions of calls to all operations that have non-trivial preconditions.

Notice that the **#include** statement in Figure 3-7 appears in a formal comment. This strange construction is used when the C++ compiler doesn’t need to include the file in order to compile this code, but the human reader does need to understand the component whose code is in the included file. In other words, from the standpoint of the logical dependencies between components that are recorded in a CCD, *Natural_Kernel_C* depends on *Natural_Kernel*. Because of the way Resolve-style components are encoded in C++, you do not directly express this dependence in the code needed by the compiler.

The logical dependence on *Natural_Kernel* appears in the template’s parameter list. The declaration of the formal parameter *Natural_Base* is followed by a **restriction clause**, a formal comment that says *Natural_Base implements Natural_Kernel*. In the *Sequence* example in Chapter 2, the formal parameter *Item* was unrestricted: the client programmer may bind any Resolve/C++ type to *Item*. Not so here. When instantiating *Natural_Kernel_C*, the client programmer must bind to the formal parameter *Natural_Base* a concrete instance that implements *Natural_Kernel*—it may be *Natural_Kernel_1*, or *Natural_Kernel_2*, or any other implementation of *Natural_Kernel* that exists now or might be developed in the future; but it may not be, say, *Text*. Thus, *Natural_Kernel_C* depends on *Natural_Kernel* because the client programmer must know about *Natural_Kernel*, at least enough to decide whether the concrete component to be used as the actual parameter claims to implement it.

The other new feature in this code is the declaration of *Natural_Kernel_C* itself, which says:

```

class Natural_Kernel_C :
    checks
        concrete_instance Natural_Base

```

This code—which the C++ compiler does care about—means that every concrete instance created from concrete template *Natural_Kernel_C*, by fixing the parameter *Natural_Base*, depends on *Natural_Base* in the stated way, i.e., it checks it. Moreover, it means that operations

of the checked implementation that have no preconditions are executed directly from that component with no screening by the checking component.⁷

Notice that this design succeeds in decoupling *Natural_Kernel_C* from any particular concrete component, replacing the concrete-to-concrete dependence in the original code (*Natural_Kernel_1_C* on *Natural_Kernel_1*) with a concrete-to-abstract dependence (*Natural_Kernel_C* on *Natural_Kernel*). The *checks* relation is a concrete-to-concrete dependence, but the concrete component at the target of the *checks* arrow in the CCD for *Natural_Kernel_C* in Figure 3-8 is a little black rectangular **parameter box**. This box stands for a restricted formal template parameter of the component *Natural_Kernel_C*, which is depicted with thick sides because it is a template.

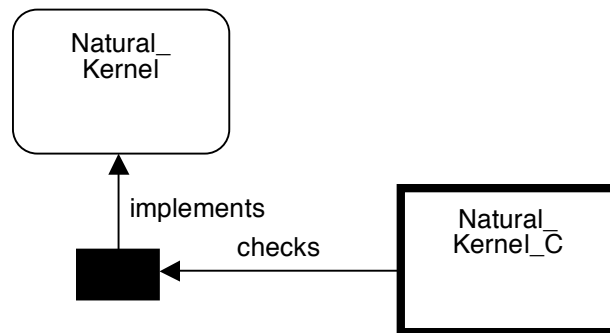


Figure 3-8: CCD for *Natural_Kernel_C*

This CCD shows that there is still a concrete-to-concrete dependence: *Natural_Kernel_C* checks some concrete component (note that the parameter box is shaped like a concrete component to emphasize that the actual parameter is a concrete instance). But which concrete component is being checked? You don't know from the CCD because this decision is not known at design time. Instead, the choice has been deferred to instantiation time by making the checked component a template parameter. What you can tell from the CCD, from the arrow out of the parameter box, is that the component being checked must be some concrete instance that implements *Natural_Kernel*.

If a component has a template parameter with no restriction on it, then there is no parameter box for that formal parameter in the component's CCD because an unrestricted template parameter induces no dependencies on any other *specific* software component. For example, the CCDs for the *Sequence* components in Chapter 2 do not have a parameter box for *Item* because it is an unrestricted template parameter. Remember, the purpose of the CCD is to show design-time dependencies, and a parameter box for an unrestricted template parameter would have no arrows out of it and therefore would introduce and make apparent no dependencies whatsoever; there is no sense even drawing it.

3.2.5 Instantiating a Checking Component

Now suppose you are a client programmer who wants to use a checked implementation of *Natural_Kernel* (say, *Natural_Kernel_1*) during the software development process. You must,

⁷ In C/C++ programs, you would use the keywords **virtual public** in place of the Resolve/C++ keyword **checks**.

as usual, first bring into scope using **#include** statements the checking component as well as the implementation to be checked. Then, you simply instantiate the checking component like any other template:

```
concrete_instance
class Natural_Kernel_1_C :
    instantiates
        Natural_Kernel_C <Natural_Kernel_1>
{};
```

Now, you can declare objects of type *Natural_Kernel_1_C*:

```
object Natural_Kernel_1_C n, m;
```

Client calls to *Multiply_By_Radix* on the objects *n* and *m* are now checked for precondition violations, where they would not have been checked if their type had been *Natural_Kernel_1*.

Moreover, you can easily make *Natural_Kernel_1_C* a reusable component and put it into a shared component catalog where you and others can use it in a variety of application programs simply by bringing it into scope using **#include**. This code is illustrated in Figure 3-9; it contains no new features.

```
// /*-----*\
// | Concrete Instance : Natural_Kernel_1_C
// \*-----*/

#ifndef CI_NATURAL_KERNEL_1_C
#define CI_NATURAL_KERNEL_1_C 1

///-----
/// Global Context -----
///-----

#include "CI/Natural/Kernel_1.h"
#include "CT/Natural/Kernel_C.h"

///-----
/// Interface -----
///-----

concrete_instance
class Natural_Kernel_1_C :
    instantiates
        Natural_Kernel_C <Natural_Kernel_1>
{};

//-----

#endif // CI_NATURAL_KERNEL_1_C
```

Figure 3-9: CI/Natural/Kernel_1_C.h

3.2.6 Combining Generalization and Decoupling: The Specializes Relation

Usually, the kernel component being checked is not a concrete instance but rather a concrete template. Consider a checking component *Sequence_Kernel_C* for implementations of

Sequence_Kernel. The difference from *Natural_Kernel* is that *Sequence_Kernel* already has a template parameter for generalization.

Fortunately, it is easy to combine the uses of templates for generalization and decoupling. You just need to combine into a single formula parameter list the template parameters that are used for both purposes. The code for *Sequence_Kernel_C* illustrates this in Figure 3-10.

```
// /*-----*\
// | Concrete Template : Sequence_Kernel_C
// \*-----*/

#ifndef CT_SEQUENCE_KERNEL_C
#define CT_SEQUENCE_KERNEL_C 1

///-----
/// Global Context -----
///-----

/*!
#include "AT/Sequence/Kernel.h"
!*/

///-----
/// Interface -----
///-----

concrete_template <
    concrete_instance class Item,
    concrete_instance class Sequence_Base
    /*!
        implements
        abstract_instance Sequence_Kernel <Item>
    !*/
>
class Sequence_Kernel_C :
    checks
        concrete_instance Sequence_Base
{
public:

    procedure_body Add (
        preserves Integer pos,
        consumes Item& x
    )
    {
        assert (0 <= pos,
            "0 <= pos");
        assert (pos <= self.Length (),
            "pos <= |self|");
        self.Sequence_Base::Add (pos, x);
    }

    procedure_body Remove (
        preserves Integer pos,
        produces Item& x
    )
    {
```

```

        assert (0 <= pos,
                "0 <= pos");
        assert (pos < self.Length (),
                "pos < |self|");
        self.Sequence_Base::Remove (pos, x);
    }

    function_body Item& operator [] (
        preserves Integer pos
    )
    {
        assert (0 <= pos,
                "0 <= pos");
        assert (pos < self.Length (),
                "pos < |self|");
        return self.Sequence_Base::operator [] (pos);
    }
};

#endif // CT_SEQUENCE_KERNEL_C

```

Figure 3-10: CT/Sequence/Kernel_C.h

There are two new features in this code. Starting at the end, the fully-qualified name for the accessor is a bit strange. Technically, **operator[]** is the name of the accessor operation, but C++ offers special concise syntax for normal use, i.e., *s[pos]* rather than *s.operator[](pos)*. The special syntax is not available with the fully-qualified name.

Second, the restriction in the formal parameter list on *Sequence_Base* (the implementation of *Sequence_Kernel* to be checked) indicates that the type *Item* must be the same when *Sequence_Kernel_C* is instantiated as it is when the implementation to be checked is instantiated. Suppose you need sequences of integers and you want the operations on them to be checked. Following an analogous approach as with *Natural_Kernel_1_C* in Section 3.2.5, you can create two instances in the client program:

```

concrete_instance
class Sequence_Of_Integer_Base :
    instantiates
        Sequence_Kernel_1a <Integer>
{};

concrete_instance
class Sequence_Of_Integer :
    instantiates
        Sequence_Kernel_C <Integer, Sequence_Of_Integer_Base>
{};

```

The name of the first concrete instance is immaterial because it is used only as an actual parameter when creating the second instance. In fact, you could combine these two instantiations into one and not even bother to give a name to the concrete instance to be checked:

```

concrete_instance
class Sequence_Of_Integer :
    instantiates
        Sequence_Kernel_C <Integer, Sequence_Kernel_1a <Integer> >
{};

```


In this case, the latter looks simpler. But be careful. C++ requires that seemingly innocuous space between the two '>' characters! The error message from failing to type it is likely to be very cryptic. Luckily, someone who has done this before has put the code of Figure 3-11 into the Resolve/C++ Component Catalog. Notice how this code formats the actual parameters one per line, so the little “gotcha” mentioned above will not arise.

```
// /*-----*\
// | Concrete Template : Sequence_Kernel_1a_C
// \*-----*/

#ifndef CT_SEQUENCE_KERNEL_1A_C
#define CT_SEQUENCE_KERNEL_1A_C 1

///-----
/// Global Context -----
///-----

#include "CT/Sequence/Kernel_1a.h"
#include "CT/Sequence/Kernel_C.h"

///-----
/// Interface -----
///-----

concrete_template <
    concrete_instance class Item
>
class Sequence_Kernel_1a_C :
    specializes
        Sequence_Kernel_C <
            Item,
            Sequence_Kernel_1a <Item>
        >
{};

//-----

#endif // CT_SEQUENCE_KERNEL_1A_C
```

Figure 3-11: CT/Sequence/Kernel_1a_C.h

After bringing this component into scope, you can instantiate it as follows:

```
concrete_instance
class Sequence_Of_Integer :
    instantiates
        Sequence_Kernel_1a_C <Integer>
{};
```

Notice also a new component relation in the code of Figure 3-11: *specializes*. In the analogous code of Figure 3-9, the program creates a new concrete instance by instantiating a concrete template. The relation connecting a template and an instance created from it is, appropriately, called *instantiates*. The code of Figure 3-11 creates a new concrete template by *partially instantiating* another concrete template. That is, the number of template parameters for the client programmer to provide goes from two (in *Sequence_Kernel_C*) to one (in *Sequence_Kernel_1a_C*)—but the client still has to instantiate the template to use it. The

relation connecting these two templates is called *specializes*⁸ because the all-purpose concrete template *Sequence_Kernel_C* has been made less universal—but therefore easier to instantiate—by fixing one of its template parameters. Figure 3-12 shows this relationship in the CCD for *Sequence_Kernel_1a_C*.

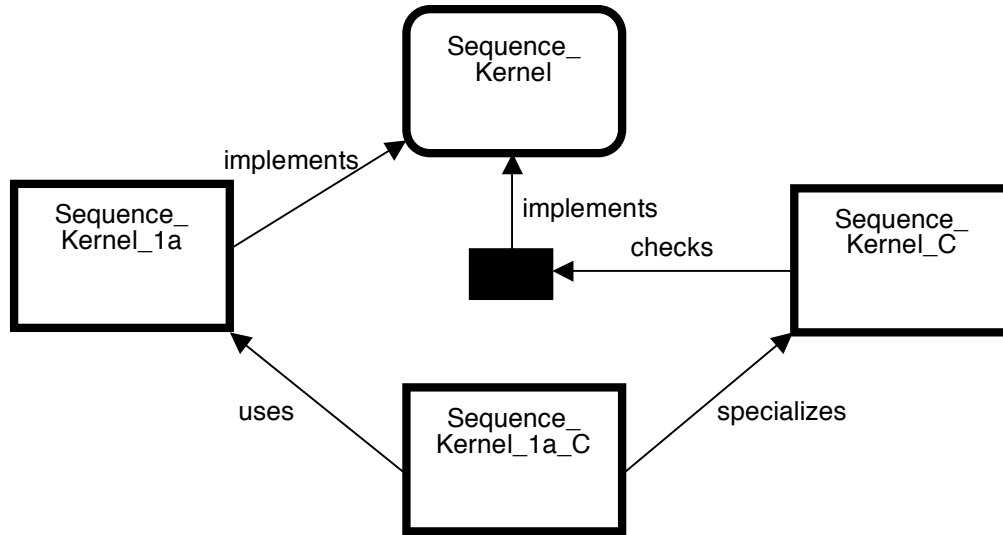


Figure 3-12: CCD for *Sequence_Kernel_1a_C* and the *specializes* relation

To summarize, then, there is a long and somewhat tortured story here. The defensiveness dilemma has to be addressed somehow; a case analysis of possible solutions leads to the selection of design-by-contract as the proper policy; that policy leads to the need for checking components; checking components introduce concrete-to-concrete dependencies that violate the concrete component dependence rule. How can you avoid these concrete-to-concrete dependencies? Use templates to decouple them. Then specialize the all-purpose checking component if you want to make template instantiation easier for a client programmer.

3.3 Decoupling the Extends Relation (Concrete-to-Concrete)

Now let's return to *Monte_Carlo*. It uses concrete instance *Random_Uniform_Generator_1*, which implements abstract instance *Random_Uniform*. Recall that *Monte_Carlo* uses the operation *Uniform_Real* from *Random_Uniform_Generator_1*. This function takes two *Real* numbers *a* and *b* and returns a random *Real* value drawn from a uniform probability distribution over the interval $[a, b)$.

⁸ In C/C++ programs, you would use the keywords **virtual public** in place of the Resolve/C++ keyword **specializes**.

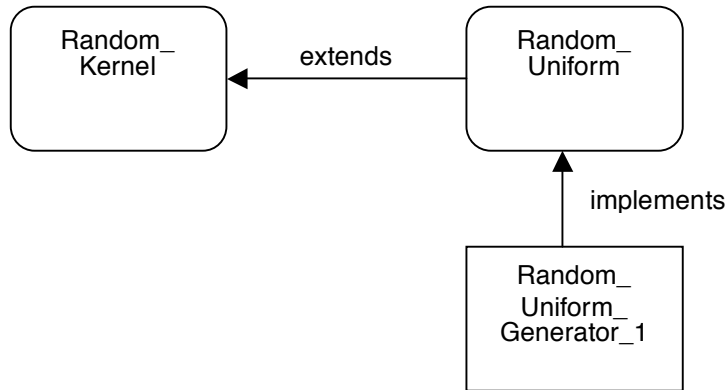


Figure 3-13: The CCD for an extension implementation

Figure 3-13 shows the CCD for *Random_Uniform_Generator_1* used in Chapter 1. The technique discussed here is how, as the client programmer of Monte_Carlo needing the functionality of *Random_Uniform*, you could have created *Random_Uniform_Generator_1* yourself, given only an implementation of *Random_Kernel* from the Resolve/C++ Catalog.

3.3.1 The Extends Relation (Concrete-to-Concrete): The Layering Approach

The computation you need to perform is rather simple. To get a random real number from a uniform distribution over $[0.0, 1.0)$, you can scale the value of a random integer between 0 and *LIMIT*–1 inclusive (i.e., *rn.Value()*) into this range by treating it as the numerator of a fraction whose denominator is *LIMIT* (i.e., *rn.Limit()*). Suppose *LIMIT* = 10000 for some particular implementation of *Random_Kernel*, as it turns out it is for the implementation *Random_Kernel_1*. Then if the value of *rn* is 2500, the expression:

```
To_Real (rn.Value ()) / To_Real (rn.Limit ())
```

evaluates to 0.25; if *rn* = 6978, then the expression evaluates to 0.6978; etc.

More generally, to get a random real number from a uniform distribution over $[a, b)$, the appropriate translation-and-scaling transformation is:

```
a + (b - a) * To_Real (rn.Value ()) / To_Real (rn.Limit ())
```

It is easy to put such code into a component for a shared catalog and make it generally useful beyond the current application. Just as a checking component can make calls to the underlying component it checks, a **concrete extension** can make calls to the underlying component it extends. (Normally, as in this case, the component being extended is a kernel implementation.) This technique for adding functionality to a component family is called **layering**. Calls to operations of the component being extended simply pass through the concrete extension layer unscathed, to be handled by the component being extended; while calls to the additional operations in the concrete extension do their work by calling the operations of the underlying component.

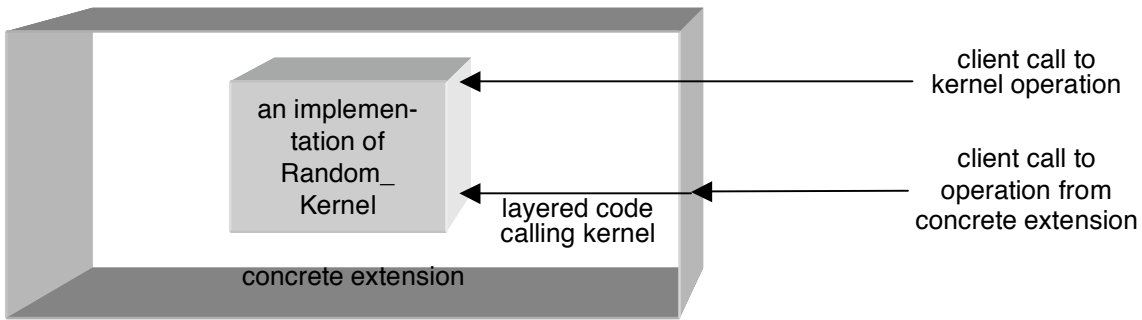


Figure 3-14: Intuitive operation of a concrete extension

The first step in the process of implementing *Random_Uniform* is to create a concrete template that *extends* some implementation of *Random_Kernel*. This concrete-to-concrete *extends* relation is entirely parallel in meaning to the abstract-to-abstract *extends* relation of Chapter 2. And the way to achieve it is entirely parallel to that used for a checking component: the code for the new operation *Uniform_Real* depends only on the interface contract *Random_Kernel*, not on any particular implementation of it. The implementation being extended therefore can be made a template parameter, to decouple what would otherwise be a concrete-to-concrete dependence.

```
// /*-----*\
// | Concrete Template : Random_Uniform_1
// \*-----*/

#ifndef CT_RANDOM_UNIFORM_1
#define CT_RANDOM_UNIFORM_1 1

///-----
/// Global Context -----
///-----

#include "AI/Random/Uniform.h"
/*!
    #include "AI/Random/Kernel.h"
!*/

///-----
/// Interface -----
///-----

concrete_template <
    concrete_instance class Random_Base
    /*!
        implements
        abstract_instance Random_Kernel
    !*/
>

class Random_Uniform_1 :
    implements
        abstract_instance Random_Uniform,
    extends
        concrete_instance Random_Base
{
```

```

public:

    function_body Real Uniform_Real (
        preserves Real a,
        preserves Real b
    )
    {
        return a + (b - a) *
            To_Real (self.Value ()) / To_Real (self.Limit ());
    }

    function_body Integer Uniform_Integer (
        preserves Integer j,
        preserves Integer k
    )
    {
        return j + To_Integer (self.Uniform_Real (0.0, To_Real (k+1-j)));
    }

};

#endif // CT_RANDOM_UNIFORM_1

```

Figure 3-15: CT/Random/Uniform_1.h

In the code in Figure 3-15, the concrete template depicted as the outer shell, or layer, in Figure 3-14 is called *Random_Uniform_1*, because it is an implementation of the abstract instance *Random_Uniform*.

There are a couple new features in this code. First, *Random_Uniform_1* takes part in two specific dependence relations with other components: it implements *Random_Uniform*, and it extends *Random_Base* (a restricted formal template parameter that must implement *Random_Kernel*). Notice that *Random_Uniform* extends *Random_Kernel*, so any implementation of *Random_Uniform* must implement *Random_Kernel* as well as the two new operations whose interface contracts are specified in *Random_Uniform*. The concrete-to-concrete *extends* relation makes this happen automatically: the kernel operations come from *Random_Base*, the component being extended, and the code for the two additional operations *Uniform_Real* and *Uniform_Integer* is provided here in *Random_Uniform_1*.

The second new feature is in the code for the new operations. Just as a checking component calls operations from the component it checks, a layered extension calls the operations from the component it extends. It does this by using **self** as the name of the receiver object, exactly as with a checking component. Notice that the body of an operation in an extension may call not only the operations from the component it extends, but other operations from the extension, as seen here in the body of *Uniform_Integer*.

Figure 3-16 shows the CCD for *Random_Uniform_1*. This pattern of organizing component dependencies is the primary technique for adding functionality to a component family. It works equally well when the components involved also use templates for generalization.

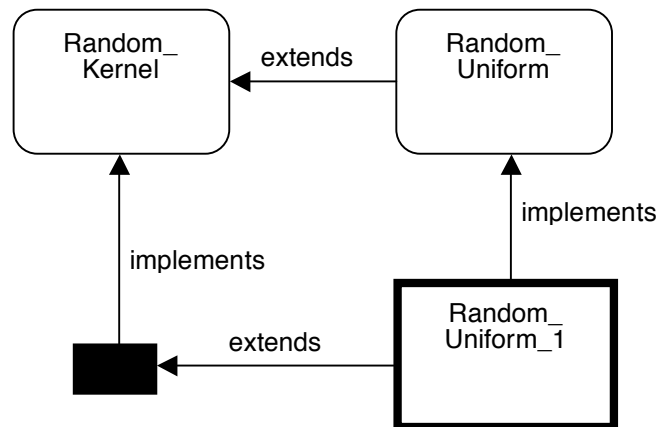


Figure 3-16: CCD for a typical layered extension

3.3.2 Instantiating a Concrete Extension

As a client programmer, instantiating a concrete extension is nothing new: you simply bring into scope the component(s) you need and then instantiate them. In this case, you need only *Random_Uniform_1* and some implementation of *Random_Kernel*. It happens that *Random_Kernel_1* is available; it is a template itself but happens to have no template parameters⁹. Here is how you might instantiate *Random_Kernel_1* and *Random_Uniform_1* to create a concrete instance class in your client code:

```
concrete_instance
class Random_Uniform_Generator :
    instantiates
        Random_Uniform_1 <Random_Kernel_1 <> >
{};
```

Note the formatting of the nested template instantiation with the space between the '>' characters.

It also is possible to make this concrete instance a component in a catalog, as shown in Figure 3-17. You simply **#include** the components used in the instantiation and instantiate them as above, giving the new component an appropriate name. In fact, this is the component used in Monte_Carlo as presented in Chapter 1.

```
// /*-----*\
// | Concrete Instance : Random_Uniform_Generator_1
// \*-----*/

#ifndef CI_RANDOM_UNIFORM_GENERATOR_1
#define CI_RANDOM_UNIFORM_GENERATOR_1 1

///-----
```

⁹ How can a template have no parameters? This sometimes arises when implementing kernel components, and is discussed only in Volume 3 because a client programmer simply has to know that such things exist and how to instantiate them. What is the difference between a template with no parameters and an instance? The latter has to be instantiated before it can be used.

```

/// Global Context -----
///-----

#include "CT/Random/Kernel_1.h"
#include "CT/Random/Uniform_1.h"

///-----
/// Interface -----
///-----

concrete_instance
class Random_Uniform_Generator_1 :
    instantiates
        Random_Uniform_1 <
            Random_Kernel_1 <>
        >
{};

///-----

#endif // CI_RANDOM_UNIFORM_GENERATOR_1

```

Figure 3-17: CI/Random/Uniform_Generator_1.h

Finally, Figure 3-18 shows the CCD for *Random_Uniform_Generator_1*. A comparison with the CCD in Figure 3-13 is instructive. The simpler CCD shown earlier is all a client of *Random_Uniform_Generator_1* need to know about it: it's a concrete instance and it implements *Random_Uniform*. The more complex CCD shows additional structure that explains how it implements *Random_Uniform*: it is an instance of a template, *Random_Uniform_1*, that implements *Random_Uniform* (and it happens to use *Random_Kernel_1* for the *Random_Kernel* operations).

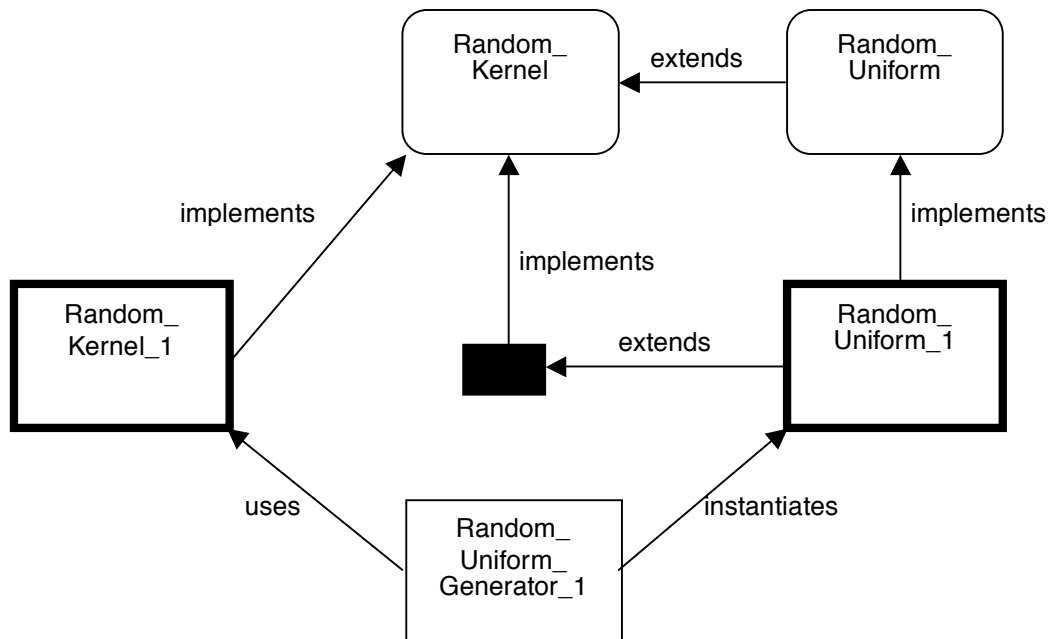


Figure 3-18: CCD for *Random_Uniform_Generator_1*

To summarize, using the same technique that decouples a checking component from the component it checks, you can decouple a concrete extension from the component it extends.

3.4 Summary

Generalizing collection components to work with arbitrary entry types is the most obvious use for abstract and concrete templates. Yet it is arguably not the most important one. That honor goes to decoupling concrete-to-concrete dependencies. Introducing concrete-to-concrete dependencies in code is as easy as falling off a log—but it leads to software that is needlessly difficult to understand and to change.

A disciplined software component designer can avoid concrete-to-concrete dependencies by observing the following principles:

- **Limited Dependence Rule:** Design each component so it depends on as few other components as possible.
- **Least Information Rule:** Design each component so it depends only on components that contain information that is required to understand the new component.

An important implication of these fundamental design rules is:

- **Concrete Component Dependence Rule:** Avoid introducing a dependence on a concrete component. If the dependence of a concrete component on another component is required, create that dependence to an abstract component.

There is a nice way to satisfy the requirements of the concrete component dependence rule, as illustrated in this chapter by two important examples of its application: checking components and concrete extensions. Simply put, in every new component create a formal template parameter (a concrete instance) for each abstract component whose interface contracts it depends on, restricting that parameter to be an implementation of the associated interface contract.

Checking components arise from an analysis of the defensiveness dilemma, the policy decision faced by every software engineer regarding which party to an interface contract is responsible for checking the preconditions of operations. A careful analysis clearly supports the policy known as design-by-contract, summarized as follows:

- **Design-by-Contract Rule:** Do not call an operation when its precondition is not satisfied, unless you are willing to accept *any behavior whatsoever* from that point onward during execution of the program.

This policy is logically sound as the one in force for a final delivered software product. During the software development process, however, a client programmer (being human) can make a mistake and inadvertently call an operation whose precondition is not satisfied. Debugging code with such an error is facilitated when it is detected and reported immediately at the point of occurrence—which does not happen under a strict interpretation of design-by-contract. Enter checking components, which are wrappers for components that have been developed using design-by-contract. The client programmer during the software development process wraps a checking component around the normal (unchecked) component that will be used in the final delivered product, substituting for it the original component only *after* he or she is convinced that it is being used fully in accordance with the design-by-contract rule.

Concrete extensions arise from the need to add functionality after a kernel component has been designed and implemented. There is no way a designer can hope to think of every useful

operation for a component “up front”, before it is released in a shared catalog to be used by others. The kernel operations must be reasonably powerful but cannot be all-encompassing. When a component family needs to be supplemented with a new operation, its interface contract specification can be added by using an abstract extension. The implementation of this new operation calls for a concrete extension: a component whose code is layered on top of an implementation of the kernel. The new operation works by calling the kernel operations to do its job.

In both these situations, there is an underlying concrete component that does the bulk of the work. Wrapped around it is a thin layer of new code that invokes the operations of the underlying component. Both the underlying component and the layer around it are concrete components, and clearly the latter depends on the former. The use of templates to decouple this concrete-to-concrete dependence is a crucial idea in software component engineering.

Client-View Appendices

- Array
- Binary_Tree
- Id_Name_Table
- List
- Natural
- Partial_Map
- Queue
- Record
- Sequence
- Set
- Sorting_Machine
- Stack
- Static_Array
- Text
- Tree

Array

Motivation, Applicability, and Indications for Use

The programming type *Array* allows you to set up and then repeatedly access and update a fixed-size table of items of an arbitrary type. You “index” into the table using an *Integer* value that lies within an interval determined when you set the “bounds” of the *Array* object. This set-up occurs after you declare the object and typically is done just once for each object; but you may reuse the same *Array* object by resetting its bounds.

For example, suppose you want to record the high temperature for each day of the month. To do this, you can declare an *Array* object whose items are of type *Real* and set it up to be indexed by the values 1 through 31 (e.g., for January). When you set these bounds, each entry in the table has an initial value for type *Real*, i.e., 0.0. You may access and update the value associated with each day. Once you are done processing data for January, you may use the same *Array* object to record the high temperature for each day of February, say: Simply reset the *Array*’s bounds to be 1 through 28 (or 29) and repeat the process.

The most closely related component is *Static_Array*, which is nearly identical except in regard to when the index bounds are fixed. Another closely related component is *Sequence*. A *Sequence* object starts out as an empty string of *Item* values, so if you want to use it as a table in the fashion mentioned above you have to add entries to it one by one until the table is populated with values. A *Array* object becomes populated with initial values of type *Item* as soon as you set its interval bounds. With a *Sequence* object you can interleave operations that change the size of the *Sequence* with operations that access and update existing entries, possibly deferring the decision to add more entries or to remove some; however, adding and removing changes the indices of some of the existing entries. With the *Array* object you can’t change the bounds of the table without completely wiping out the previous table and starting over with new bounds; you decide up front on the required size and bounds, and then you access and update table entries whose indices remained fixed throughout. The *Sequence* is always indexed starting with 0. The *Array* object may be set up to be indexed starting with any *Integer* value.

Related Components

- *Sequence* — a type that is similar to *Array* except that it is incrementally resizable, and that it is always indexed from 0
- *Static_Array* — a type that is essentially identical to *Array* except that the index bounds are fixed not at run time, but at template instantiation time

Component Family Members

Abstract Components

- *Array_Kernel* — the programming type of interest, with the operations below
 - *Set_Bounds*
 - *Accessor*
 - *Lower_Bound*

- *Upper_Bound*

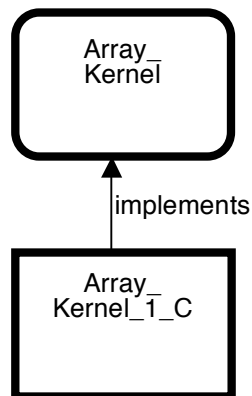
Concrete Components

- *Array_Kernel_1_C* — This is a checking implementation of *Array_Kernel* in which the execution time for each of the operations constructor, the accessor, *Lower_Bound*, and *Upper_Bound* is constant; the execution time for the destructor is proportional to the length of the interval between the upper and lower bounds; and the execution time for *Set_Bounds* is proportional to the sum of the length of the interval between the current upper and lower bounds and the length of the interval between the new lower and upper bounds. All objects of this type have the interface of *Array_Kernel*, with the concrete template name *Array_Kernel_1_C* substituted for the abstract template name *Array_Kernel*.

To bring this component into the context you write:

```
#include "CT/Array/Kernel_1_C.h"
```

Component Coupling Diagram



Descriptions

Array_Kernel Type and Standard Operations

Formal Contract Specification

```

    /*!
      math subtype INDEXED_TABLE is finite set of (
        i: integer,
        x: Item
      )
      exemplar t
      constraint
        for all i1, i2: integer, x1, x2: Item
          where ((i1,x1) is in t and
                (i2,x2) is in t)
            (if i1 = i2 then x1 = x2)

      math subtype ARRAY_MODEL is (
        lb: integer,
        ub: integer,
        table: INDEXED_TABLE
      )
      exemplar a
      constraint
        for all i: integer
          (there exists x: Item
            ((i,x) is in a.table)
            iff a.lb <= i <= a.ub)
    !*/

    standard_abstract_operations (Array_Kernel);
    /*!
      Array_Kernel is modeled by ARRAY_MODEL
      initialization ensures
        self = (1, 0, empty_set)
    !*/

```

Informal Description

The model for *Array_Kernel* is a three-tuple: an integer which is the lower bound of the interval of legal indices into a table, an integer which is the upper bound of that interval, and the table itself: a finite set of ordered pairs of type (integer, *Item*) that contains exactly one pair for each integer within the interval from the lower bound to the upper bound, inclusive. Initially the lower bound is 1 and upper bound is 0. The constraint of the model implies that the set of pairs is, therefore, initially empty.

Like all Resolve/C++ types, *Array_Kernel* comes with operator `&=` and *Clear*. Notice that the bounds on the interval of legal indices is part of the model of an individual *Array_Kernel* object, so it is perfectly OK to swap two *Array_Kernel* objects that have different bounds. (Of course, these two *Array_Kernel* objects still must have the same *Item* type, or they are not of the same type and therefore cannot be swapped with each other.)

Note — The sample traces for this and the other operation descriptions refer to the type *Array_Of_Text*, which is the result of the following instantiation:

```
concrete_instance
class Array_Of_Text :
    instantiates
        Array_Kernel_1_C <Text>
{};
```

Sample Traces

Statement	Object Values
	(a1 is not in scope)
object Array_Of_Text a1;	
	a1 = (1, 0, { })

Set_Bounds

Formal Contract Specification

```

procedure Set_Bounds (
    preserves Integer lower,
    preserves Integer upper
) is_abstract;
/*!
    ensures
        self.lb = lower and
        self.ub = upper and
        for all i: integer
            (there exists x: Item
                ((i,x) is in self.table and
                 is_initial (x)) iff
                 self.lb <= i <= self.ub)
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Array_Of_Text a1, a2;
object Text t;
object Integer i1, i2;

```

A call to the *Set_Bounds* operation has the form:

```

a1.Set_Bounds (i1, i2);

```

This operation sets the lower and upper bounds on the interval of legal indices into *a1* to be *i1* and *i2*, respectively. It also makes the value in each position of *a1.table* an initial value for type *Item*.

Because the initial value of a newly declared *Array_Type* object makes it essentially useless until the bounds are changed, *Set_Bounds* is generally the first operation called for a new object of the type.

Sample Traces

Statement	Object Values
	a1 = (7, 9, { (7,"abcd"), (8,"XYZ"), (9,"bucks") }) i1 = 3 i2 = 4
a1.Set_Bounds (i1, i2);	
	a1 = (3, 4, { (3,""), (4,"") }) i1 = 3 i2 = 4

Accessor

Formal Contract Specification

```
function Item& operator [] (
    preserves Integer i
) is_abstract;
/*!
    requires
        self.lb <= i <= self.ub
    ensures
        (i, self[i]) is in self.table
!*/
```

Informal Description

A call to the accessor operator [] has the form of an expression:

```
... a1[i1] ...;
```

This expression acts as the name of an object of type *Item* (i.e., *Text* in this case) whose value is that paired with *i1* in *a1*. You may make this call only if there is a pair in *a1.table* whose first component is *i1*, i.e., only if *i1* lies between *a1.lb* and *a1.ub* inclusive.

You may use *a1[i1]* wherever any other object of type *Item* (i.e., *Text* in this case) may appear.

The accessor operator [], like all functions, preserves its arguments. But it is important to realize that the accessor expression *a1[i1]* does not simply denote the value associated with *i1* in *a1*, it acts as the name of an object of type *Item* which you may consider to be in that particular pair in *a1.table*. This means that not only may you use the expression *a1[i1]* as a value of type *Item*; you may even change the value of the object called *a1[i1]*—but remember that this also changes the value of *a1*.

The sample traces help illustrate some important points regarding the convenience and danger of accessor expressions. The first and second sample statements show how an accessor expression such as *a1[i1]* acts like an object of type *Item* whose value may be changed, depending on the statement in which the expression occurs. The third sample statement's call to the swap operator involves other arguments (e.g., *a2*), but none of the other arguments may be *a1* or may involve an accessor expression for *a1* because this would violate the repeated argument rule. So, while you might be tempted to write something like:

```
a1[i1] &= a1[i2];
```

this statement violates the repeated argument rule. Be aware that the C++ compiler will not report this as an error. You need to avoid the pitfall by personal discipline.

Sample Traces

Statement	Object Values
	<pre> a1 = (7, 9, { (7, "abcd"), (8, "XYZ"), (9, "bucks") }) i1 = 8 t = "go" </pre>
<code>a1[i1] &= t;</code>	
	<pre> a1 = (7, 9, { (7, "abcd"), (8, "go"), (9, "bucks") }) i1 = 8 t = "XYZ" </pre>
<code>a1[8] = t;</code>	
	<pre> a1 = (7, 9, { (7, "abcd"), (8, "XYZ"), (9, "bucks") }) i1 = 8 t = "XYZ" </pre>
<code>...</code>	
	<pre> a1 = (7, 9, { (7, "abcd"), (8, "go"), (9, "bucks") }) a2 = (10, 11, { (10, ""), (11, "cd") }) i1 = 8 i2 = 11 </pre>
<code>a1[i1] &= a2[i2];</code>	
	<pre> a1 = (7, 9, { (7, "abcd"), (8, "cd"), (9, "bucks") }) a2 = (10, 11, { (10, ""), (11, "go") }) i1 = 8 i2 = 11 </pre>

Lower_Bound

Formal Contract Specification

```

function Integer Lower_Bound () is_abstract;
/*!
    ensures
        Lower_Bound = self.lb
!*/

```

Informal Description

A call to the *Lower_Bound* operation has the form of an expression:

```
... a1.Lower_Bound () ...;
```

This operation returns an *Integer* value which is the lower bound on the interval of legal indices into *a1*, i.e., *a1.lb*.

Sample Traces

Statement	Object Values
	<pre> a1 = (7, 9, { (7,"abcd"), (8,"go"), (9,"bucks") }) i1 = -32 </pre>
<code>i1 = a1.Lower_Bound ();</code>	
	<pre> a1 = (7, 9, { (7,"abcd"), (8,"go"), (9,"bucks") }) i1 = 7 </pre>

Upper_Bound

Formal Contract Specification

```

function Integer Upper_Bound () is_abstract;
/*!
    ensures
        Upper_Bound = self.ub
!*/

```

Informal Description

A call to the *Upper_Bound* operation has the form of an expression:

```
... a1.Upper_Bound () ...;
```

This operation returns an *Integer* value which is the upper bound on the interval of legal indices into *a1*, i.e., *a1.ub*.

Sample Traces

Statement	Object Values
	<pre> a1 = (7, 9, { (7,"abcd"), (8,"go"), (9,"bucks") }) i1 = -32 </pre>
<code>i = a1.Upper_Bound ();</code>	
	<pre> a1 = (7, 9, { (7,"abcd"), (8,"go"), (9,"bucks") }) i1 = 9 </pre>

Binary_Tree

Motivation, Applicability, and Indications for Use

The programming type *Binary_Tree* is ubiquitous in computing. It is commonly used in formal language processing applications, as the basis for fast searching algorithms (as might be used in an implementation of *Set* or *Partial_Map*), and in many other situations where the problem at hand has the distinctive recursive structure of divide-and-conquer algorithms.

The mathematics used to model trees requires some explanation. Figure 1 shows a tree called *T*. The **root** of *T* is *d*, the item (also called a **node**) at the top of the tree; *d* has two children, where *a* is the **left child** and *k* is the **right child**. Similarly, *a* has only a left child *h*, and *h* has only a right child *g*; while *k* has two children, with *s* being the left child and *t* the right. Since *g*, *s*, *x*, and *y* have no children, they are called the **leaves** of *T*. All other items in *T* are **interior** items.

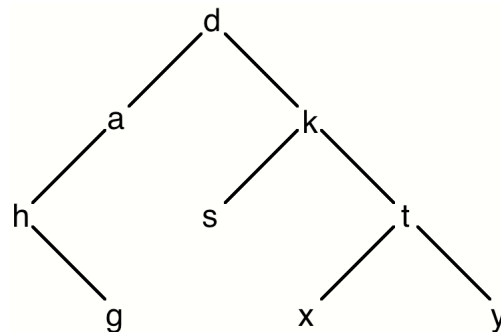


Figure 1: A binary tree *T*

Continuing the “family tree” metaphor that gives rise to some (but, as you can see, not all) of the terms here, *k* is the **parent** of both *s* and *t*, and *s* and *t* are **siblings**. *T* is a **binary** tree since every item in *T* has at most two children; that is, every item has 0, 1, or 2 children. The string of items $\langle g, s, x, y \rangle$ is the **frontier** or **yield** of *T*: the string of leaves of *T* from left to right.

There are several **subtrees** of *T*. Some of these are pictured in Figure 2.

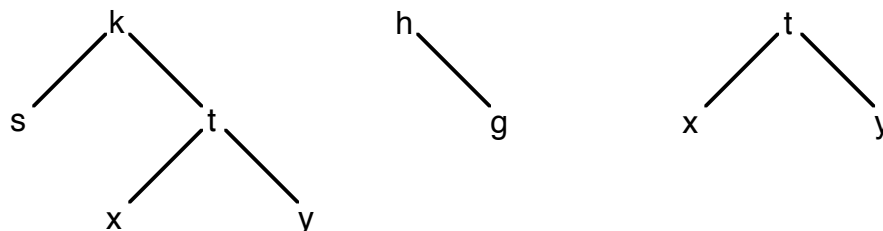


Figure 2: Some subtrees of *T*

Binary trees have a particular structure or composition. Specifically, every binary tree (except for the empty binary tree; see below) consists of a root item together with two subtrees, called the **left subtree** and **right subtree**. The tree *T* consists of the root item *d* and the left and right subtrees *L* and *R* pictured in Figure 3. Notice that each of the left and right subtrees of *T* has exactly the same type of structure as *T* itself. For example, the right subtree of *T*, in Figure 3 called *R*, consists of a root item *k* and the left and right subtrees *L'* and *R'*; and so on for *L'* and *R'*.

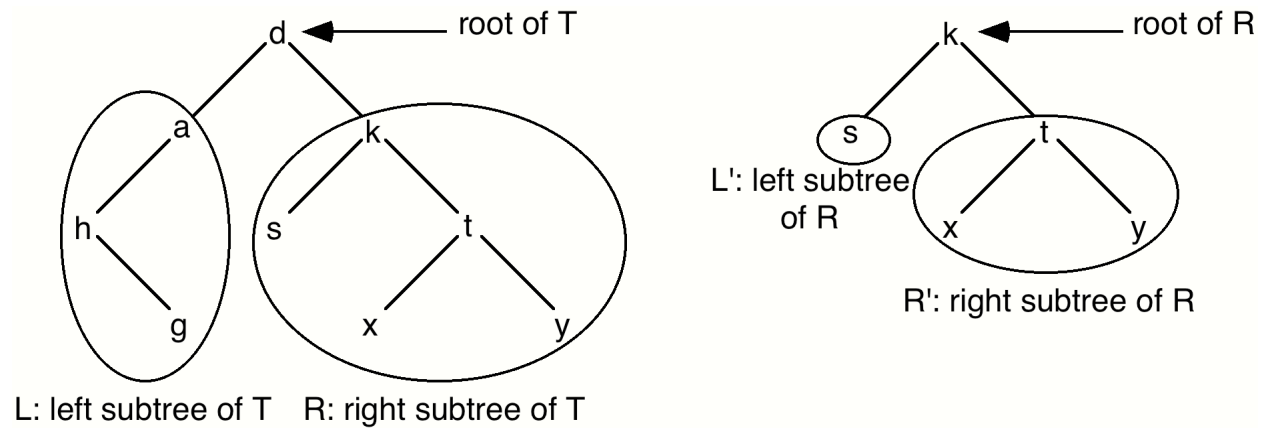


Figure 3: The abbreviated structure of T and R

The left subtree L' of R is interesting. It appears to consist of only a root item s , and nothing more. This is not quite right. Actually, it consists of a root item s and left and right subtrees, each of which is an *empty tree*. Empty trees are trees containing no items. (Empty trees in tree theory play the same role as do empty strings in string theory and empty sets in set theory.) So in fact, every leaf item in tree T is the root of a subtree where both the left and right subtrees are empty trees. With this in mind, a more accurate depiction of the tree T is given in Figure 4.

Generally, trees are drawn as in Figure 1, omitting the empty trees seen in Figure 4. Even though they may not be shown, don't forget that the empty trees are there! Also, note that this short cut in drawing trees results in some ambiguity. Specifically, when we write s , by itself, it is ambiguous as to whether we mean the tree with root s and left and right subtrees that are empty, or just the item s . In most cases, the type and therefore the intention should be clear from context. When it is not, we'll try to be careful to say exactly what we mean.

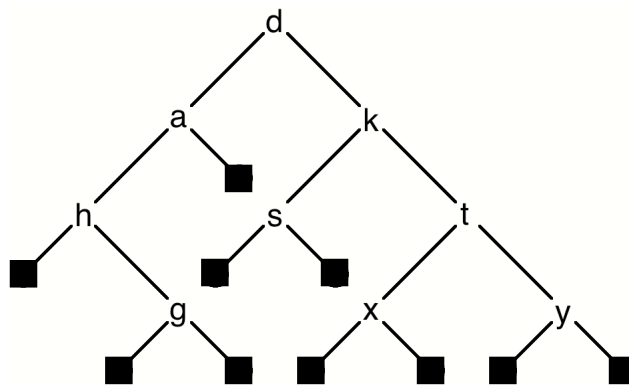


Figure 4: The full structure of T

The *size* of T , denoted $|T|$, is the number of items in T ; e.g., in Figure 1 as in Figure 4, $|T| = 9$. The *height* of T is the number of items on the longest *path* from the root to some leaf. (The height of an empty tree is defined to be 0.) Again referring to Figure 1 or Figure 4, you can see the height of T is 4 since the path (string of items) $\langle d, k, t, x \rangle$ has length 4; so do two other paths, but no path is longer than 4. The notion of a path also gives rise to that of a level. An item

occurs on **level** k in a tree if the length of the path¹ from the root to that item has length k . In T of Figure 1 and Figure 4, d is on level 1; a and k are on level 2; h , s , and t are on level 3; etc. Notice that the empty trees shown in the full structure of Figure 4 do not contribute either to the size or the height of T .

A binary tree is **complete** if all the leaves occur on at most two different levels *and* if all the leaves at the bottom (highest-numbered) level are as far to the left as possible. This means that T is not a complete binary tree, nor are the first two trees shown in Figure 2. But the third tree in Figure 2 is complete. The first tree of Figure 2 would be complete if the subtrees whose roots are s and t were exchanged, and the second tree of Figure 2 would be complete if g were the left subtree and not the right subtree of h .

We are now ready to define the mathematical idea of “binary trees over a set of items”. That is, *binary tree of A* is the name of a mathematical type, and we need to define the set of values associated with the type, assuming we know the set of values associated with mathematical type A . In tree theory there is a special mathematical operation (function), the **compose** function, that takes as arguments a root item from A , say x , a binary tree of A , say $T1$, and another binary tree of A , say $T2$, and produces a binary tree of A , say $T3$, with x as the root of $T3$, with $T1$ as the left subtree of $T3$, and with $T2$ as the right subtree of $T3$. This is pictured in Figure 5.

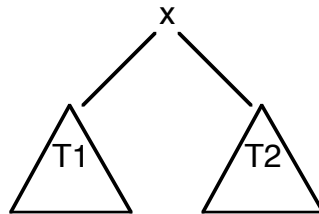


Figure 5: The tree $T3 = \text{compose}(x, T1, T2)$

We now define inductively the set of “binary trees over set A ” (or *binary tree of A*, which is the type name you use when writing the mathematics of trees). This set consists of exactly those binary trees that can be built up from the empty tree by zero or more applications of the *compose* function according to the following cases:

- Base case: The empty tree, denoted *empty_tree*, is an element of binary trees over set A .
- Inductive case: If $T1$ and $T2$ are elements of binary trees over set A and if x is an element of A , then *compose* ($x, T1, T2$) is an element of binary trees over set A .

As an example, here is a complete justification that the leftmost tree pictured in Figure 2 is a binary tree according to the definition just given:

- *compose* ($s, \text{empty_tree}, \text{empty_tree}$) is a binary tree, say $T1$.
- *compose* ($x, \text{empty_tree}, \text{empty_tree}$) is a binary tree, say $T2$.
- *compose* ($y, \text{empty_tree}, \text{empty_tree}$) is a binary tree, say $T3$.
- *compose* ($t, T2, T3$) is a binary tree, say $T4$.
- *compose* ($k, T1, T4$) is a binary tree, the one we’re looking for.

¹ Notice that we say *the* path because there is *exactly one* path from the root to each item in a tree. This property is the basis for another way to characterize trees.

Related Components

- None

Component Family Members

Abstract Components

- *Binary_Tree_Kernel* — the programming type of interest, with the operations below
 - *Compose*
 - *Decompose*
 - *Accessor*
 - *Height*
 - *Size*

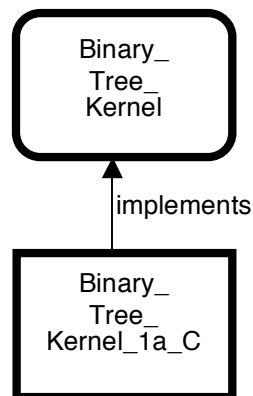
Concrete Components

- *Binary_Tree_Kernel_1a_C* — This is a checking implementation of *Binary_Tree_Kernel* in which the execution time for each of the operations constructor, *Compose*, *Decompose*, accessor, and *Size* is constant, while the execution time for the destructor is proportional to the size of the tree. All objects of this type have the interface of *Binary_Tree_Kernel*, with the concrete template name *Binary_Tree_Kernel_1a_C* substituted for the abstract template name *Binary_Tree_Kernel*.

To bring this component into the context you write:

```
#include "CT/Binary_Tree/Kernel_1a_C.h"
```

Component Coupling Diagram



Descriptions

Binary_Tree_Kernel Type and Standard Operations

Formal Contract Specification

```

standard_abstract_operations (Binary_Tree_Kernel);
/*!
    Binary_Tree_Kernel is modeled by binary tree of Item
    initialization ensures
        self = empty_tree
!*/

```

Informal Description

The model for *Binary_Tree_Kernel* is a binary tree of items which is initially empty. A *Binary_Tree_Kernel* object may be arbitrarily large. Like all Resolve/C++ types, *Binary_Tree_Kernel* comes with operator `&=` and *Clear*.

Note — The sample traces for this and the other operation descriptions refer to the type *Binary_Tree_Of_Integer*, which is the result of the following instantiation:

```

concrete_instance
class Binary_Tree_Of_Integer :
    instantiates
        Binary_Tree_Kernel_1a_C <Integer>
{};

```

Sample Traces

Statement	Object Values
	(t1 is not in scope)
object Binary_Tree_Of_Integer t1;	
	t1 = ■

Compose

Formal Contract Specification

```
procedure Compose (  
    consumes Item& x,  
    consumes Binary_Tree_Kernel& left_subtree,  
    consumes Binary_Tree_Kernel& right_subtree  
    ) is_abstract;  
/*!  
    produces self  
    ensures  
        self = compose (#x, #left_subtree, #right_subtree)  
!*/
```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:








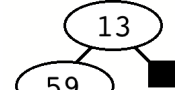




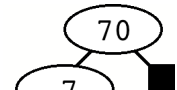



```
object Binary_Tree_Of_Integer t1, t2, t3, t4;  
object Integer i;
```

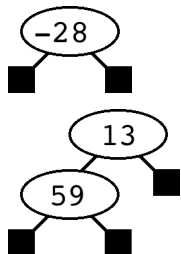
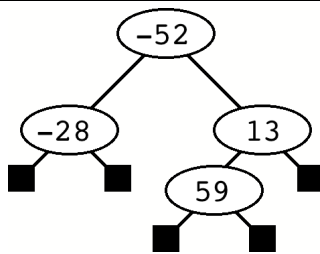
A call to the *Compose* operation has the form:

```
t1.Compose (i, t2, t3);
```

This operation produces as the value of *t1* a binary tree whose root value is *i*, and whose left and right subtree values are *t2* and *t3*.

Sample Traces

Statement	Object Values
	t1 =  t2 =  t3 =  i = 4
t1.Compose (i, t2, t3);	
	 t1 =  t2 =  t3 =  i = 0
...	
	 t1 =   t2 =  t3 =  i = 70
t1.Compose (i, t2, t3);	
	 t1 =  t2 =  t3 =  i = 0
...	

	<p>t1 = ■</p> <p>t2 = </p> <p>t3 = ■</p> <p>i = -52</p>
<code>t1.Compose (i, t2, t3);</code>	
	<p></p> <p>t1 = ■</p> <p>t2 = ■</p> <p>t3 = ■</p> <p>i = 0</p>

Decompose

Formal Contract Specification

```
procedure Decompose (  
    produces Item& x,  
    produces Binary_Tree_Kernel& left_subtree,  
    produces Binary_Tree_Kernel& right_subtree  
    ) is_abstract;  
/*!  
    consumes self  
    requires  
        self /= empty_tree  
    ensures  
        #self = compose (x, left_subtree, right_subtree)  
!*/
```

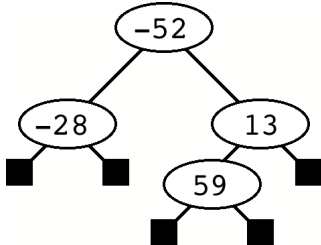




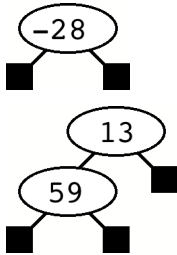









Informal Description

A call to the *Decompose* operation has the form:

```
t1.Decompose (i, t2, t3);
```

This operation breaks *t1* into three pieces: the root, whose value it returns in *i*, and the left and right subtrees, whose values it returns in *t2* and *t3*, respectively. You may make this call only if *t1* is not empty.

Sample Traces

Statement	Object Values
	<div><p>t1 = </p><p>t2 = </p><p>t3 = </p><p>i = 21</p></div>
t1.Decompose (i, t2, t3);	
	<div><p>t1 = </p><p>t2 = </p><p>t3 = </p><p>i = -52</p></div>
t3.Decompose (i, t1, t2);	
	<div><p>t1 = </p><p>t2 = </p><p>t3 = </p><p>i = 13</p></div>
t1.Decompose (i, t2, t3);	
	<div><p>t1 = </p><p>t2 = </p><p>t3 = </p><p>i = 59</p></div>

Accessor

Formal Contract Specification

```
function Item& operator [] (
    preserves Accessor_Position& current
) is_abstract;
/*!
    requires
        self != empty_tree
    ensures
        there exists left, right: binary tree of Item
            (self = compose (self[current], left, right))
!*/
```

Informal Description

A call to the accessor operator [] has the form of an expression:

```
... t1[current] ...;
```

This expression acts as the name of an object of type *Item* whose value is the item at the root of *t1*. You may make this call only if *t1* is not empty.

The special word *current* is the only thing that can appear between [] in the accessor. It always refers to the single item at the root of the tree, which is the one that would be separated from the two subtrees if you called the *Decompose* operation at the point where you are using the accessor operator. You may use *t1[current]* wherever any other object of type *Item* may appear.

The accessor operator [], like all functions, preserves its arguments. But it is important to realize that the accessor expression *t1[current]* does not simply denote the value of the root item in *t1*, it acts as the name of an object of type *Item* which you may consider to lie at the root of *t1*. This means that not only may you use the expression *t1[current]* as a value of type *Item*; you may even change the value of the object called *t1[current]*—but remember that this also changes the value of *t1*.

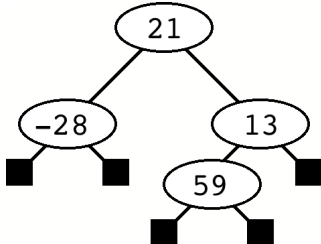
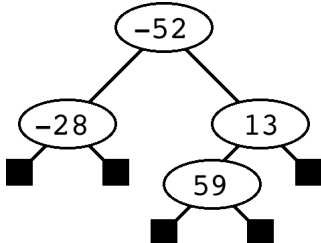
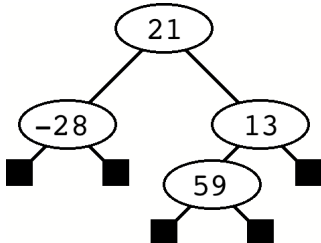
The sample traces help illustrate some important points regarding the convenience and danger of accessor expressions. The first and second sample statements show how an accessor expression such as *t1[current]* acts like an object of type *Item* whose value may be changed, depending on the statement in which the expression occurs. The third sample statement shows that *t1[current]* may appear as an argument in a call where an *Item* is required. This call involves other arguments (e.g., *t2*), but none of the other arguments may be *t1* or may involve an accessor expression for *t1* because this would violate the repeated argument rule. So, while you might be tempted to write something like:

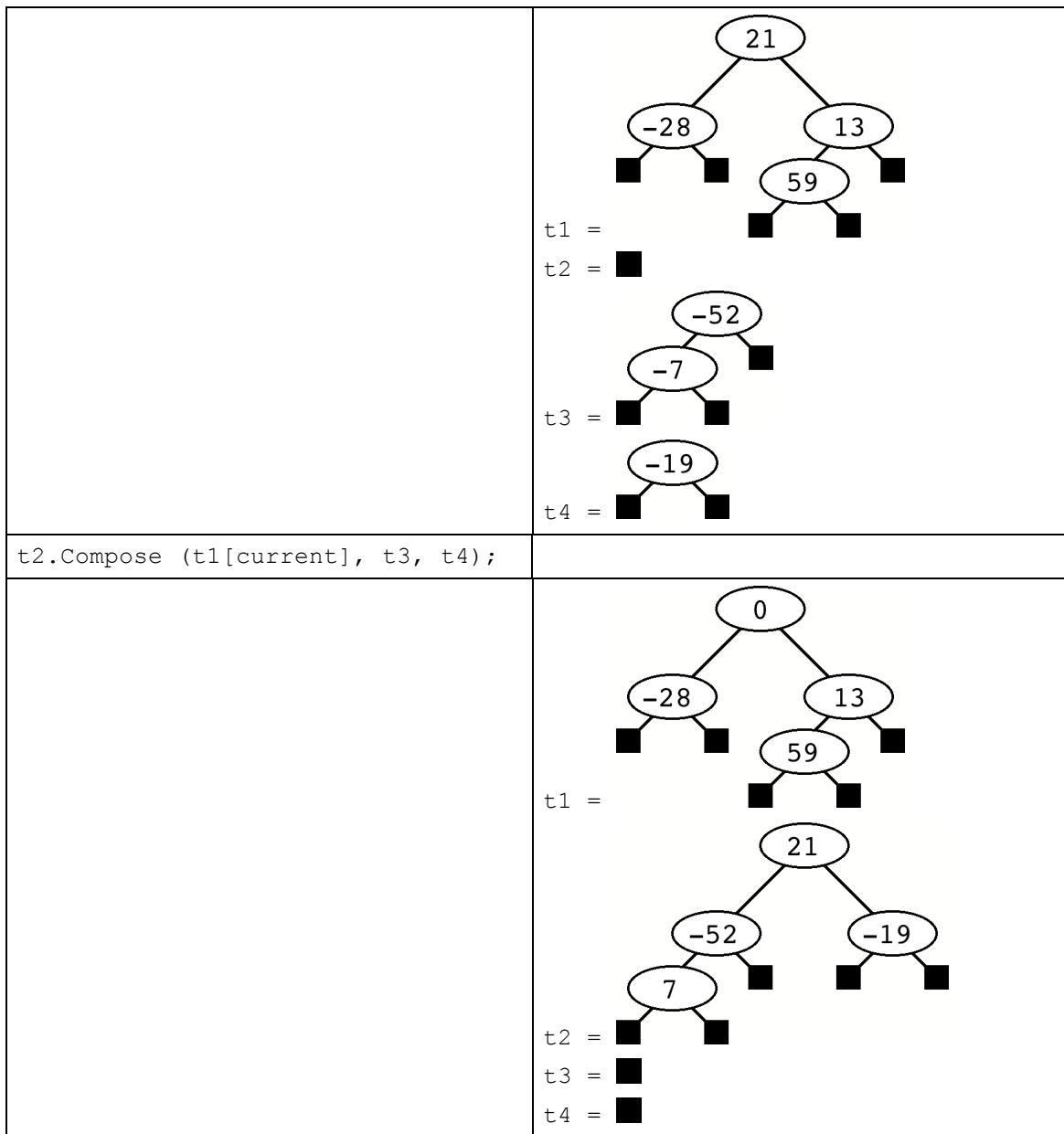
```
t1.Compose (t2[current], t2, t3);
```

this statement violates the repeated argument rule. Be aware that the C++ compiler will not report this as an error. You need to avoid the pitfall by personal discipline.

The last statement in the tracing table also illustrates that an accessor expression acts exactly like an object of type *Item*. Notice that *t1[current]*, which is being composed as the new root of *t2* in this statement, is consumed (its new value is 0 since *Item* is *Integer* here) because *Compose* consumes its arguments. So this statement changes both *t1* and *t2*.

Sample Traces

Statement	Object Values
	<div><pre>graph TD; 21((21)) --- -28((-28)); 21 --- 13((13)); -28 --- L1[]; -28 --- L2[]; 13 --- 59((59)); 13 --- L3[]; 59 --- L4[]; 59 --- L5[];</pre></div> <div>t1 = i = -52</div>
t1[current] &= i;	<div><pre>graph TD; -52((-52)) --- -28((-28)); -52 --- 13((13)); -28 --- L1[]; -28 --- L2[]; 13 --- 59((59)); 13 --- L3[]; 59 --- L4[]; 59 --- L5[];</pre></div> <div>t1 = i = 21</div>
t1[current] = i;	<div><pre>graph TD; 21((21)) --- -28((-28)); 21 --- 13((13)); -28 --- L1[]; -28 --- L2[]; 13 --- 59((59)); 13 --- L3[]; 59 --- L4[]; 59 --- L5[];</pre></div> <div>t1 = i = 21</div>
...	



Height

Formal Contract Specification

```

    /*!
      math operation HEIGHT (
        t: binary tree of Item
      ): integer satisfies
      if t = empty_tree
      then HEIGHT(t) = 0
      else there exists x: Item,
        left, right: binary tree of Item
        (t = compose (x, left, right) and
         HEIGHT(t) = 1 + max (HEIGHT(left),
                               HEIGHT(right)))

    !*/

    function Integer Height () is_abstract;
    /*!
      ensures
        Height = HEIGHT (self)
    !*/

```

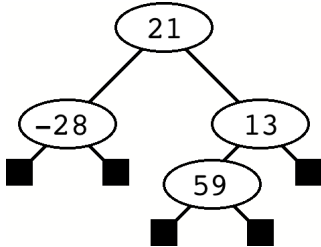
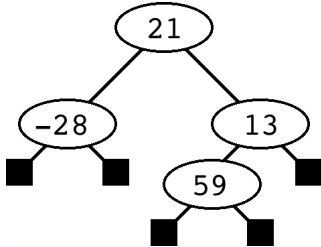


Informal Description

A call to the *Height* operation has the form of an expression:

```
... t1.Height () ...;
```

This operation returns an *Integer* value which is the height of *t1*, i.e., the number of items on the longest path from the root of *t1* to any leaf of *t1*.

Sample Traces

Statement	Object Values
	<div><pre>graph TD; 21((21)) --- -28((-28)); 21 --- 13((13)); -28 --- null1[]; -28 --- null2[]; 13 --- 59((59)); 13 --- null3[]; 59 --- null4[]; 59 --- null5[]</pre></div> <div>t1 = i = -68</div>
i = t1.Height ();	
	<div><pre>graph TD; 21((21)) --- -28((-28)); 21 --- 13((13)); -28 --- null1[]; -28 --- null2[]; 13 --- 59((59)); 13 --- null3[]; 59 --- null4[]; 59 --- null5[]</pre></div> <div>t1 = i = 3</div>
...	
	<div>t1 = </div> <div>i = 2</div>
i = t1.Height ();	
	<div>t1 = </div> <div>i = 0</div>

Size

Formal Contract Specification

```
function Integer Size () is_abstract;
/*!
    ensures
        Size = |self|
!*/
```

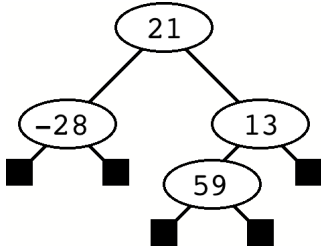
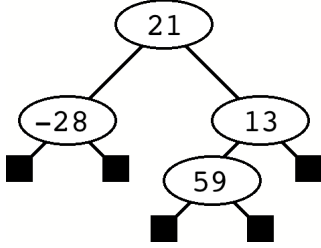
Informal Description

A call to the *Size* operation has the form of an expression:

```
... t1.Size () ...;
```

This operation returns an *Integer* value which is the number of items in the binary tree *t1*, i.e., *|t1|*.

Sample Traces

Statement	Object Values
	<div><pre>t1 = i = -68</pre></div>
i = t1.Size ();	
	<div><pre>t1 = i = 4</pre></div>
...	
	<pre>t1 = ■ i = 2</pre>
i = t1.Size ();	
	<pre>t1 = ■ i = 0</pre>

Id_Name_Table

Motivation, Applicability, and Indications for Use

The programming type *Id_Name_Table* allows you to build, query, and dismantle a set of ordered pairs (2-tuples) whose two components are an integer, called an *id*, and a string of characters, called a *name*. Each *id* and each *name* can appear at most once in any given *Id_Name_Table* object.

For example, suppose you need to keep a simple database of employee *ids* and their names (assuming all employees have unique names). To do this, you can create an *Id_Name_Table* object. To enter the name of an employee you simply add to the *Id_Name_Table* object the pair consisting of the employee's *id* and the employee's name. Later you can look up the employee's name given the employee's *id*, and vice versa. Other operations allow you to manipulate and observe properties of an *Id_Name_Table* object, such as its size, i.e., the number of pairs it holds.

Related Components

- *Partial_Map* — a type that is similar to *Id_Name_Table* except that (a) it is a template, so the client can select the types of both components of the ordered pairs; and (b) you can look up a pair based on the value of only the first of its two components, not both.

Component Family Members

Abstract Components

- *Id_Name_Table_Kernel* — the programming type of interest, with the operations below
 - *Add_Id_Name_Pair*
 - *Remove_Id_Name_Pair*
 - *Remove_Any_Pair*
 - *Look_Up_Name_Via_Id*
 - *Look_Up_Id_Via_Name*
 - *Id_Is_In_Table*
 - *Name_Is_In_Table*
 - *Size*

Concrete Components

- *Id_Name_Table_I_C* — This is a checking implementation of the abstract instance *Id_Name_Table_Kernel*. It is a concrete instance, i.e., you may declare objects of type *Id_Name_Table_I_C*. *Id_Name_Table_I_C* is a checking component, which means that it checks the preconditions of its operations. So, if you call any of the *Id_Name_Table_I_C* operations with a violated precondition then your program will immediately stop and report the violation.

To bring this component into the context you write:

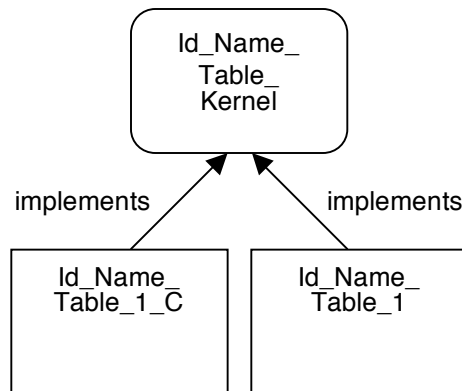
```
#include "CI/Id_Name_Table/1_C.h"
```

- *Id_Name_Table_1* — This is a non-checking implementation of the abstract instance *Id_Name_Table_Kernel*. It is a concrete instance, i.e., you may declare objects of type *Id_Name_Table_1*.

To bring this component into the context you write:

```
#include "CI/Id_Name_Table/1.h"
```

Component Coupling Diagram



Descriptions

Id_Name_Table_Kernel Type and Standard Operations

Formal Contract Specification

```

    /*!
      math subtype ID_NAME_TABLE_MODEL is finite set of (
        id: integer,
        name: string of character
      )
      exemplar m
      constraint
        for all id1, id2: integer,
          name1, name2: string of character
          where ((id1,name1) is in m and
                (id2,name2) is in m)
            (id1 = id2 iff name1 = name2)

      math definition ID_IS_DEFINED_IN (
        m: ID_NAME_TABLE_MODEL,
        id: integer
      ): boolean is
        there exists name: string of character
          ((id,name) is in m)

      math definition NAME_IS_DEFINED_IN (
        m: ID_NAME_TABLE_MODEL,
        name: string of character
      ): boolean is
        there exists id: integer
          ((id,name) is in m)

    !*/

    standard_abstract_operations (Id_Name_Table_Kernel);
    /*!
      Id_Name_Table_Kernel is modeled by ID_NAME_TABLE_MODEL
      initialization ensures
        self = empty_set
    !*/

```

Informal Description

The model for a *Id_Name_Table_Kernel* object is a finite set of ordered pairs, each of which is an integer and a string of characters. Each integer and each string of characters is unique among all the pairs in the set.

An *Id_Name_Table_Kernel* object is initially empty. An *Id_Name_Table_Kernel* object may be an arbitrarily large set. Like all Resolve/C++ types, *Id_Name_Table_Kernel* comes with operator `&=` and *Clear*.

Sample Traces

Statement	Object Values
	(t1 is not in scope)
object Id_Name_Table_1_C t1;	
	t1 = {}

Add_Id_Name_Pair

Formal Contract Specification

```

procedure Add_Id_Name_Pair (
    preserves Integer id,
    consumes Text& name
) is_abstract;
/*!
    requires
        not ID_IS_DEFINED_IN (self, id) and
        not NAME_IS_DEFINED_IN (self, name)
    ensures
        self = #self union {(id,#name)}
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Id_Name_Table_1_C t1, t2;
object Text name1, name2;
object Integer id1, id2, i;
object Boolean b;

```

A call to the *Add_Id_Name_Pair* operation has the form:

```
t1.Add_Id_Name_Pair (id1, name1);
```

This operation adds the pair whose value is *(id1,name1)* to the set *t1*. You may make this call only if *t1* does not contain already a pair whose id component is *id1* and if *t1* does not contain already a pair whose name component is *name1*.

Sample Traces

Statement	Object Values
	t1 = {(3,"Santa Fe"), (8,"San Francisco")} id1 = 2 name1 = "Columbus"
t1.Add_Id_Name_Pair (id1, name1);	
	t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} id1 = 2 name1 = ""

Remove_Id_Name_Pair

Formal Contract Specification

```
procedure Remove_Id_Name_Pair (  
    preserves Integer id,  
    preserves Text name,  
    produces Integer& id_copy,  
    produces Text& name_copy  
    ) is_abstract;  
/*!  
    requires  
        (id,name) is in self  
    ensures  
        id_copy = id and  
        name_copy = name and  
        self = #self - {(id,name)}  
!*/
```

Informal Description

A call to the *Remove_Id_Name_Pair* operation has the form:

```
t1.Remove_Id_Name_Pair (id1, name1, id2, name2);
```

This operation removes from *t1* the pair whose id component is *id1* and whose name component is *name1* and returns its value in (*id2*, *name2*). You may make this call only if there is a pair in *t1* whose id component is *id1* and whose name component is *name1*.

Sample Traces

Statement	Object Values
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} id1 = 8 name1 = "San Francisco" id2 = 10047 name2 = "anything" </pre>
<code>t1.Remove_Id_Name_Pair (id1, name1, id2, name2);</code>	
	<pre> t1 = {(3,"Santa Fe"), (2,"Columbus")} id1 = 8 name1 = "San Francisco" id2 = 8 name2 = "San Francisco" </pre>
...	
	<pre> t1 = {(3,"Santa Fe"), (2,"Columbus")} id2 = -133 name2 = "doesn't matter" </pre>
<code>t1.Remove_Id_Name_Pair (2, "Columbus", id2, name2);</code>	
	<pre> t1 = {(3,"Santa Fe")} id2 = 2 name2 = "Columbus" </pre>

Remove_Any_Pair

Formal Contract Specification

```
procedure Remove_Any_Pair (  
    produces Integer& id,  
    produces Text& name  
    ) is_abstract;  
/*!  
    requires  
        self /= empty_set  
    ensures  
        (id,name) is in #self and  
        self = #self - {(id,name)}  
!*/
```

Informal Description

A call to the *Remove_Any_Pair* operation has the form:

```
t1.Remove_Any_Pair (id1, name1);
```

This operation removes some pair from *t1* — any pair — and returns its value in *(id1,name1)*. You may make this call only if *t1* is not empty.

Notice that this operation may remove a different pair from *t1* if called twice in identical situations, as illustrated in the sample tracing table.

Sample Traces

Statement	Object Values
	<pre>t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} idl = 17 name1 = "don't care"</pre>
<code>t1.Remove_Any_Pair (idl, name1);</code>	
	<pre>t1 = {(8,"San Francisco"), (2,"Columbus")} idl = 3 name1 = "Santa Fe"</pre>
<code>...</code>	
	<pre>t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} idl = 17 name1 = "don't care"</pre>
<code>t1.Remove_Any_Pair (idl, name1);</code>	
	<pre>t1 = {(3,"Santa Fe"), (8,"San Francisco")} idl = 2 name1 = "Columbus"</pre>

Look_Up_Name_Via_Id

Formal Contract Specification

```
procedure Look_Up_Name_Via_Id (  
    preserves Integer id,  
    produces Text& name  
    ) is_abstract;  
/*!  
    preserves self  
    requires  
        ID_IS_DEFINED_IN (self, id)  
    ensures  
        (id,name) is in self  
!*/
```

Informal Description

A call to the *Look_Up_Name_Via_Id* operation has the form:

```
t1.Look_Up_Name_Via_Id (id1, name1);
```

This operation finds the pair in *t1* with id component *id1* and returns a copy of the name associated with *id1* as the value of *name1*. You may make this call only if there is a pair in *t1* whose id component equals *id1*.

Sample Traces

Statement	Object Values
	<pre>t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} id1 = 2 name1 = "don't care"</pre>
<code>t1.Look_Up_Name_Via_Id (id1, name1);</code>	
	<pre>t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} id1 = 2 name1 = "Columbus"</pre>
<code>...</code>	
	<pre>t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} name1 = "who cares"</pre>
<code>t1.Look_Up_Name_Via_Id (8, name1);</code>	
	<pre>t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} name1 = "San Francisco"</pre>

Look_Up_Id_Via_Name

Formal Contract Specification

```
procedure Look_Up_Id_Via_Name (  
    produces Integer& id,  
    preserves Text name  
) is_abstract;  
/*!  
    preserves self  
    requires  
        NAME_IS_DEFINED_IN (self, name)  
    ensures  
        (id,name) is in self  
!*/
```

Informal Description

A call to the *Look_Up_Id_Via_Name* operation has the form:

```
t1.Look_Up_Id_Via_Name (idl, name1);
```

This operation finds the pair in *t1* with name *name1* and returns a copy of the id associated with *name1* as the value of *idl*. You may make this call only if there is a pair in *t1* whose name component equals *name1*.

Sample Traces

Statement	Object Values
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} idl = 27702 name1 = "Santa Fe" </pre>
<code>t1.Look_Up_Id_Via_Name (idl, name1);</code>	
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} idl = 3 name1 = "Santa Fe" </pre>
<code>...</code>	
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} idl = 49 </pre>
<code>t1.Look_Up_Id_Via_Name (idl, "Columbus");</code>	
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} idl = 2 </pre>

Id_Is_In_Table

Formal Contract Specification

```

function Boolean Id_Is_In_Table (
    preserves Integer id
) is_abstract;
/*!
    ensures
        Id_Is_In_Table = ID_IS_DEFINED_IN (self, id)
!*/

```

Informal Description

A call to the *Id_Is_In_Table* operation has the form of an expression:

```
... t1.Id_Is_In_Table (idl) ...;
```

This operation returns true if there is a pair in *t1* with an id component of *idl* and returns false otherwise.

Sample Traces

Statement	Object Values
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} b = true idl = 99 </pre>
<code>b = t1.Id_Is_In_Table (idl);</code>	
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} b = false idl = 99 </pre>
<code>...</code>	
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} b = false </pre>
<code>b = t1.Id_Is_In_Table (2);</code>	
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} b = true </pre>

Name_Is_In_Table

Formal Contract Specification

```

function Boolean Name_Is_In_Table (
    preserves Text name
) is_abstract;
/*!
    ensures
        Name_Is_In_Table =
            NAME_IS_DEFINED_IN (self, name)
!*/

```

Informal Description

A call to the *Name_Is_In_Table* operation has the form of an expression:

```
... t1.Name_Is_In_Table (name1) ...;
```

This operation returns true if there is a pair in *t1* with a name component of *name1* and returns false otherwise.

Sample Traces

Statement	Object Values
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} b = true name1 = "Cincinnati" </pre>
<code>b = t1.Name_Is_In_Table (name1);</code>	
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} b = false name1 = "Cincinnati" </pre>
<code>...</code>	
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} b = false </pre>
<code>b = t1.Name_Is_In_Table ("Columbus");</code>	
	<pre> t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} b = true </pre>

Size

Formal Contract Specification

```

function Integer Size () is_abstract;
  /*!
    ensures
      Size = |self|
  !*/

```

Informal Description

A call to the *Size* operation has the form of an expression:

```
... t1.Size () ...;
```

This operation returns an *Integer* value which is the size of the set *t1*, i.e., *|t1|*.

Sample Traces

Statement	Object Values
	t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} i = -68
i = t1.Size ();	
	t1 = {(3,"Santa Fe"), (8,"San Francisco"), (2,"Columbus")} i = 3

List

Motivation, Applicability, and Indications for Use

The programming type *List* allows you to collect and then process items in sequential order. But unlike its close cousins *Queue* and *Stack* it allows you to add and remove items not just from the ends of a string of items but from anywhere in the middle as well. So wherever you might use a *Queue* or *Stack* object but want a bit more flexibility in terms of which items you may access, then consider a *List* object.

On the other hand, a *List* object does not provide as much flexibility as a *Sequence* object because you cannot directly access an item based on its position in a *List* object. The relative advantage of using a *List* object is that there is a very efficient implementation of the *List* kernel operations (all can run in constant time) which cannot be matched by any implementation of *Sequence*.

Related Components

- *Queue* — a type that allows you to add items in sequence one at a time, but when you remove an item it is the first one that was put in
- *Stack* — a type that allows you to add items in sequence one at a time, but when you remove an item it is the last (most recent) one that was put in
- *Sequence* — a type that allows you to add or remove items in sequence one at a time, accessing an item at any time by “random access” based on its current position

Component Family Members

Abstract Components

- *List_Kernel* — the programming type of interest, with the operations below
 - *Move_To_Start*
 - *Move_To_Finish*
 - *Advance*
 - *Add_Right*
 - *Remove_Right*
 - *Accessor*
 - *Left_Length*
 - *Right_Length*

Concrete Components

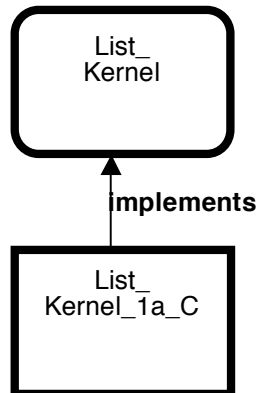
- *List_Kernel_1a_C* — This is a checking implementation of *List_Kernel* in which the execution time for each of the operations constructor, *Move_To_Start*, *Move_To_Finish*, *Advance*, *Add_Right*, *Remove_Right*, the accessor, *Left_Length*, and *Right_Length* is constant, while the execution time for the destructor is proportional to the sum of the lengths

of the two strings in the tuple. All objects of this type have the interface of *List_Kernel*, with the concrete template name *List_Kernel_1a_C* substituted for the abstract template name *List_Kernel*.

To bring this component into the context you write:

```
#include "CT/List/Kernel_1a_C.h"
```

Component Coupling Diagram



Descriptions

List_Kernel Type and Standard Operations

Formal Contract Specification

```

    /*!
      math subtype LIST_MODEL is (
        left: string of Item
        right: string of Item
      )
    !*/

    standard_abstract_operations (List_Kernel);
    /*!
      List_Kernel is modeled by LIST_MODEL
      initialization ensures
        self = (empty_string, empty_string)
    !*/

```

Informal Description

The model for *List_Kernel* is a pair of strings of items, each of which is initially empty. The concatenation of the two strings contains all the items in a *List_Kernel* object. These strings are called “left” and “right” to suggest that you should imagine a separator, called “the fence”, to lie between the two strings: the first string is to the left of the fence and the other string is to the right of the fence.

A *List_Kernel* object may be arbitrarily large, i.e., both strings may be arbitrarily long. Like all Resolve/C++ types, *List_Kernel* comes with operator `&=` and *Clear*.

Note — The sample traces for this and the other operation descriptions refer to the type *List_Of_Integer*, which is the result of the following instantiation:

```

concrete_instance
class List_Of_Integer :
  instantiates
    List_Kernel_1a_C <Integer>
{};

```

Prototypical objects of this type in the examples that follow have names beginning with “s”, not “l”, because it is difficult to visually distinguish the letter “l” from the numeral “1”.

Sample Traces

Statement	Object Values
	(s1 is not in scope)
object List_Of_Integer s1;	
	s1 = (< >, < >)

Move_To_Start

Formal Contract Specification

```

procedure Move_To_Start () is_abstract;
  /*!
    ensures
      self = (empty_string, #self.left * #self.right)
  !*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object List_Of_Integer s1, s2;
object Integer i;

```

A call to the *Move_To_Start* operation has the form:

```

s1.Move_To_Start ();

```

This operation moves the fence to the “start” (left end) of *s1* but does not otherwise affect the items of *s1*.

The left end is called the start of a *List* object because, ordinarily, sequential processing of the items proceeds from left to right using the *Advance* operation (see below). The *Move_To_Start* operation takes you directly to the start by skipping backward over all items in *s1.left*, thereby preparing for a typical sequential processing activity.

Sample Traces

Statement	Object Values
	s1 = (< 58, 12 >, < 9, 90 >)
s1.Move_To_Start ();	
	s1 = (< >, < 58, 12, 9, 90 >)

Move_To_Finish

Formal Contract Specification

```

procedure Move_To_Finish () is_abstract;
  /*!
    ensures
      self = (#self.left * #self.right, empty_string)
  !*/

```

Informal Description

A call to the *Move_To_Finish* operation has the form:

```
s1.Move_To_Finish ();
```

This operation moves the fence to the “finish” (right end) of *s1* but does not otherwise affect the items of *s1*.

The right end is called the finish of a *List* object because, ordinarily, sequential processing of the items proceeds from left to right using the *Advance* operation (see below). The *Move_To_Finish* operation takes you directly to the finish by skipping over all items in *s1.right*.

Sample Traces

Statement	Object Values
	s1 = (< 58, 12 >, < 9, 90 >)
s1.Move_To_Finish ();	
	s1 = (< 58, 12, 9, 90 >, < >)

Advance

Formal Contract Specification

```

procedure Advance () is_abstract;
/*!
    requires
        self.right /= empty_string
    ensures
        self.left * self.right = #self.left * #self.right  and
        |self.left| = |#self.left| + 1
!*/

```

Informal Description

A call to the *Advance* operation has the form:

```
s1.Advance ();
```

This operation moves the fence forward (toward the right) by one position in *s1* but does not otherwise affect the items of *s1*. You may make this call only if *s1.right* is not empty.

Sample Traces

Statement	Object Values
	s1 = (< 58, 12 >, < 9, 90 >)
s1.Advance ();	
	s1 = (< 58, 12, 9 >, < 90 >)

Add_Right

Formal Contract Specification

```

procedure Add_Right (
    consumes Item& x
) is_abstract;
/*!
    ensures
        self = (#self.left, <#x> * #self.right)
!*/

```

Informal Description

A call to the *Add_Right* operation has the form:

```
s1.Add_Right (i);
```

This operation adds the item whose value is *i* just to the right of the fence in *s1*, i.e., it adds it at the left end of *s1.right*.

Sample Traces

Statement	Object Values
	s1 = (< 58, 12 >, < 9, 90 >) i = 13
s1.Add_Right (i);	
	s1 = (< 58, 12 >, < 13, 9, 90 >) i = 0

Remove_Right

Formal Contract Specification

```

procedure Remove_Right (
    produces Item& x
) is_abstract;
/*!
    requires
        self.right /= empty_string
    ensures
        #self = (self.left, <x> * self.right)
!*/

```

Informal Description

A call to the *Remove_Right* operation has the form:

```
s1.Remove_Right (i);
```

This operation removes and returns in *i* the item just to the right of the fence in *s1*, i.e., the item at the left end of *s1.right*. You may make this call only if *s1.right* is not empty.

Sample Traces

Statement	Object Values
	s1 = (< 58, 12 >, < 13, 9, 90 >) i = -67
s1.Remove_Right (i);	
	s1 = (< 58, 12 >, < 9, 90 >) i = 13

Accessor

Formal Contract Specification

```
function Item& operator [] (
    preserves Accessor_Position& current
) is_abstract;
/*!
    requires
        self.right != empty_string
    ensures
        there exists a: string of Item
            (self.right = <self[current]> * a)
!*/
```

Informal Description

A call to the accessor operator [] has the form of an expression:

```
... s1[current] ...;
```

This expression acts as the name of an object of type *Item* whose value is the item at the left end of *s1.right*. You may make this call only if *s1.right* is not empty.

The special word *current* is the only thing that can appear between [] in the accessor for type *List*. It always refers to the single item that you can access directly: the one just to the right of the fence, which is the one that would be removed if you called the *Remove_Right* operation at the point where you are using the accessor operator. You may use *s1[current]* wherever any other object of type *Item* may appear.

The accessor operator [], like all functions, preserves its arguments. But it is important to realize that the accessor expression *s1[current]* does not simply denote the value of the current item in *s1*, it acts as the name of an object of type *Item* which you may consider to lie at the left end of the string *s1.right*. This means that not only may you use the expression *s1[current]* as a value of type *Item*; you may even change the value of the object called *s1[current]*—but remember that this also changes the value of *s1*.

The sample traces help illustrate some important points regarding the convenience and danger of accessor expressions. The first and second sample statements show how an accessor expression such as *s1[current]* acts like an object of type *Item* whose value may be changed, depending on the statement in which the expression occurs. The third sample statement shows that *s1[current]* may appear as an argument in a call where an *Item* is required. This call involves other arguments (e.g., *s2*), but none of the other arguments may be *s1* or may involve an accessor expression for *s1* because this would violate the repeated argument rule. So, while you might be tempted to write something like:

```
s1.Add_Right (s1[current]);
```

this statement violates the repeated argument rule. Be aware that the C++ compiler will not report this as an error. You need to avoid the pitfall by personal discipline.

The last statement in the tracing table also illustrates that an accessor expression acts exactly like an object of type *Item*. Notice that *s1[current]*, which is being added to *s2* in this statement, is consumed (its new value is 0 since *Item* is *Integer* here) because *Add_Right* consumes its argument. So this statement changes both *s1* and *s2*.

Sample Traces

Statement	Object Values
	s1 = (< 92, 6 >, < 18, 53 >) i = 37
s1[current] &= i;	
	s1 = (< 92, 6 >, < 37, 53 >) i = 18
s1[current] = i;	
	s1 = (< 92, 6 >, < 18, 53 >) i = 18
...	
	s1 = (< 92, 6 >, < 18, 53 >) s2 = (< 47 >, < -3 >)
s2.Add_Right (s1[current]);	
	s1 = (< 92, 6 >, < 0, 53 >) s2 = (< 47 >, < 18, -3 >)

Left_Length

Formal Contract Specification

```

function Integer Left_Length () is_abstract;
/*!
    ensures
        Left_Length = |self.left|
!*/

```

Informal Description

A call to the *Left_Length* operation has the form of an expression:

```
... s1.Left_Length () ...;
```

This operation returns an *Integer* value which is the length of the string *s1.left*, i.e., *ls1.leftl*.

Sample Traces

Statement	Object Values
	s1 = (< 58, 12 >, < 13, 9, 90 >) i = -37
i = s1.Left_Length ();	
	s1 = (< 58, 12 >, < 13, 9, 90 >) i = 2

Right_Length

Formal Contract Specification

```

function Integer Right_Length () is_abstract;
/*!
    ensures
        Right_Length = |self.right|
!*/

```

Informal Description

A call to the *Right_Length* operation has the form of an expression:

```
... s1.Right_Length () ...;
```

This operation returns an *Integer* value which is the length of the string *s1.right*, i.e., *ls1.rightl*.

Sample Traces

Statement	Object Values
	s1 = (< 58, 12 >, < 13, 9, 90 >) i = -37
i = s1.Right_Length ();	
	s1 = (< 58, 12 >, < 13, 9, 90 >) i = 3

Natural

Motivation, Applicability, and Indications for Use

RESOLVE/C++ includes the built-in programming type *Integer*, whose mathematical model is a mathematical integer lying between two bounds. On most modern computer systems these are -2^{31} and $2^{31}-1$ ($-2,147,483,648$ and $+2,147,483,647$).

These bounds make the potential *Integer* values “big enough” for most routine purposes. However, there are many situations where (non-negative) integer-valued objects are appropriate, but where two billion or so is not the largest value ever encountered. For example, the amount of a bank transaction in U.S. currency might be recorded in an *Integer* object: the number of cents being transferred. But then a figure like the U.S. national debt or even a plausible inter-bank electronic fund transfer is too large to handle. If you need to accommodate a larger (arbitrarily large) non-negative number, then you should use a *Natural* object, whose mathematical model is an integer which is bounded below by zero but which is, for practical purposes, not bounded above.

Related Components

- *Integer* — the type to use if the numbers in your calculations are always small

Component Family Members

Abstract Components¹

- *Natural_Kernel* — the programming type of interest, with the operations below
 - *Multiply_By_Radix*
 - *Divide_By_Radix*
 - *Discrete_Log*
 - *Radix*
- *Natural_Arithmetic* — a bundle combining *Natural_Kernel* and the operations below
 - *Convert_From_Integer*
 - *Increment*
 - *Decrement*
 - *Compare*
 - *Add*
 - *Subtract*

¹ This list of abstract components is not entirely accurate for the *Natural* family, which includes a number of templates (for technical reasons). But it is equivalent from the standpoint of a client using any of the concrete components listed next.

- *Multiply*
- *Divide*
- *Natural_Number* — a bundle combining *Natural_Arithmetic* and the operations below
 - *Copy_To*
 - *Get_From* (>>)
 - *Put_To* (<<)

Concrete Components

- *Natural_1_C* — This is a checking implementation of the abstract instance *Natural_Kernel*, so it is not very powerful; you probably want *Natural_Number_1_C* instead. It is a concrete instance, i.e., you may declare objects of type *Natural_1_C*. *Natural_1_C* is a checking component, which means that it checks the preconditions of its operations. So, if you call any of the *Natural_1_C* operations with a violated precondition then your program will immediately stop and report the violation.

To bring this component into the context you write:

```
#include "CI/Natural/1_C.h"
```

- *Natural_1* — This is a non-checking implementation of the abstract instance *Natural_Kernel*, so it is not very powerful; you probably want *Natural_Number_1* instead. It is a concrete instance, i.e., you may declare objects of type *Natural_1*.

To bring this component into the context you write:

```
#include "CI/Natural/1.h"
```

- *Natural_Number_1_C* — This is a checking implementation of the abstract instance *Natural_Number*. It is a concrete instance, i.e., you may declare objects of type *Natural_Number_1_C*. *Natural_Number_1_C* is a checking component, which means that it checks the preconditions of its operations. So, if you call any of the *Natural_Number_1_C* operations with a violated precondition then your program will immediately stop and report the violation.

To bring this component into the context you write:

```
#include "CI/Natural/Number_1_C.h"
```

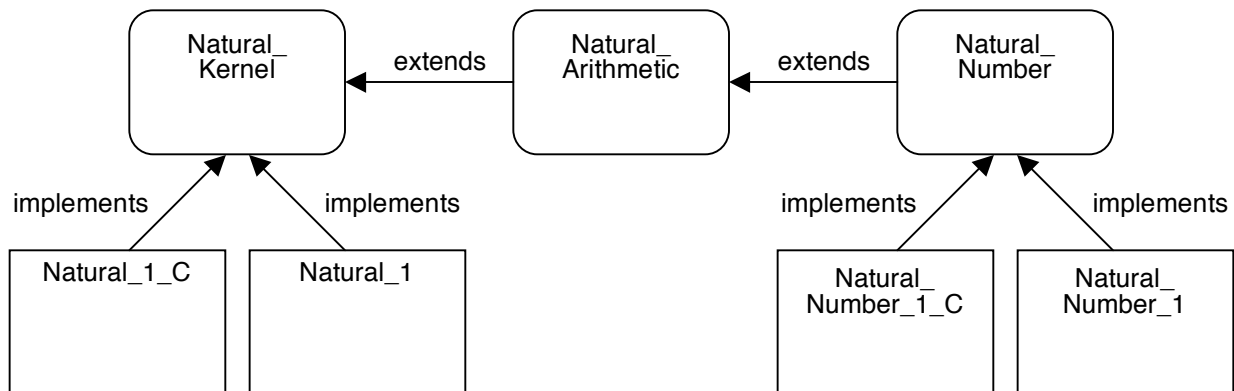
- *Natural_Number_1* — This is a non-checking implementation of the abstract instance *Natural_Number*. It is a concrete instance, i.e., you may declare objects of type *Natural_Number_1*.

To bring this component into the context you write:

```
#include "CI/Natural/Number_1.h"
```

Note — In all of the above concrete components, the implementation-supplied value of the constant *RADIX* is 10.

Component Coupling Diagram²



² This CCD is not the actual CCD for the *Natural* family, which includes a number of templates (for technical reasons). But it is equivalent to the true CCD from the standpoint of a client using any of the concrete components.

Descriptions

Natural_Kernel Type and Standard Operations

Formal Contract Specification

```

    /*!
      math subtype NATURAL_MODEL is integer
      exemplar n
      constraint
        n >= 0

      math definition RADIX: integer satisfies restriction
        RADIX >= 2
    !*/

    standard_abstract_operations (Natural_Kernel);
    /*!
      Natural_Kernel is modeled by NATURAL_MODEL
      initialization ensures
        self = 0
    !*/

```

Informal Description

The model for *Natural_Kernel* is a mathematical integer which is initially zero. A *Natural_Kernel* object may have an arbitrarily large (but always non-negative) value. Like all Resolve/C++ types, *Natural_Kernel* comes with operator `&=` and *Clear*.

Sample Traces

Statement	Object Values
	(n1 is not in scope)
object Natural_Number_1_C n1;	
	n1 = 0

Multiply_By_Radix

Formal Contract Specification

```

procedure Multiply_By_Radix (
    preserves Integer k
) is_abstract;
/*!
    requires
        0 <= k < RADIX
    ensures
        self = #self * RADIX + k
!*/

```

Informal Description

Note — This and the other operation descriptions that follow refer to these objects in examples:

```

object Natural_Number_1_C n1, n2, n3;
object Integer i;
object Character_IStream input;
object Character_OStream output;

```

A call to the *Multiply_By_Radix* operation has the form:

```

n1.Multiply_By_Radix (i);

```

This operation multiplies *n1* by *RADIX*, and then adds *i* to it. You may make this call only if *i* could be a digit in base *RADIX*, i.e., $0 \leq i < \text{RADIX}$.

This operation and three others — *Divide_By_Radix*, *Discrete_Log*, and *Radix* — allow you, as a client, to think about and manipulate a *Natural_Kernel* object's value through a view of its radix representation in the *base RADIX*.

The key to understanding *Natural_Kernel* is to realize that there is a difference among the following concepts:

a *number* — an abstract idea,

the way we normally think about that number to perform mathematical calculations, which involves the radix representation discussed below, and

the way we write that number so we can communicate with each other, also discussed below.

For instance, here are some ways to think about, and write, the single abstract natural number we usually call “42”:

$4 \cdot 10^1 + 2 \cdot 10^0$, in *decimal* (base 10), ordinarily written 42 because 10 is the default base in normal mathematical usage

$5 \cdot 8^1 + 2 \cdot 8^0$, in *octal* (base 8), often written 52_8

$1 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 0 \cdot 3^0$, in *ternary* (base 3), often written 1120_3

$1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$, in *binary* (base 2), often written 101010_2

The important point is that the same abstract number, i.e., the one we usually call “42”, might be represented internally in the computer, and displayed externally, in many different ways. The abstract number is characterized by its mathematical properties; for computational purposes we

might choose to think of it in binary as $1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$; for communication purposes we might choose to write it in decimal as 42.

These examples suggest that the base must be a small integer. Actually, it can be any value greater than or equal to 2; an efficient implementation of *Natural_Kernel* might use a number in the thousands or more as its value of its base, *RADIX*. For instance, here is another way to think of and write the number we call “42”; notice the base is greater than 10:

$2 \cdot 16^1 + 10 \cdot 16^0$, in hexadecimal (base 16), often written $2A_{16}$

The digits³ of a number viewed in this way always lie between 0 and *RADIX*–1, inclusive. For *RADIX* > 10, there are not enough numerals to use for some of the digits, so the letters A, B, etc., are customarily introduced to stand for 10, 11, etc., when writing a number in a base other than decimal.

Different implementations of *Multiply_By_Radix* might, therefore, use different values for *RADIX*. The concrete instances *Natural_1_C*, *Natural_1*, *Natural_Number_1_C*, and *Natural_Number_1* all use *RADIX* = 10, which conveniently matches the way we ordinarily write numbers. This helps in understanding the sample trace tables that follow.

Stated in these terms, another way to explain the call:

```
n1.Multiply_By_Radix (i);
```

is to say that it appends *i* as the new low-order (rightmost) digit of *n1*.

Sample Traces

Statement	Object Values
	n1 = 45142398 i = 1
n1.Multiply_By_Radix (i);	
	n1 = 451423981 i = 1
n1.Multiply_By_Radix (0);	
	n1 = 4514239810 i = 1

³ Technically, the word “digit” applies only when *RADIX* = 10. For *RADIX* = 2 a digit usually is called a “bit” (a contraction of “binary digit”). However, it is customary to use “digit” regardless of the value of *RADIX* — especially when *RADIX* is a fixed but unknown constant, as it is here.

Divide_By_Radix

Formal Contract Specification

```

procedure Divide_By_Radix (
    produces Integer& k
) is_abstract;
/*!
    ensures
        #self = self * RADIX + k  and
        0 <= k < RADIX
!*/

```

Informal Description

A call to the *Divide_By_Radix* operation has the form:

```
n1.Divide_By_Radix (i);
```

This operation divides *n1* by *RADIX* and returns with the remainder in *i*.

Stated in terms of the radix representation of *n1* (see *Multiply_By_Radix* above), another way to explain the call:

```
n1.Divide_By_Radix (i);
```

is to say that it removes the low-order (rightmost) digit of *n1* and returns with that digit as the value of *i*.

Sample Traces

Statement	Object Values
	n1 = 45142398 i = 1
n1.Divide_By_Radix (i);	
	n1 = 4514239 i = 8

Discrete_Log

Formal Contract Specification

```

function Integer Discrete_Log () is_abstract;
/*!
    ensures
        if self = 0
        then Discrete_Log = 0
        else RADIX ^ (Discrete_Log - 1) <=
            self < RADIX ^ (Discrete_Log)
!*/

```

Informal Description

A call to the *Discrete_Log* operation has the form of an expression:

```
... n1.Discrete_Log () ...;
```

This operation returns an *Integer* value equal to the discrete logarithm of *n1* in base *RADIX*.

What is the discrete logarithm? Stated in terms of the radix representation of *n1* (see *Multiply_By_Radix* above), another way to explain the call:

```
... n1.Discrete_Log () ...;
```

is to say that it returns the number of digits needed to write *n1* in base *RADIX* — except that it always returns 0 (not 1) in case *n1* = 0.

Sample Traces

Statement	Object Values
	n1 = 45142398 i = 1
i = n1.Discrete_Log ();	
	n1 = 45142398 i = 8
...	
	n1 = 0 i = 8
i = n1.Discrete_Log ();	
	n1 = 0 i = 0

Radix

Formal Contract Specification

```

function Integer Radix () is_abstract;
  /*!
    ensures
      radix = RADIX
  !*/

```

Informal Description

A call to the *Radix* operation has the form of an expression:

```
... n1.Radix () ...;
```

This operation returns an *Integer* value which is *n1*'s *RADIX*.

In order for this operation to be useful, “*n1*'s *RADIX*” must have the same value here as it does for the other operations that involve *RADIX* (i.e., *Multiply_By_Radix*, *Divide_By_Radix*, and *Discrete_Log*). This property holds for the concrete instance *Natural_Number_1_C* used in the tracing table examples, because it implements all of these operations with *RADIX* = 10.

Sample Traces

Statement	Object Values
	n1 = 45142398 i = 46
i = n1.Radix ();	
	n1 = 45142398 i = 10

Convert_From_Integer

Formal Contract Specification

```

procedure Convert_From_Integer (
    preserves Integer k
) is_abstract;
/*!
    produces self
    requires
        k >= 0
    ensures
        self = k
!*/

```

Informal Description

A call to the *Convert_From_Integer* operation has the form:

```
n1.Convert_From_Integer (i);
```

This operation sets *n1* equal to *i*. You may make this call only if *i* is non-negative.

Sample Traces

Statement	Object Values
	n1 = 13 i = 497
n1.Convert_From_Integer (i);	
	n1 = 497 i = 497
n1.Convert_From_Integer (78);	
	n1 = 78 i = 497

Increment

Formal Contract Specification

```
procedure Increment () is_abstract;  
  /*!  
    ensures  
      self = #self + 1  
  !*/
```

Informal Description

A call to the *Increment* operation has the form:

```
n1.Increment ();
```

This operation adds one to *n1*.

Sample Traces

Statement	Object Values
	n1 = 345475543524
n1.Increment();	
	n1 = 345475543525

Decrement

Formal Contract Specification

```

procedure Decrement () is_abstract;
/*!
    requires
        self > 0
    ensures
        self = #self - 1
!*/

```

Informal Description

A call to the *Decrement* operation has the form:

```
n1.Decrement ();
```

This operation subtracts one from *n1*. You may make this call only if *n1* is strictly positive, since *Natural_Type* objects always have non-negative values.

Sample Traces

Statement	Object Values
	n1 = 68935645902
n1.Decrement ();	
	n1 = 68935645901

Compare

Formal Contract Specification

```

function Integer Compare (
    preserves Natural_Number& n
) is_abstract;
/*!
    ensures
        if self < n
        then Compare = -1
        else if self = n
        then Compare = 0
        else Compare = 1
!*/

```

Informal Description

A call to the *Compare* operation has the form of an expression:

```
... n1.Compare (n2) ...;
```

This operation returns an *Integer* value of -1 if $n1 < n2$; 0 if $n1 = n2$; and $+1$ if $n1 > n2$.

Sample Traces

Statement	Object Values
	n1 = 45142398 n2 = 1996 i = 13
i = n1.Compare (n2);	
	n1 = 45142398 n2 = 1996 i = 1
i = n2.Compare (n1);	
	n1 = 45142398 n2 = 1996 i = -1
...	
	n1 = 1996 n2 = 1996 i = 34
i = n1.Compare (n2);	
	n1 = 1996 n2 = 1996 i = 0

Add

Formal Contract Specification

```

procedure Add (
    preserves Natural_Number& n
) is_abstract;
/*!
    ensures
        self = #self + n
!*/

```

Informal Description

A call to the *Add* operation has the form:

```
n1.Add (n2);
```

This operation adds *n2* to *n1*.

Sample Traces

Statement	Object Values
	n1 = 26 n2 = 952
n1.Add (n2);	
	n1 = 978 n2 = 952

Subtract

Formal Contract Specification

```

procedure Subtract (
    preserves Natural_Number& n
) is_abstract;
/*!
    requires
        self >= n
    ensures
        self = #self - n
!*/

```

Informal Description

A call to the *Subtract* operation has the form:

```
n1.Subtract (n2);
```

This operation subtracts $n2$ from $n1$. You may make this call only if $n1$ is at least as large as $n2$.

Sample Traces

Statement	Object Values
	n1 = 101 n2 = 37
n1.Subtract (n2);	
	n1 = 64 n2 = 37

Multiply

Formal Contract Specification

```

procedure Multiply (
    preserves Natural_Number& n
) is_abstract;
/*!
    ensures
        self = #self * n
!*/

```

Informal Description

A call to the *Multiply* operation has the form:

```
n1.Multiply (n2);
```

This operation multiplies *n1* by *n2*.

Sample Traces

Statement	Object Values
	n1 = 123 n2 = 94
n1.Multiply (n2);	
	n1 = 11562 n2 = 94

Divide

Formal Contract Specification

```

procedure Divide (
    preserves Natural_Number& n,
    produces Natural_Number& rem
) is_abstract;
/*!
    requires
         $n > 0$ 
    ensures
         $\#self = self * n + rem$  and
         $0 \leq rem < n$ 
!*/

```

Informal Description

A call to the *Divide* operation has the form:

```
n1.Divide (n2, n3);
```

This operation divides *n1* by *n2* and returns with the remainder in *n3*.

Sample Traces

Statement	Object Values
	n1 = 647 n2 = 103 n3 = 3243455
n1.Divide (n2, n3);	
	n1 = 6 n2 = 103 n3 = 29

Copy_To

Formal Contract Specification

```

procedure Copy_To (
    produces Natural_Number& n
) is_abstract;
/*!
    preserves self
    ensures
        n = self
!*/

```

Informal Description

A call to the *Copy_To* operation has the form:

```
n1.Copy_To (n2);
```

This operation copies *n1* to *n2*.

Note — There is no assignment operator (=) for *Natural_Number* objects, but the same effect can be accomplished using *Copy_To*.

Sample Traces

Statement	Object Values
	n1 = 45142398 n2 = 1996
n1.Copy_To (n2);	
	n1 = 45142398 n2 = 45142398

Get_From (>>)

Formal Contract Specification

```

/*!
  math operation IS_DIGIT (
    c: character
  ): boolean is
    c is in {'0','1','2','3','4','5','6','7','8','9'}

  math subtype DIGIT is character
    exemplar d
    constraint
      IS_DIGIT (d)

  math operation IS_DIGITS (
    s: string of character
  ): boolean satisfies
    if s = empty_string
    then IS_DIGITS (s) = true
    else there exists a: string of character,
      c: character
      (s = a * <c> and
       IS_DIGITS (s) =
         (IS_DIGITS (a) and IS_DIGIT (c)))

  math operation INTEGER_VAL (
    s: string of DIGIT
  ): NATURAL_MODEL satisfies
    if s = empty_string
    then INTEGER_VAL (s) = 0
    else there exists a: string of DIGIT, d: DIGIT
      (s = a * <d> and
       INTEGER_VAL (s) = 10 * INTEGER_VAL (a)
        + TO_INTEGER (d) - TO_INTEGER ('0'))

  math operation IS_SPACES_AND_TABS (
    s: string of character
  ): boolean is
    elements (s) is subset of {' ', '\t'}

  math operation IS_AN_INPUT_REP (
    s: string of character,
    n: NATURAL_NUMBER
  ): boolean is
    there exists s1, s2, s3: string of character,
      c character
      (s = s1 * s2 * s3 * "\n" and
       IS_SPACES_AND_TABS (s1) and
       IS_DIGITS (s2) and
       n = INTEGER_VAL (s2) and
       IS_SPACES_AND_TABS (s3))

!*/

procedure Get_From (

```

```

        alters Character_IStream& str
    ) is_abstract;
    /*!
    produces self
    requires
        str.is_open = true and
        there exists x: NATURAL_MODEL,
            s: string of character
            (s is prefix of str.content and
             IS_AN_INPUT_REP (s, x))
    ensures
        str.is_open = true and
        str.ext_name = #str.ext_name and
        there exists s: string of character
            (#str.content = s * str.content and
             IS_AN_INPUT_REP (s, self))
    !*/

```

Informal Description

A call to the *Get_From* operation has the form:

```
n1.Get_From (input);
```

This operation reads *n1* from the input stream *input*. It alters *input* by reading and passing all characters up to and including the next newline ('\n') character. Before this newline must be only blanks (' ') or tabs ('\t'), followed by at decimal digits ('0', '1', ... ,9'), followed by only blanks or tabs. The digits denote the value of *n1* in the usual manner of writing decimal numbers — regardless of the value of *RADIX* (see *Multiply_By_Radix* above) — with the additional provision that an empty string of digits is read as 0. You may make this call only if *input* is open and if its content has a prefix that satisfies this requirement.

Along with *Get_From* comes the shorthand operator *>>*. The following two statements are equivalent:

```

n1.Get_From (input);
input >> n1;

```

Sample Traces

Statement	Object Values
	n1 = 45142398 input.is_open = true input.ext_name = "" input.content = "13\n 7\n\n5"
n1.Get_From (input);	
	n1 = 13 input.is_open = true input.ext_name = "" input.content = " 7\n\n5"
input >> n1;	
	n1 = 7 input.is_open = true input.ext_name = "" input.content = "\n5"
input >> n1;	
	n1 = 0 input.is_open = true input.ext_name = "" input.content = "5"

Put_To (<<)

Formal Contract Specification

```

    /*!
      math operation DIGIT_CHARACTER (
        i: integer
      ): character satisfies
      if 0 <= i < 10
      then DIGIT_CHARACTER (i) =
        TO_CHARACTER (TO_INTEGER ('0') + i)
      else DIGIT_CHARACTER (i) = '\0'

      math operation POSITIVE_OUTPUT_REP (
        n: NATURAL_MODEL
      ): string of character satisfies
      if n = 0
      then POSITIVE_OUTPUT_REP (n) = empty_string
      else there exists k, d: integer
        (n = 10 * k + d and
         0 <= d < 10 and
         POSITIVE_OUTPUT_REP (n) =
           POSITIVE_OUTPUT_REP (k) *
           <DIGIT_CHARACTER (d)>)

      math operation OUTPUT_REP (
        n: NATURAL_MODEL
      ): string of character satisfies
      if n = 0
      then OUTPUT_REP (n) = "0"
      else OUTPUT_REP (n) = POSITIVE_OUTPUT_REP (n)

    !*/

    procedure Put_To (
      alters Character_OStream& str
    ) is_abstract;

    /*!
      preserves self
      requires
        str.is_open = true
      ensures
        str.is_open = true and
        str.ext_name = #str.ext_name and
        str.content = #str.content * OUTPUT_REP (self)

    !*/

```

Informal Description

A call to the *Put_To* operation has the form:

```
n1.Put_To (output);
```

This operation writes the decimal representation of *n1* to the (file or text) output stream *output* — decimal, regardless of the value of *RADIX* (see *Multiply_By_Radix* above). It alters *output* only by appending to it all the characters (decimal digits) of the decimal representation of *n1*. You may make this call only if *output* is open.

Along with *Put_To* comes the shorthand operator <<. The following two statements are equivalent:

```
n1.Put_To (output);
output << n1;
```

Values output using operator << may be combined into single values, or may be strung together using operator << repeatedly, as illustrated in the last sample trace statement.

Sample Traces

Statement	Object Values
	n1 = 13 output.is_open = true output.ext_name = "" output.content = "x"
n1.Put_To (output);	
	n1 = 13 output.is_open = true output.ext_name = "" output.content = "x13"
...	
	n1 = 0 output.is_open = true output.ext_name = "" output.content = "x13"
output << n1;	
	n1 = 0 output.is_open = true output.ext_name = "" output.content = "x130"
output << "\nn1=" << n1;	
	n1 = 0 output.is_open = true output.ext_name = "" output.content = "x130\nn1=0"

Partial_Map

Motivation, Applicability, and Indications for Use

The programming type *Partial_Map* allows you to build, query, and dismantle a set of ordered pairs (2-tuples) whose two components are of two arbitrary types. Generally, the first component of each pair, which is considered to be a unique “key”, allows you to lookup the second component of that pair, which is considered to be information “associated with” the key value. There can be at most one pair in the set with any given key value.

For example, suppose you need to keep a simple database of employees and their salaries. To do this, you can create a *Partial_Map* object whose key type is *Text* (to store an employee name; this is also called the domain type, hence its name *D_Item*) and whose associated information type is *Integer* (to store an employee salary; this is also called the range type, hence its name *R_Item*). To enter the salary of an employee you simply define into the *Partial_Map* object the pair consisting of the employee’s name and the employee’s salary. Later you can lookup the employee’s salary by using the employee’s name as the key. Other operations allow you to manipulate and observe properties of a *Partial_Map* object, such as its size, i.e., the number of pairs it holds.

Related Components

- None

Component Family Members

Abstract Components

- *Partial_Map_Kernel* — the programming type of interest, with the operations below
 - *Define*
 - *Undefine*
 - *Undefine_Any*
 - *Accessor*
 - *Is_Defined*
 - *Size*

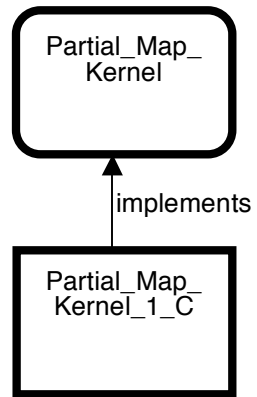
Concrete Components

- *Partial_Map_Kernel_1_C* — This is a checking implementation of *Partial_Map_Kernel* in which the execution time for each of the operations constructor, *Define*, *Undefine_Any*, and *Size* is constant, while the execution time for each of the operations destructor, *Undefine*, the accessor, and *Is_Defined* is proportional to the size of the set. All objects of this type have the interface of *Partial_Map_Kernel*, with the concrete template name *Partial_Map_Kernel_1_C* substituted for the abstract template name *Partial_Map_Kernel*.

To bring this component into the context you write:

```
#include "CT/Partial_Map/Kernel_1_C.h"
```

Component Coupling Diagram



Descriptions

Partial_Map_Kernel Type and Standard Operations

Formal Contract Specification

```

    /*!
      math subtype PARTIAL_FUNCTION is finite set of (
        d: D_Item,
        r: R_Item
      )
      exemplar m
      constraint
        for all d1, d2: D_Item, r1, r2: R_Item
          where ((d1,r1) is in m and
                (d2,r2) is in m)
            (if d1 = d2 then r1 = r2)

      math operation IS_DEFINED_IN (
        m: PARTIAL_FUNCTION,
        d: D_Item
      ): boolean is
        there exists r: R_Item
          ((d,r) is in m)
    !*/

    standard_abstract_operations (Partial_Map_Kernel);
    /*!
      Partial_Map_Kernel is modeled by PARTIAL_FUNCTION
      initialization ensures
        self = empty_set
    !*/

```

Informal Description

The model for *Partial_Map_Kernel* is a finite set of ordered pairs of type (D_Item, R_Item) which is initially empty. A *Partial_Map_Kernel* object may be arbitrarily large. Like all Resolve/C++ types, *Partial_Map_Kernel* comes with operator `&=` and *Clear*.

Note — The sample traces for this and the other operation descriptions refer to the type *Partial_Map_Instance*, which is the result of the following instantiation:

```

concrete_instance
class Partial_Map_Instance :
  instantiates
    Partial_Map_Kernel_1_C <Text, Integer>
{};

```

Sample Traces

Statement	Object Values
	(m1 is not in scope)
object Partial_Map_Instance m1;	
	m1 = { }

Define

Formal Contract Specification

```

procedure Define (
    consumes D_Item& d,
    consumes R_Item& r
) is_abstract;
/*!
    requires
        not IS_DEFINED_IN (self, d)
    ensures
        self = #self union {(#d,#r)}
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Partial_Map_Instance m1, m2;
object Text t1, t2;
object Integer i;
object Boolean b;

```

A call to the *Define* operation has the form:

```
m1.Define (t1, i);
```

This operation adds the pair whose value is $(t1, i)$ to the set $m1$. You may make this call only if $m1$ does not already contain a pair whose first component is $t1$.

Sample Traces

Statement	Object Values
	<pre> m1 = { ("you", 94), ("me", 15) } t1 = "her" i = 33 </pre>
m1.Define (t1, i);	
	<pre> m1 = { ("you", 94), ("me", 15), ("her", 33) } t1 = "" i = 0 </pre>

Undefine

Formal Contract Specification

```

procedure Undefine (
    preserves D_Item& d,
    produces D_Item& d_copy,
    produces R_Item& r
) is_abstract;
/*!
    requires
        IS_DEFINED_IN (self, d)
    ensures
        d_copy = d and
        (d_copy, r) is in #self and
        self = #self - {(d_copy, r)}
!*/

```

Informal Description

A call to the *Undefine* operation has the form:

```
m1.Undefine (t1, t2, i);
```

This operation removes from *m1* the pair whose first component is *t1* and returns its value in (*t2*, *i*). You may make this call only if there is a pair in *m1* whose first component is *t1*.

Sample Traces

Statement	Object Values
	m1 = { ("you", 94), ("me", 15), ("her", 33) } t1 = "me" t2 = "anything" i = -47
m1.Undefine (t1, t2, i);	
	m1 = { ("you", 94), ("her", 33) } t1 = "me" t2 = "me" i = 15

Undefine_Any

Formal Contract Specification

```

procedure Undefine_Any (
    produces D_Item& d,
    produces R_Item& r
) is_abstract;
/*!
    requires
        self /= empty_set
    ensures
        (d, r) is in #self and
        self = #self - {(d,r)}
!*/

```

Informal Description

A call to the *Undefine_Any* operation has the form:

```
m1.Undefine_Any (t1, i);
```

This operation removes some pair from *m1*—any pair—and returns its value in *(t1, i)*. You may make this call only if *m1* is not empty.

Notice that this operation may remove a different pair from *m1* if called twice in identical states, as illustrated in the sample tracing table.

Sample Traces

Statement	Object Values
	m1 = { ("you", 94), ("me", 15), ("her", 33) } t1 = "who?" i = -47
m1.Undefine_Any (t1, i);	
	m1 = { ("you", 94), ("her", 33) } t1 = "me" i = 15
...	
	m1 = { ("you", 94), ("me", 15), ("her", 33) } t1 = "who?" i = -47
m1.Undefine_Any (t1, i);	
	m1 = { ("me", 15), ("her", 33) } t1 = "you" i = 94

Accessor

Formal Contract Specification

```
function R_Item& operator [] (
    preserves D_Item& d
) is_abstract;
/*!
    requires
        IS_DEFINED_IN (self, d)
    ensures
        (d, self[d]) is in self
!*/
```

Informal Description

A call to the accessor operator [] has the form of an expression:

```
... m1[t1] ...;
```

This expression acts as the name of an object of type *R_Item* (i.e., *Integer* in this example) whose value is that paired with *t1* in *m1*. You may make this call only if there is a pair in *m1* whose first component is *t1*.

You may use *m1[t1]* wherever any other object of type *R_Item* (i.e., *Integer* in this case) may appear.

The accessor operator [], like all functions, preserves its arguments. But it is important to realize that the accessor expression *m1[t1]* does not simply denote the value associated with *t1* in *m1*, it acts as the name of an object of type *R_Item* which you may consider to be in that particular pair in *m1*. This means that not only may you use the expression *m1[t1]* as a value of type *R_Item*; you may even change the value of the object called *m1[t1]*—but remember that this also changes the value of *m1*.

The sample traces help illustrate some important points regarding the convenience and danger of accessor expressions. The first and second sample statements show how an accessor expression such as *m1[t1]* acts like an object of type *R_Item* whose value may be changed, depending on the statement in which the expression occurs. The third sample statement shows that *m1[t1]* may appear as an argument in a call where an *R_Item* is required. This call involves other arguments (e.g., *m2*), but none of the other arguments may be *m1* or may involve an accessor expression for *m1* because this would violate the repeated argument rule. So, while you might be tempted to write something like:

```
m1.Define (t1, m1[t2]);
```

this statement violates the repeated argument rule. Be aware that the C++ compiler will not report this as an error. You need to avoid the pitfall by personal discipline.

The last statement in the tracing table also illustrates that an accessor expression acts exactly like an object of type *R_Item*. Notice that both *t1* and *m1[t2]*, which are being *Defined* into *m2* in this statement, are consumed (their new values are "" and 0, respectively) because *Define* consumes its arguments. So this statement changes both *m1* and *m2*. This kind of complicated call is officially legal but it is not recommended because it is very hard to understand.

Sample Traces

Statement	Object Values
	<pre> m1 = { ("you", 94), ("me", 15), ("her", 33) } t1 = "her" i = -47 </pre>
<code>m1[t1] &= i;</code>	
	<pre> m1 = { ("you", 94), ("me", 15), ("her", -47) } t1 = "her" i = 33 </pre>
<code>m1[t1] = i;</code>	
	<pre> m1 = { ("you", 94), ("me", 15), ("her", 33) } t1 = "her" i = 33 </pre>
<code>...</code>	
	<pre> m1 = { ("you", 94), ("me", 15), ("her", -47) } m2 = { ("him", 65), ("you", 4) } t1 = "her" t2 = "you" </pre>
<code>m2.Define (t1, m1[t2]);</code>	
	<pre> m1 = { ("you", 0), ("me", 15), ("her", -47) } m2 = { ("him", 65), ("you", 4), ("her", 94) } t1 = "" t2 = "you" </pre>

Is_Defined

Formal Contract Specification

```

function Boolean Is_Defined (
    preserves D_Item& d
) is_abstract;
/*!
    ensures
        Is_Defined = IS_DEFINED_IN (self, d)
!*/

```

Informal Description

A call to the *Is_Defined* operation has the form of an expression:

```
... m1.Is_Defined (t1) ...;
```

This operation returns **true** iff there is a pair in *m1* whose first component is *t1*.

Sample Traces

Statement	Object Values
	m1 = { ("you",0), ("me",15), ("her",-47) } t1 = "him" b = true
b = m1.Is_Defined (t1);	
	m1 = { ("you",0), ("me",15), ("her",-47) } t1 = "him" b = false
t1 = "her";	
	m1 = { ("you",0), ("me",15), ("her",-47) } t1 = "her" b = false
b = m1.Is_Defined (t1);	
	m1 = { ("you",0), ("me",15), ("her",-47) } t1 = "her" b = true

Size

Formal Contract Specification

```
function Integer Size () is_abstract;
/*!
    ensures
        Size = |self|
!*/
```

Informal Description

A call to the *Size* operation has the form of an expression:

```
... m1.Size () ...;
```

This operation returns an *Integer* value which is the size of the set *m1*, i.e., $|m1|$.

Sample Traces

Statement	Object Values
	<pre>m1 = { ("you",0), ("me",15), ("her",-47) } i = -68</pre>
<code>i = m1.Size ();</code>	
	<pre>m1 = { ("you",0), ("me",15), ("her",-47) } i = 3</pre>

Queue

Motivation, Applicability, and Indications for Use

The programming type *Queue* allows you to collect and later process items in first-in-first-out (“FIFO”) order. For example, suppose you need to keep track of requests for service and to service them in the order in which they were received. To do this, you can create a *Queue* object whose elements are of a type that records the information for a single request, enqueue each request as it arrives, and dequeue a request whenever you are ready to service the next one. Other operations allow you to access the front item in a *Queue* object and to determine its length.

Related Components

- *Stack* — a type that is similar to *Queue* except that when you remove an item it is not the first one that was put in, but rather the last one
- *Sorting_Machine* — a type that is similar to *Queue* except that when you remove an item it is not the first one that was put in, but the next one in “sorted” order according to some ordering based on the values of the items

Component Family Members

Abstract Components

- *Queue_Kernel* — the programming type of interest, with the operations below
 - *Enqueue*
 - *Dequeue*
 - *Accessor*
 - *Length*

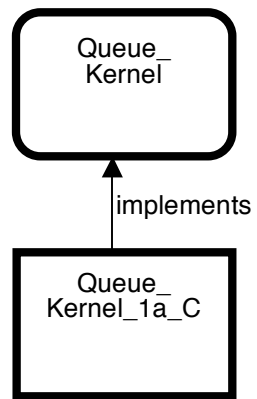
Concrete Components

- *Queue_Kernel_1a_C* — This is a checking implementation of *Queue_Kernel* in which the execution time for each of the operations constructor, *Enqueue*, *Dequeue*, the accessor, and *Length* is constant, while the execution time for the destructor is proportional to the length of the string. All objects of this type have the interface of *Queue_Kernel*, with the concrete template name *Queue_Kernel_1a_C* substituted for the abstract template name *Queue_Kernel*.

To bring this component into the context you write:

```
#include "CT/Queue/Kernel_1a_C.h"
```

Component Coupling Diagram



Descriptions

Queue_Kernel Type and Standard Operations

Formal Contract Specification

```

standard_abstract_operations (Queue_Kernel);
/*!
    Queue_Kernel is modeled by string of Item
    initialization ensures
        self = empty_string
!*/

```

Informal Description

The model for *Queue_Kernel* is a string of items which is initially empty. A *Queue_Kernel* object may be arbitrarily large. Like all Resolve/C++ types, *Queue_Kernel* comes with operator *&=* and *Clear*.

Note — The sample traces for this and the other operation descriptions refer to the type *Queue_Of_Integer*, which is the result of the following instantiation:

```

concrete_instance
class Queue_Of_Integer :
    instantiates
        Queue_Kernel_1a_C <Integer>
{};

```

Sample Traces

Statement	Object Values
	(q1 is not in scope)
object Queue_Of_Integer q1;	
	q1 = < >

Enqueue

Formal Contract Specification

```

procedure Enqueue (
    consumes Item& x
) is_abstract;
/*!
    ensures
        self = #self * <#x>
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Queue_Of_Integer q1, q2;
object Integer i;

```

A call to the *Enqueue* operation has the form:

```
q1.Enqueue (i);
```

This operation adds the item whose value is *i* to the “rear” (right end) of *q1*.

Sample Traces

Statement	Object Values
	q1 = < 58, 12, 21 > i = 13
q1.Enqueue (i);	
	q1 = < 58, 12, 21, 13 > i = 0

Dequeue

Formal Contract Specification

```

procedure Dequeue (
    produces Item& x
) is_abstract;
/*!
    requires
        self /= empty_string
    ensures
        #self = <x> * self
!*/

```

Informal Description

A call to the *Dequeue* operation has the form:

```
q1.Dequeue (i);
```

This operation removes the item from the “front” (left end) of *q1* and returns its value in *i*. You may make this call only if *q1* is not empty.

Sample Traces

Statement	Object Values
	q1 = < 58, 12, 21, 13 > i = 92
q1.Dequeue (i);	
	q1 = < 12, 21, 13 > i = 58

Accessor

Formal Contract Specification

```
function Item& operator [] (
    preserves Accessor_Position& current
) is_abstract;
/*!
    requires
        self /= empty_string
    ensures
        there exists a: string of Item
            (self = <self[current]> * a)
!*/
```

Informal Description

A call to the accessor operator [] has the form of an expression:

```
... q1[current] ...;
```

This expression acts as the name of an object of type *Item* whose value is the item at the front of *q1*. You may make this call only if *q1* is not empty.

The special word *current* is the only thing that can appear between [] in the accessor for type *Queue*. It always refers to the single item that you can access directly without dequeuing any others: the front (leftmost) item, which is the one that would be removed if you called the *Dequeue* operation at the point where you are using the accessor operator. You may use *q1[current]* wherever any other object of type *Item* may appear.

The accessor operator [], like all functions, preserves its arguments. But it is important to realize that the accessor expression *q1[current]* does not simply denote the value of the front item in *q1*, it acts as the name of an object of type *Item* which you may consider to lie at the left end of the string *q1*. This means that not only may you use the expression *q1[current]* as a value of type *Item*; you may even change the value of the object called *q1[current]* — but remember that this also changes the value of *q1*.

The sample traces help illustrate some important points regarding the convenience and danger of accessor expressions. The first and second sample statements show how an accessor expression such as *q1[current]* acts like an object of type *Item* whose value may be changed, depending on the statement in which the expression occurs. The third sample statement shows that *q1[current]* may appear as an argument in a call where an *Item* is required. This call involves other arguments (e.g., *q2*), but none of the other arguments may be *q1* or may involve an accessor expression for *q1* because this would violate the repeated argument rule. So, while you might be tempted to write something like:

```
q1.Enqueue (q1[current]);
```

this statement violates the repeated argument rule. Be aware that the C++ compiler will not report this as an error. You need to avoid the pitfall by personal discipline.

The last statement in the tracing table also illustrates that an accessor expression acts exactly like an object of type *Item*. Notice that *q1[current]*, which is being enqueued onto *q2* in this statement, is consumed (its new value is 0 since *Item* is *Integer* here) because *Enqueue* consumes its argument. So this statement changes both *q1* and *q2*.

Sample Traces

Statement	Object Values
	q1 = < 92, 6, 18 > i = 37
q1[current] &= i;	
	q1 = < 37, 6, 18 > i = 92
q1[current] = i;	
	q1 = < 92, 6, 18 > i = 92
...	
	q1 = < 92, 255, 6 > q2 = < 76, 921, 34, 7 >
q2.Enqueue (q1[current]);	
	q1 = < 0, 255, 6 > q2 = < 76, 921, 34, 7, 92 >

Length

Formal Contract Specification

```

function Integer Length () is_abstract;
  /*!
    ensures
      Length = |self|
  !*/

```

Informal Description

A call to the *Length* operation has the form of an expression:

```
... q1.Length () ...;
```

This operation returns an *Integer* value which is the length of the string *q1*, i.e., $|q1|$.

Sample Traces

Statement	Object Values
	q1 = < 6, 92, 13, 18 > i = -68
i = q1.Length ();	
	q1 = < 6, 92, 13, 18 > i = 4

Record

Record is a built-in type in Resolve/C++, which is included when you bring `RESOLVE_Foundation.h` into the global context. The name *Record* is therefore unusual: it is the name of a concrete template even though it does not end in “_2” or “_1a” or the like.

Motivation, Applicability, and Indications for Use

The programming type *Record* allows you to set up and access the individual objects in a small, fixed-size (no larger than ten) collection of objects of various and arbitrary types. You should consider *Record* whenever you want to treat such a collection of diverse objects as a single object for some purposes, and as individual constituent sub-objects for other purposes. The way you achieve the former behavior is straightforward, since this is what an “object” ordinarily is. The way you achieve the latter behavior at the same time is to give symbolic names (called *field names*) to the sub-objects (also called *fields*) of a *Record* object. By using these field names with accessor operations, you can directly manipulate the fields of a *Record* object. In fact, this is the only way to manipulate the sub-objects. The mathematical model of a *Record*, then, is just a tuple whose members are the *Record* fields.

Let’s consider an example. Suppose you want to keep track of the following information for each employee of a company: first name, middle name, last name, address, and salary. For some purposes you want all the information about a particular employee to stay together (e.g., perhaps you want a *Sequence* whose *Items* are this *Record* type; or you want this *Record* type to be the *R_Item* in a *Partial_Map*, with the employee’s ID number as the *D_Item*). For other purposes you want to be able to get to the individual pieces of information about a particular employee (e.g., perhaps you want to give the employee a raise).

There are several different ways to use *Record* to do this. Here are two approaches:

- You might instantiate the **concrete_template class** *Record* with five fields of the types *Text*, *Text*, *Text*, *Text*, and *Integer*, calling the resulting concrete instance *Employee*. Then you might declare *Employee*’s field names to be *first_name*, *middle_name*, *last_name*, *address*, and *salary*, respectively. Schematically, you could view as follows an object of this type that contains information about John Q. Doe, an employee who lives at 123 Easy Street and makes \$85,600 per year:

“Employee” field name	field value
<code>first_name</code>	"John"
<code>middle_name</code>	"Q."
<code>last_name</code>	"Doe"
<code>address</code>	"123 Easy St."
<code>salary</code>	85600

The mathematical model of this value is a five-tuple:

`("John", "Q.", "Doe", "123 Easy St.", 85600)`

- You might instantiate the **concrete_template class** *Record* with three fields of types *Text*, *Text*, and *Text*, calling the resulting concrete instance *Full_Name*. The you might declare

Full_Name's field names to be *first_name*, *middle_name*, and *last_name*. Then you might instantiate the **concrete_template class** *Record* again, creating the concrete instance *Employee* with three fields of types *Full_Name*, *Text*, and *Integer*, and field names *name*, *address*, and *salary*. Schematically, you could view the situation like this:

"Full_Name" field name	field value
first_name	"John"
middle_name	"Q."
last_name	"Doe"

"Employee" field name	field value
name	("John", "Q.", "Doe")
address	"123 Easy St."
salary	85600

The mathematical model of this value is a three-tuple whose first component is another three-tuple:

```
(( "John", "Q.", "Doe" ), "123 Easy St.", 85600)
```

The first approach gives a flat *Record* with five fields and no further organization. The second approach—probably the better one from the standpoint of human understanding—gives a more structured *Record*, one of whose fields is itself a *Record* that is meaningful even outside the context of employee information. In fact, in practice you probably would need more detailed address information. So you'd also want to subdivide the address into street address, city, etc., by making a concrete instance *Address* as an instance of **concrete_template class** *Record* in much the same fashion as *Full_Name* above.

On the other hand, suppose you also wanted to keep track of each employee's birthdate. This suggests another concrete instance *Date*—which also would be generally useful outside this application. The mathematical model of *Date* probably should be a three-tuple of integers with some constraints on the values of the fields. However, *Date* should not simply be a *Record* instance because there also are mutual constraints on the values of the fields. That is, the day must be between 1 and 31 inclusive if the month is 1, but between 1 and 28 inclusive if the month is 2 (except if the year is a leap year, in which case 1 through 29 is allowed), and so on. The complication introduced by such a constraint is a clue that it is inappropriate to treat a date as merely three independent numbers which are placed together for convenience—which is how you'd be treating it if you made *Date* an instance of *Record*.

You can guess from the above description that *Record* is an unusual template component in the sense that different instantiations of it involve different numbers of template parameters! Indeed this is so; here are the Resolve/C++ statements that declare the concrete instances and their field names for two approaches above:

```
// First approach: flat record
```

```
concrete_instance
class Employee :
    instantiates
```



```

    Record <
        Text,
        Text,
        Text,
        Text,
        Integer
    >

{};

field_name (Employee, 0, Text, first_name);
field_name (Employee, 1, Text, middle_name);
field_name (Employee, 2, Text, last_name);
field_name (Employee, 3, Text, address);
field_name (Employee, 4, Integer, salary);

// Second approach: structured record (alternatively, not in addition)

concrete_instance
class Full_Name :
    instantiates
        Record <
            Text,
            Text,
            Text
        >
{};

field_name (Full_Name, 0, Text, first_name);
field_name (Full_Name, 1, Text, middle_name);
field_name (Full_Name, 2, Text, last_name);

concrete_instance
class Employee :
    instantiates
        Record <
            Full_Name,
            Text,
            Integer
        >
{};

field_name (Employee, 0, Full_Name, name);
field_name (Employee, 1, Text, address);
field_name (Employee, 2, Integer, salary);

```

A closely related component is *Array*. The most important difference is that the individual objects in a *Record* may be of different types, while all the individual objects in an *Array*—and in most other collections of objects—must be of the same type. However, a *Record* can contain at most ten fields, and the number of fields is fixed at instantiation time; while an *Array* may contain arbitrarily many objects and its size is determined at run-time by calling the *Set_Bounds* operation.

One other component is closely related to *Record*. It is essentially identical to *Record*—except that its name is *Representation*—and it is used for one and only one thing in Resolve/C++ programs: If you are implementing a kernel component, then you always collect together the individual objects that you are using to represent an object of the new type, and make them fields

of an instance of *Representation*. Client programmers need to know nothing about *Representation* except that (a) it is essentially *Record* with a different name, and (b) it shouldn't be used in ordinary client programs.

Related Components

- *Array* and *Static_Array* — types that are similar to *Record* except that they may contain objects of only one type; but they may contain far more objects than a *Record*
- *Representation* — a type that is virtually identical to *Record* except that it is used in only one place: to hold the individual objects that comprise the representation of a kernel abstraction (and, hence, of no use to client programmers)

Component Family Members

Abstract Components

- *Record_Kernel* — the programming type of interest, with the operations below
 - *Accessor* — one accessor for each field

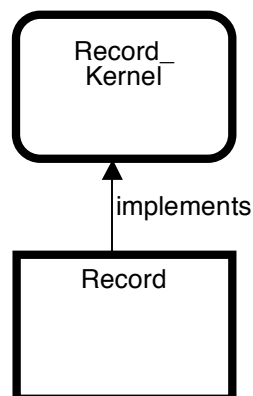
Concrete Components

- *Record* — This is the standard implementation of the abstract template *Record_Kernel* that comes built-in to Resolve/C++. It uses a technique called “lazy initialization” so the constructor is very fast and takes time independent of the number and types of the fields. You pay the cost of the constructors for the fields only upon the first use of some *Record* accessor. The destructor's execution time is essentially the sum of the execution times of the destructors for the fields (if the constructors have actually have been invoked). All the *Record* accessors take constant time once field construction takes place.

To bring this component into the context you write:

```
#include "RESOLVE_Foundation.h"
```

Component Coupling Diagram



Descriptions

Record_Kernel Type and Standard Operations

Formal Contract Specification

```

standard_abstract_operations (Record_Kernel);
/*!
    Record_Kernel is modeled by (
        field0: Item0
        field1: Item1
        ...
        field9: Item9
    )
    initialization ensures
        is_initial (self.field0)    and
        is_initial (self.field1)    and
        ... and
        is_initial (self.field9)    and
!*/

```

Informal Description

Record_Kernel is a special component that can be instantiated with any number of fields, up to ten. The above formal description, then, is not legitimate C++ code because, in C++, a template has a fixed and definite number of parameters. The above “schema” describes (with “...”) the nature of *Record_Kernel* as a client should view it—but not the exact code used for it. The intended meaning is that when you instantiate this template with a particular number of fields, say 3, then the *Item* types are *Item0* through *Item2* and the component names in the tuple model are *field0* through *field2*.

Upon declaration, an object of this type has a value in which all its fields have initial values for their respective types. Like all Resolve/C++ types, *Record_Kernel* comes with operator `&=` and *Clear*.

The discussion and sample traces for this and the other operation descriptions refer to the type *Full_Name*, which is the result of the following instantiation and field name declarations (repeated from the example above):

```

concrete_instance
class Full_Name :
    instantiates
        Record <
            Text,
            Text,
            Text
        >
    {};

field_name (Full_Name, 0, Text, first_name);
field_name (Full_Name, 1, Text, middle_name);
field_name (Full_Name, 2, Text, last_name);
...

```

```

object Full_Name n1, n2;
object Character c;
object Text t;

```

The names of the fields are declared separately from the instantiation of the template. Typically, these declarations appear immediately following the instantiation, but are needed only if the fields are to be accessed there. This might not be so in all cases; for example, a *Record* instantiation might be needed to create a concrete instance to pass as a template parameter in another instantiation. Perhaps the fields of objects of this type will be accessed only in the second template, in which case the field names should only be declared there.

The declaration of a field name is not a call to an operation, though it looks like one, e.g.:

```

field_name (Full_Name, 1, Text, middle_name);

```

The first argument is the name of the *Record* instance to which the new field name applies, in this case *Full_Name*. The second argument is the field number to which the new field name refers, in this case the second field (i.e., field number 1). The third argument is the type of the field being named, in this case *Text*. The fourth argument is the field name being declared, in this case *middle_name*. Notice that a field name must have the syntax of an identifier in C++, i.e., it must look like the name of an object. Moreover, for all other intents and purposes *middle_name* is treated exactly like the name of an object: its scope is identical to that of any other object declared at this point, and there must be no other object in the current scope with this name.

Two other possibly surprising facts about field names are:

- You may have two names for the same field of the same *Record* instance, e.g.:

```

field_name (Full_Name, 1, Text, middle_name);
field_name (Full_Name, 1, Text, m_name);

```

Ordinarily you would not make these two declarations in the same scope because it is sure to be confusing to a reader. But a field name has the same scope as an object declaration made at the same point. So sometimes you will have different names for the same field because one field name is declared in one scope and another is declared in a different scope. This happens, for example, when a *Record* instance is passed as a template parameter, and there is nothing wrong with it.

- The first and third arguments to **field_name** are for human consumption only! Among other things, this means that the C++ compiler will let you—but you should not—use field names from one *Record* instance to access the fields of a different *Record* instance.

Sample Traces

Statement	Object Values
	(n1 is not in scope)
object Full_Name n1;	
	n1 = ("", "", "")

Accessor

Note — There is one accessor operation per field. The description here uses the accessor operation for the first field (i.e., *field0*) as a prototype for the others, which work similarly.

Formal Contract Specification

```
function Item0& operator [ ] (
    preserves Accessor0& fn0
) is_abstract;
/*!
    ensures
        self = (self[fn0], self.field1, ..., self.field9)
!*/
```

Informal Description

A call to the accessor operator [] has the form of an expression:

```
... n1[first_name] ...;
```

This expression acts as the name of an object of type *Item0* (i.e., *Text* in this case) which is the sub-object in the first field of *n1* (i.e., the one whose field name is *first_name* in this case). You may use *n1[first_name]* wherever any other object of type *Text* may appear.

The accessor operator [], like all functions, preserves its arguments. But it is important to realize that the accessor expression *n1[first_name]* does not simply denote the value of the object in field *first_name* of *n1*, it acts as the name of an object of type *Text* which is the sub-object in that particular field of *n1*. This means that not only may you use the expression *n1[first_name]* as a value of type *Text*; you may even change the value of the object called *n1[first_name]*—but remember that this also changes the value of *n1*.

The sample traces help illustrate some important points regarding the accessor expressions. The first and second sample statements show how an accessor expression such as *n1[first_name]* acts like an object of type *Text* whose value may be changed, depending on the statement in which the expression occurs. The accessors for *Record* are different than other accessors in an important way, however:

It is **not** a violation of the repeated argument rule to use two or more different fields of the same *Record* object in a single call.

You may use the same field of two different *Record* objects in a single call, e.g., as illustrated in the tracing table where the *first_name* fields of *n1* and *n2* are swapped. This would not be a repeated argument violation under any circumstances because *n1* and *n2* are different objects. What's different with *Record* objects is that it is also acceptable to use two different fields of the same *Record*, as illustrated in the last tracing table example where the *first_name* and *middle_name* fields of *n1* are swapped.

Here's how to think of it: *Record* is merely a way to declare a group of objects with names of the form *r[f]*, where *r* is the name of the common *Record* they're part of (presumably because they have something in common with each other), and the various *f*'s allow you distinguish among the objects in that *Record*.

Sample Traces

Statement	Object Values
	n1 = ("John", "Q.", "Doe") t = "Adams"
n1[last_name] &= t;	
	n1 = ("John", "Q.", "Adams") t = "Doe"
...	
	n1 = ("John", "Q.", "Adams") c = 'x'
n1[first_name].Remove (2, c);	
	n1 = ("Jon", "Q.", "Adams") c = 'h'
...	
	n1 = ("Jon", "Q.", "Adams") n2 = ("J.", "F.", "Kennedy")
n1[middle_name] &= n2[middle_name];	
	n1 = ("Jon", "F.", "Adams") n2 = ("J.", "Q.", "Kennedy")
n1[middle_name] &= n1[first_name];	
	n1 = ("F.", "Jon", "Adams") n2 = ("J.", "Q.", "Kennedy")

Sequence

Motivation, Applicability, and Indications for Use

Occasionally you need something like *Text*, but with the individual items in the string as something other than *Characters*. The interesting functionality of *Text* is that you can create and manipulate arbitrarily long strings by adding and/or removing items, and that you can directly index into a string by position. If you need the generalized (template) version of *Text* in which you, as a client programmer, can decide what type the items of the string can be, then you should use *Sequence*.

Related Components

- *Text* — the built-in RESOLVE/C++ type whose generalization is *Sequence*
- *Array* — a type that is similar to *Sequence* in that you can directly index into an *Array* object by position, but different in that an *Array* object always has the same number of items in it and this number cannot be changed dynamically

Component Family Members

Abstract Components

- *Sequence_Kernel* — the programming type of interest, with the operations below
 - *Add*
 - *Remove*
 - *Accessor*
 - *Length*

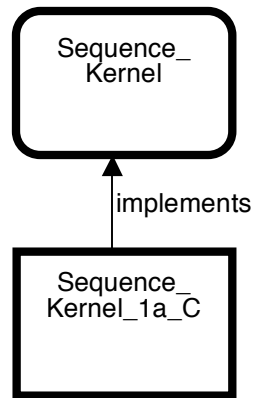
Concrete Components

- *Sequence_Kernel_1a_C* — This is a checking implementation of *Sequence_Kernel* in which the execution time for each of the operations constructor and *Length* is constant; the execution time each of the operations *Add*, *Remove*, and the accessor is proportional to the position being indexed; and the execution time for the destructor is proportional to the length of the string. All objects of this type have the interface of *Sequence_Kernel*, with the concrete template name *Sequence_Kernel_1a_C* substituted for the abstract template name *Sequence_Kernel*.

To bring this component into the context you write:

```
#include "CT/Sequence/Kernel_1a_C.h"
```

Component Coupling Diagram



Descriptions

Sequence_Kernel Type and Standard Operations

Formal Contract Specification

```

standard_abstract_operations (Sequence_Kernel);
/*!
    Sequence_Kernel is modeled by string of Item
    initialization ensures
        self = empty_string
!*/

```

Informal Description

The model for *Sequence_Kernel* is a string of items which is initially empty. A *Sequence_Kernel* object may be arbitrarily long. Like all Resolve/C++ types, *Sequence_Kernel* comes with operator `&=` and *Clear*.

Note — The sample traces for this and the other operation descriptions refer to the type *Sequence_Of_Integer*, which is the result of the following instantiation:

```

concrete_instance
class Sequence_Of_Integer :
    instantiates
        Sequence_Kernel_1_C <Integer>
{};

```

Sample Traces

Statement	Object Values
	(s1 is not in scope)
object Sequence_Of_Integer s1;	
	s1 = < >

Add

Formal Contract Specification

```

procedure Add (
    preserves Integer pos,
    consumes Item& x
) is_abstract;
/*!
    requires
        0 <= pos <= |self|
    ensures
        there exists a, b: string of Item
            (|a| = pos and
             #self = a * b and
             self = a * <#x> * b)
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Sequence_Of_Integer s1, s2;
object Integer i, n;

```

A call to the *Add* operation has the form:

```

s1.Add (i, n);

```

This operation adds n in position i of string $s1$, i.e., in a position such that the number of items before the newly added one is equal to i . Notice that the value of n is consumed, so afterward $n = 0$. You may make this call only if $0 \leq i \leq |s1|$.

You can see from this example that the numbering of positions in the string $s1$ effectively starts at 0. For instance, to add n at the beginning of $s1$ (i.e., as the leftmost item of $s1$) you might write:

```

s1.Add (0, n);

```

Sample Traces

Statement	Object Values
	s1 = < 9 6 90 > i = 2 n = 13
s1.Add (i, n);	
	s1 = < 9 6 13 90 > i = 2 n = 0
s1.Add (4, n);	
	s1 = < 9 6 13 90 0 > i = 2 n = 0

Remove

Formal Contract Specification

```

procedure Remove (
    preserves Integer pos,
    produces Item& x
) is_abstract;
/*!
    requires
        0 <= pos < |self|
    ensures
        there exists a, b: string of Item
            (|a| = pos and
             #self = a * <x> * b and
             self = a * b)
!*/

```

Informal Description

A call to the *Remove* operation has the form:

```
s1.Remove (i, n);
```

This operation removes the item in position i of string $s1$ and returns it in n . You may make this call only if $0 \leq i < |s1|$.

The numbering of positions in the string $s1$ starts at 0. For example, to remove the leftmost item of $s1$ and return it in n you might write:

```
s1.Remove (0, n);
```

Sample Traces

Statement	Object Values
	$s1 = \langle 9 \ 6 \ 90 \ 13 \rangle$ $i = 1$ $n = 279$
<code>s1.Remove (i, n);</code>	
	$s1 = \langle 9 \ 90 \ 13 \rangle$ $i = 1$ $n = 6$
<code>s1.Remove (2, n);</code>	
	$s1 = \langle 9 \ 90 \rangle$ $i = 1$ $n = 13$

Accessor

Formal Contract Specification

```

function Item& operator [ ] (
    preserves Integer pos
) is_abstract;
/*!
    requires
        0 <= pos < |self|
    ensures
        there exists a, b: string of Item
            (|a| = pos and
             #self = a * <self[pos]> * b)
!*/

```

Informal Description

A call to the accessor operator [] has the form of an expression:

```
... s1[i] ...;
```

This expression acts as the name of an object of type *Item* whose value is the item in position i of string $s1$. You may make this call only if $0 \leq i < |s1|$.

The numbering of positions in the string $s1$ starts at 0. For example, to set the leftmost item of $s1$ to 17 you might write (assuming $|s1| \geq 1$):

```
s1[0] = 17;
```

Notice that you can do the above assignment only because the assignment operator is defined for the type *Item*, in this case *Integer*. You may do to $s1[i]$ only what you may do to any other object of type *Item*.

The accessor operator [], like all functions, preserves its arguments ($s1$ and i in the example). But it is important to realize that the accessor expression $s1[i]$ does not simply denote the value of the item in position i of string $s1$; it acts as the name of an object of type *Item* which you may consider to lie in that position of the string $s1$. This means that not only may you use the expression $s1[i]$ as a value of type *Item*; you may even change the value of the object called $s1[i]$ — but remember that this also changes the value of $s1$.

The sample traces help illustrate some important points regarding the convenience and danger of accessor expressions. The first and second sample statements show an accessor expression such as $s1[i]$ acts like an object of type *Item* whose value may be changed, depending on the statement in which the expression occurs. The third sample statement shows that $s1[i]$ may appear as an argument in a call where an *Item* value (object or constant) is required. This call involves other arguments (e.g., $s2$), but none of the other arguments may be $s1$ or may involve an accessor expression for $s1$ because this would violate the repeated argument restriction.

So, while you might be tempted to write something like:

```
s1[i] &= s1[24];
```

this statement violates the repeated argument restriction for the swap operator. This conclusion is especially clear if $i = 24$ at the time the statement is executed, but it is true even if i has some other value since both swap operator arguments are accessor expressions for the same object,

namely *s1*. Be aware that the C++ compiler will not report this as an error. You need to avoid the pitfall by personal discipline.

The last statement in the tracing table again illustrates that an accessor expression acts exactly like an object of type *Item*. Notice that *s1[i]*, which is being added to *s2* in this statement, is consumed (its new value is 0 since *Item* is *Integer* here) because *Add* consumes its second argument. So this statement changes both *s1* and *s2*.

Sample Traces

Statement	Object Values
	s1 = < 92 6 18 > i = 1 n = 255
s1[i] &= n;	
	s1 = < 92 255 18 > i = 1 n = 6
s1[2] = n;	
	s1 = < 92 255 6 > i = 1 n = 6
...	
	s1 = < 92 255 6 > i = 2 s2 = < 76 921 34 7 >
s2.Add (3, s1[i]);	
	s1 = < 92 255 0 > i = 2 s2 = < 76 921 34 6 7 >

Length

Formal Contract Specification

```

function Integer Length () is_abstract;
/*!
    ensures
        Length = |self|
!*/

```

Informal Description

A call to the *Length* operation has the form of an expression:

```
... s1.Length () ...;
```

This operation returns an *Integer* value which is the length of string *s1* , i.e., |*s1*|.

Sample Traces

Statement	Object Values
	s1 = < 12 21 58 > i = 1
i = s1.Length ();	
	s1 = < 12 21 58 > i = 3

Set

Motivation, Applicability, and Indications for Use

A programming type *Set* is the direct counterpart of a mathematical set (which is its mathematical model). You can create a *Set* object whose elements are of some particular type, add and remove elements, check membership of an element, and determine the cardinality (size, or number of elements) the *Set* contains.

Related Components

- *Partial_Map* — a type that is similar to *Set* except that it is modeled by a set of ordered pairs that must satisfy the function property (i.e., there must be at most one pair in the set with any given value of the first component of the pair)

Component Family Members

Abstract Components

- *Set_Kernel* — the programming type of interest, with the operations below
 - *Add*
 - *Remove*
 - *Remove_Any*
 - *Is_Member*
 - *Size*

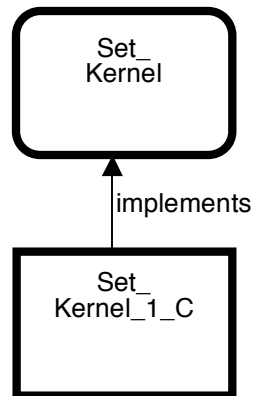
Concrete Components

- *Set_Kernel_1_C* — This is a checking implementation of *Set_Kernel* in which the execution time for each of the operations constructor, *Remove_Any*, and *Size* is constant, while the execution time for each of the operations destructor, *Add*, *Remove*, and *Is_Member* is proportional to the size of the set. All objects of this type have the interface of *Set_Kernel*, with the concrete template name *Set_Kernel_1_C* substituted for the abstract template name *Set_Kernel*.

To bring this component into the context you write:

```
#include "CT/Set/Kernel_1_C.h"
```

Component Coupling Diagram



Descriptions

Set_Kernel Type and Standard Operations

Formal Contract Specification

```

standard_abstract_operations (Set_Kernel);
/*!
    Set_Kernel is modeled by finite set of Item
    initialization ensures
        self = empty_set
!*/

```

Informal Description

The model for *Set_Kernel* is a finite set of items which is initially empty. A *Set_Kernel* object may be arbitrarily large (but still finite). Like all Resolve/C++ types, *Set_Kernel* comes with operator `&=` and *Clear*.

Notice that the properties of mathematical sets include that the elements in a set are unordered, and that there are no duplicate elements in a set.

Note — The sample traces for this and the other operation descriptions refer to the type *Set_Of_Integer*, which is the result of the following instantiation:

```

concrete_instance
class Set_Of_Integer :
    instantiates
        Set_Kernel_1_C <Integer>
{};

```

Sample Traces

Statement	Object Values
	(s1 is not in scope)
object Set_Of_Integer s1;	
	s1 = { }

Add

Formal Contract Specification

```

procedure Add (
    consumes Item& x
) is_abstract;
/*!
    requires
        x is not in self
    ensures
        self = #self union {#x}
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Set_Of_Integer s1;
object Boolean b;
object Integer i, n;

```

A call to the *Add* operation has the form:

```
s1.Add (i);
```

This operation adds the element whose value is *i* to set *s1*. You may make this call only if *i* is not already in *s1*.

Sample Traces

Statement	Object Values
	s1 = {6, 92, 18} i = 13
s1.Add (i);	
	s1 = {6, 92, 13, 18} i = 0

Remove

Formal Contract Specification

```

procedure Remove (
    preserves Item& x,
    produces Item& x_copy
) is_abstract;
/*!
    requires
        x is in self
    ensures
        self = #self - {x}  and
        x_copy = x
!*/

```

Informal Description

A call to the *Remove* operation has the form:

```
s1.Remove (i, n);
```

This operation removes from set *s1* the element whose value is *i*, and returns with *n* having that value as well. You may make this call only if *i* is in *s1*.

Sample Traces

Statement	Object Values
	s1 = {6, 92, 13, 18} i = 92 n = 847
s1.Remove (i, n);	
	s1 = {6, 13, 18} i = 92 n = 92
i = 18;	
	s1 = {6, 13, 18} i = 18 n = 92
s1.Remove (i, n);	
	s1 = {6, 13} i = 18 n = 18

Remove_Any

Formal Contract Specification

```

procedure Remove_Any (
    produces Item& x
) is_abstract;
/*!
    requires
        self /= empty_set
    ensures
        x is in #self and
        self = #self - {x}
!*/

```

Informal Description

A call to the *Remove_Any* operation has the form:

```
s1.Remove_Any (i);
```

This operation removes from set *s1* some — any — element, and returns with *i* having that item's value. You may make this call only if *s1* is non-empty.

Notice that this operation does not have functional (deterministic) behavior, i.e., even if it is called twice with exactly the same value of *s1* it might return with different results, since it is permitted to remove *any* element from *s*. This is illustrated in the tracing table below.

Sample Traces

Statement	Object Values
	s1 = {6, 92, 13, 18} i = -379
s1.Remove_Any (i);	
	s1 = {6, 92, 18} i = 13
...	
	s1 = {6, 92, 13, 18} i = -379
s1.Remove_Any (i);	
	s1 = {6, 13, 18} i = 92

Is_Member

Formal Contract Specification

```

function Boolean Is_Member (
    preserves Item& x
) is_abstract;
/*!
    ensures
        Is_Member = (x is in self)
!*/

```

Informal Description

A call to the *Is_Member* operation has the form of an expression:

```
... s1.Is_Member (i) ...;
```

This operation returns a *Boolean* value which is **true** iff an element with value *i* is in set *s1*.

Sample Traces

Statement	Object Values
	s1 = {6, 92, 13, 18} i = 59 b = true
b = s1.Is_Member (i);	
	s1 = {6, 92, 13, 18} i = 59 b = false
b = s1.Is_Member (92);	
	s1 = {6, 92, 13, 18} i = 59 b = true

Size

Formal Contract Specification

```

function Integer Size () is_abstract;
/*!
    ensures
        Size = |self|
!*/

```

Informal Description

A call to the *Size* operation has the form of an expression:

```
... s1.Size () ...;
```

This operation returns an *Integer* value which is the size of (cardinality of, or number of elements in) set *s1* , i.e., $|s1|$.

Sample Traces

Statement	Object Values
	$s1 = \{6, 92, 13, 18\}$ $i = -68$
$i = s1.Size () ;$	
	$s1 = \{6, 92, 13, 18\}$ $i = 4$

Sorting_Machine

Motivation, Applicability, and Indications for Use

One of the more common requirements of a software system is to process or list some data in “sorted” order. The data may be of any type (e.g., *Text*), and the ordering may be anything (e.g., reverse alphabetical order).

A typical approach to meeting this requirement is to put the data in some collection object—perhaps an *Array* or *Queue* or *Sequence* or *Static_Array*—and then to pass that object to a procedure operation that rearranges the entries into the desired order. This conventional way of doing things has at least two serious disadvantages:

- You have to write a different implementation of sorting for each collection type, even if you can parameterize it by the type of data in the collection and the ordering. This leads to a proliferation of implementations of the same sorting algorithm, which differ only in how they access and manipulate the data in the collection. It would be much better if sorting could be done without regard for the type of collection that holds the data to be sorted.
- The design is inefficient if you don’t really need all the data in sorted order. For instance, if you need to find the top ten reasons (out of 1000) that software component engineering is important, then you can do it by sorting all 1000 reasons in decreasing order of importance and looking at just the first ten. But there is no need to have sorted the last 990 into order, and all the time spent doing that is simply wasted. It would be much better to be able to pay only for work actually needed.

The component family *Sorting_Machine* allows you to collect and later process items in sorted order. It addresses both problems noted above. A *Sorting_Machine* object is a collection, just like a *Queue* or a *Stack*. The main difference is that with a *Queue* you put items in and get them back in first-in-first-out order; with a *Stack* you put items in and get them back in last-in-first-out order; with a *Sorting_Machine* you put items in and get them back in sorted order. So—no surprise—there are operations to insert items and remove items as well as to report the number of items in the object.

An important difference is that *Sorting_Machine* has “two-phase” behavior. Initially, a *Sorting_Machine* object is in the insertion phase, which means it is ready for you to insert some items. Once you have inserted all the items of interest, you explicitly tell the object to change to extraction phase. At this point you can remove items as many items as you want, one at a time, according to the ordering of interest. For completeness, there is also an operation to remove any (arbitrary) item, which you can invoke in either phase. This operation ignores the ordering among entries.

Related Components

- *Queue* — a type that is similar to *Sorting_Machine* except that when you remove an item it is not the next one in sorted order, but the first one that was put in (i.e., it observes a temporal, not a value-based, ordering)
- *Stack* — a type that is similar to *Sorting_Machine* except that when you remove an item it is not the next one in sorted order, but the last one that was put in (i.e., it observes a different temporal, not a value-based, ordering)

Component Family Members

Abstract Components

- *Sorting_Machine_Kernel* — the programming type of interest, with the operations below
 - *Insert*
 - *Remove_First*
 - *Remove_Any*
 - *Size*
 - *Change_To_Extraction_Phase*
 - *Is_In_Extraction_Phase*

Concrete Components

- *Sorting_Machine_Kernel_1_C* — This is a checking implementation of *Sorting_Machine_Kernel* in which the execution time for each of the operations constructor, *Insert*, *Remove_Any*, *Size*, *Change_To_Extraction_Phase*, and *Is_In_Extraction_Phase* is constant; while the execution time for *Remove_First* and for the destructor are proportional to the number of items in the object. All objects of this type have the interface of *Sorting_Machine_Kernel*, with the concrete template name *Sorting_Machine_Kernel_1_C* substituted for the abstract template name *Sorting_Machine_Kernel*.

To bring this component into the context you write:

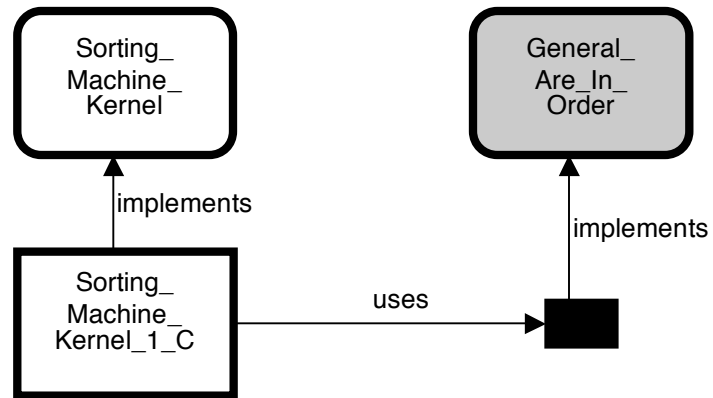
```
#include "CT/Sorting_Machine/Kernel_1_C.h"
```

You also need to have in scope a utility class that implements the abstract template utility class *General_Are_In_Order* for the type of *Item* you want to use, which determines the sorting order. For example, to sort *Text* values in non-decreasing lexicographic order, there is a catalog component already available: *Text_Are_In_Order_1*. Before instantiating *Sorting_Machine_Kernel_1_C* as in the *Sorting_Machine_Kernel* description below, you must bring this component into scope as follows:

```
#include "CI/Text/Are_In_Order_1.h"
```

There are a few other available components of this kind for the built-in types, but often you'll have to build your own utility class for a new type and/or a new ordering, by following these as examples.

Component Coupling Diagram



Descriptions

Sorting_Machine_Kernel Type and Standard Operations

Formal Contract Specification

```

/*!
  math subtype SORTING_MACHINE_MODEL is (
    inserting: boolean
    contents: finite multiset of Item
  )
!*/

standard_abstract_operations (Sorting_Machine_Kernel);
/*!
  Sorting_Machine_Kernel is modeled by SORTING_MACHINE_MODEL
  initialization ensures
    self = (true, empty_multiset)
!*/

```

Informal Description

The model for *Sorting_Machine_Kernel* is an ordered pair: a boolean that records the phase of the object (true if the machine is in insertion phase, false if it is in extraction phase), and a finite multiset¹ of *Items*. Initially the machine is in insertion phase and the multiset is empty.

A *Sorting_Machine_Kernel* object may be arbitrarily large. Like all Resolve/C++ types, *Sorting_Machine_Kernel* comes with operator `&=` and *Clear*.

Note — The sample traces for this and the other operation descriptions refer to the type *Sorting_Machine_Of_Text*, which is the result of the following instantiation:

```

concrete_instance
class Sorting_Machine_Of_Text :
  instantiates
    Sorting_Machine_Kernel_1_C <Text, Text_Are_In_Order_1>
{};

```

The component *Text_Are_In_Order_1* is a concrete instance utility class that implements *General_Are_In_Order* <*Text*>. This means that it exports a *Boolean*-valued function operation *Are_In_Order* which takes two *Text* objects as arguments and returns true iff the first and second arguments are in lexicographic order according to ASCII character ordering.

Text_Are_In_Order_1 also gives a realization-supplied mathematical definition for *ARE_IN_ORDER*, which formally describes this ordering.

For example, "aardvark" and "zebra" are in order, i.e., *ARE_IN_ORDER* ("aardvark", "zebra") is true. This means the call *Text_Are_In_Order_1::Are_In_Order* ("aardvark", "zebra") returns true. Similarly, *ARE_IN_ORDER* ("zebra", "zebras") is true, but not *ARE_IN_ORDER* ("bird", "ape"). Note that *Text_Are_In_Order_1::ARE_IN_ORDER* has the following properties:

¹ A *multiset* is just like a set except that it can have many “copies” of any value.

- Any string of characters is in order with itself.
- In ASCII, all the upper-case characters come before any of the lower-case characters. This means, for example, that *ARE_IN_ORDER* ("ZZZ", "an").

The description of *Remove_First* explains how the second template parameter of *Sorting_Machine_Kernel_1_C* is used.

Sample Traces

Statement	Object Values
	(s1 is not in scope)
object Sorting_Machine_Of_Text s1;	
	s1 = (true, {})

Insert

Formal Contract Specification

```

procedure Insert (
    consumes Item& x
) is_abstract;
/*!
    requires
        self.inserting = true
    ensures
        self = (true, #self.contents union {#x})
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Sorting_Machine_Of_Text s1;
object Text t;
object Integer i;
object Boolean b;

```

A call to the *Insert* operation has the form:

```
s1.Insert (t);
```

This operation adds the item whose value is *t* to the contents of *s1*. You may make this call only if *s1* is in insertion phase.

Sample Traces

Statement	Object Values
	s1 = (true, {"cse"}) t = "ece"
s1.Insert (t);	
	s1 = (true, {"ece", "cse"}) t = ""

Remove_First

Formal Contract Specification

```

    /*!
      math operation IS_FIRST (
        s: multiset of Item,
        x: Item
      ): boolean is
        x is in s and
        for all y: Item where (y is in s)
          (ARE_IN_ORDER (x, y))
    !*/

    procedure Remove_First (
      produces Item& x
    ) is_abstract;
    /*!
      requires
        self.inserting = false and
        self.contents /= empty_multiset
      ensures
        IS_FIRST (self.contents, x) and
        self = (false, #self.contents - {x})
    !*/

```

Informal Description

A call to the *Remove_First* operation has the form:

```
s1.Remove_First (t);
```

This operation removes from *s1* a first item (in general, this is not unique, though it is in the example) according to the *ARE_IN_ORDER* definition supplied by the utility class *Item_Are_In_Order*, and returns its value in *t*. You may make this call only if *s1* is not in insertion phase and is not empty.

Sample Traces

Statement	Object Values
	s1 = (false, {"ece", "cse"}) t = "abracadabra"
s1.Remove_First (t);	
	s1 = (false, {"ece"}) t = "cse"
s1.Remove_First (t);	
	s1 = (false, {}) t = "ece"

Remove_Any

Formal Contract Specification

```

procedure Remove_Any (
    produces Item& x
) is_abstract;
/*!
    requires
        self.contents /= empty_multiset
    ensures
        x is in #self.contents and
        self = (#self.inserting, #self.contents - {x})
!*/

```

Informal Description

A call to the *Remove_Any* operation has the form:

```
s1.Remove_Any (t);
```

This operation removes from *s1* some (i.e., any arbitrary) item and returns its value in *t*. You may make this call only if *s1* is not empty; it doesn't matter whether *s1* is in insertion phase or extraction phase.

Notice that this operation may remove a different item from *s1* if called twice in identical states, as illustrated in the sample tracing table.

Sample Traces

Statement	Object Values
	s1 = (false, {"ece", "cse"}) t = "abracadabra"
s1.Remove_Any (t);	
	s1 = (false, {"cse"}) t = "ece"
...	
	s1 = (false, {"ece", "cse"}) t = "abracadabra"
s1.Remove_Any (t);	
	s1 = (false, {"ece"}) t = "cse"

Size

Formal Contract Specification

```

function Integer Size () is_abstract;
  /*!
    ensures
      Size = |self.contents|
  !*/

```

Informal Description

A call to the *Size* operation has the form of an expression:

```
... s1.Size () ...;
```

This operation returns an *Integer* value which is the number of elements of the multiset *s1.contents* (including copies of a single value as many times as it occurs), i.e., *ls1.contents*l.

Sample Traces

Statement	Object Values
	s1 = (true, {"ece", "cse"}) i = -68
i = s1.Size ();	
	s1 = (true, {"ece", "cse"}) i = 2

Change_To_Extraction_Phase

Formal Contract Specification

```

procedure Change_To_Extraction_Phase () is_abstract;
/*!
    requires
        self.inserting = true
    ensures
        self = (false, #self.contents)
!*/

```

Informal Description

A call to the *Change_To_Extraction_Phase* operation has the form:

```
s1.Change_To_Extraction_Phase ();
```

This operation changes *s1* from insertion phase to extraction phase. You may make this call only if *s1* is in insertion phase.

Sample Traces

Statement	Object Values
	s1 = (true, {"ece", "cse"})
s1.Change_To_Extraction_Phase ();	
	s1 = (false, {"ece", "cse"})

Is_In_Extraction_Phase

Formal Contract Specification

```

function Boolean Is_In_Extraction_Phase () is_abstract;
/*!
    ensures
        Is_In_Extraction_Phase = not self.inserting
!*/

```

Informal Description

A call to the *Is_In_Extraction_Phase* operation has the form of an expression:

```
... s1.Is_In_Extraction_Phase () ...;
```

This operation returns a *Boolean* value which is true iff *s1* is in extraction phase.

Sample Traces

Statement	Object Values
	s1 = (true, {"ece", "cse"}) b = true
b = s1.Is_In_Extraction_Phase ();	
	s1 = (true, {"ece", "cse"}) b = false

Stack

Motivation, Applicability, and Indications for Use

The programming type *Stack* allows you to collect and later process items in last-in-first-out (“LIFO”) order. For example, suppose you need to keep track of the locations where a mouse has been while exploring a maze—sort of like bread crumbs that it leaves on the ground so it can retrace its path to look for unexplored corridors. To do this, you can create a *Stack* object whose elements are of a type that records the location of a single bread crumb. Then you can push its location whenever the mouse drops a bread crumb, and pop a location to determine where the mouse should go whenever it wants to back up. Other operations allow you to access the top item in a *Stack* object and to determine its length.

Related Components

- *Queue* — a type that is similar to *Stack* except that when you remove an item it is not the last one that was put in, but rather the first one
- *Sorting_Machine* — a type that is similar to *Stack* except that when you remove an item it is not the last one that was put in, but the next one in “sorted” order according to some ordering based on the values of the items

Component Family Members

Abstract Components

- *Stack_Kernel* — the programming type of interest, with the operations below
 - *Push*
 - *Pop*
 - *Accessor*
 - *Length*

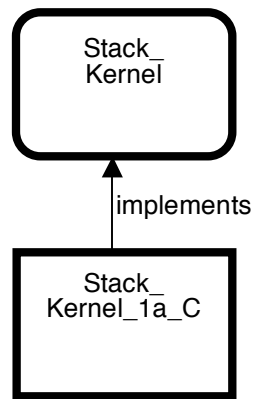
Concrete Components

- *Stack_Kernel_1a_C* — This is a checking implementation of *Stack_Kernel* in which the execution time for each of the operations constructor, *Push*, *Pop*, the accessor, and *Length* is constant, while the execution time for the destructor is proportional to the length of the string. All objects of this type have the interface of *Stack_Kernel*, with the concrete template name *Stack_Kernel_1a_C* substituted for the abstract template name *Stack_Kernel*.

To bring this component into the context you write:

```
#include "CT/Stack/Kernel_1a_C.h"
```

Component Coupling Diagram



Descriptions

Stack_Kernel Type and Standard Operations

Formal Contract Specification

```

standard_abstract_operations (Stack_Kernel);
/*!
    Stack_Kernel is modeled by string of Item
    initialization ensures
        self = empty_string
!*/

```

Informal Description

The model for *Stack_Kernel* is a string of items which is initially empty. A *Stack_Kernel* object may be arbitrarily large. Like all Resolve/C++ types, *Stack_Kernel* comes with operator `&=` and *Clear*.

Note — The sample traces for this and the other operation descriptions refer to the type *Stack_Of_Integer*, which is the result of the following instantiation:

```

concrete_instance
class Stack_Of_Integer :
    instantiates
        Stack_Kernel_1a_C <Integer>
{};

```

Sample Traces

Statement	Object Values
	(s1 is not in scope)
object Stack_Of_Integer s1;	
	s1 = < >

Push

Formal Contract Specification

```

procedure Push (
    consumes Item& x
) is_abstract;
/*!
    ensures
        self = <#x> * #self
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Stack_Of_Integer s1, s2;
object Integer i;

```

A call to the *Push* operation has the form:

```
s1.Push (i);
```

This operation adds the item whose value is *i* to the “top” (left end) of *s1*.

Sample Traces

Statement	Object Values
	s1 = < 58, 12, 21 > i = 13
s1.Push (i);	
	s1 = < 13, 58, 12, 21 > i = 0

Pop

Formal Contract Specification

```

procedure Pop (
    produces Item& x
) is_abstract;
/*!
    requires
        self /= empty_string
    ensures
        #self = <x> * self
!*/

```

Informal Description

A call to the *Pop* operation has the form:

```
s1.Pop (i);
```

This operation removes the item from the “top” (left end) of *s1* and returns its value in *i*. You may make this call only if *s1* is not empty.

Sample Traces

Statement	Object Values
	s1 = < 13, 58, 12, 21 > i = 92
s1.Pop (i);	
	s1 = < 58, 12, 21 > i = 13

Accessor

Formal Contract Specification

```
function Item& operator [] (
    preserves Accessor_Position& current
) is_abstract;
/*!
    requires
        self /= empty_string
    ensures
        there exists a: string of Item
            (self = <self[current]> * a)
!*/
```

Informal Description

A call to the accessor operator [] has the form of an expression:

```
... s1[current] ...;
```

This expression acts as the name of an object of type *Item* whose value is the item at the top of *s1*. You may make this call only if *s1* is not empty.

The special word *current* is the only thing that can appear between [] in the accessor for type *Stack*. It always refers to the single item that you can access directly without popping any others: the top (leftmost) item, which is the one that would be removed if you called the *Pop* operation at the point where you are using the accessor operator. You may use *s1[current]* wherever any other object of type *Item* may appear.

The accessor operator [], like all functions, preserves its arguments. But it is important to realize that the accessor expression *s1[current]* does not simply denote the value of the top item in *s1*, it acts as the name of an object of type *Item* which you may consider to lie at the left end of the string *s1*. This means that not only may you use the expression *s1[current]* as a value of type *Item*; you may even change the value of the object called *s1[current]*—but remember that this also changes the value of *s1*.

The sample traces help illustrate some important points regarding the convenience and danger of accessor expressions. The first and second sample statements show how an accessor expression such as *s1[current]* acts like an object of type *Item* whose value may be changed, depending on the statement in which the expression occurs. The third sample statement shows that *s1[current]* may appear as an argument in a call where an *Item* is required. This call involves other arguments (e.g., *s2*), but none of the other arguments may be *s1* or may involve an accessor expression for *s1* because this would violate the repeated argument rule. So, while you might be tempted to write something like:

```
s1.Push (s1[current]);
```

this statement violates the repeated argument rule. Be aware that the C++ compiler will not report this as an error. You need to avoid the pitfall by personal discipline.

The last statement in the tracing table also illustrates that an accessor expression acts exactly like an object of type *Item*. Notice that *s1[current]*, which is being pushed onto *s2* in this statement, is consumed (its new value is 0 since *Item* is *Integer* here) because *Push* consumes its argument. So this statement changes both *s1* and *s2*.

Sample Traces

Statement	Object Values
	s1 = < 92, 6, 18 > i = 37
s1[current] &= i;	
	s1 = < 37, 6, 18 > i = 92
s1[current] = i;	
	s1 = < 92, 6, 18 > i = 92
...	
	s1 = < 92, 255, 6 > s2 = < 76, 921, 34, 7 >
s2.Push (s1[current]);	
	s1 = < 0, 255, 6 > s2 = < 92, 76, 921, 34, 7 >

Length

Formal Contract Specification

```

function Integer Length () is_abstract;
  /*!
    ensures
      Length = |self|
  !*/

```

Informal Description

A call to the *Length* operation has the form of an expression:

```
... s1.Length () ...;
```

This operation returns an *Integer* value which is the length of the string *qI*, i.e., $|qI|$.

Sample Traces

Statement	Object Values
	s1 = < 6, 92, 13, 18 > i = -68
i = s1.Length ();	
	s1 = < 6, 92, 13, 18 > i = 4

Static_Array

Motivation, Applicability, and Indications for Use

The programming type *Static_Array* allows you to repeatedly access and update a fixed-size table of items of an arbitrary type. You “index” into the table using an *Integer* value that lies within an interval determined when you instantiate a concrete template that implements *Static_Array_Kernel*.

For example, suppose you want to record the high temperature for each month of the year. To do this, you can declare an *Static_Array* object whose items are of type *Real* and set it up to be indexed by the values 1 through 12 (for the months of the year). Each entry in the table initially has an initial value for type *Real*, i.e., 0.0. You may access and update the value associated with each month by using that month’s index (an integer between 1 and 12).

The most closely related component is *Array*, which is nearly identical except in regard to when the index bounds are fixed. Another closely related component is *Sequence*. A *Sequence* object starts out as an empty string of *Item* values, so if you want to use it as a table in the fashion mentioned above you have to add entries to it one by one until the table is populated with values. A *Static_Array* object becomes populated with initial values of type *Item* as soon as you declare it. With a *Sequence* object you can interleave operations that change the size of the *Sequence* with operations that access and update existing entries, possibly deferring the decision to add more entries or to remove some; however, adding and removing changes the indices of some of the existing entries. With the *Static_Array* object you can’t change the bounds of the table; you decide up front on the required size and bounds, and then you access and update table entries whose indices remained fixed throughout. The *Sequence* is always indexed starting with 0. The *Static_Array* object may be set up to be indexed starting with any *Integer* value.

Related Components

- *Array* — a type that is essentially identical to *Static_Array* except that the index bounds are fixed not at template instantiation time, but at run time
- *Sequence* — a type that is similar to *Static_Array* except that it is incrementally resizable, and that it is always indexed from 0

Component Family Members

Abstract Components

- *Static_Array_Kernel* — the programming type of interest, with the operations below
 - *Accessor*
 - *Lower_Bound*
 - *Upper_Bound*

Concrete Components

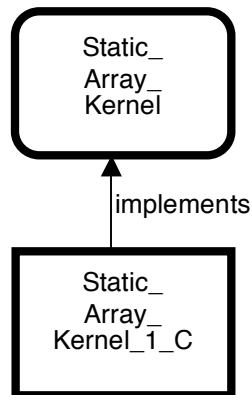
- *Static_Array_Kernel_I_C* — This is a checking implementation of *Static_Array_Kernel* in which the execution time for each of the operations *accessor*, *Lower_Bound*, and

Upper_Bound is constant; the execution times for the constructor and destructor are proportional to the length of the interval between the upper and lower bounds. All objects of this type have the interface of *Static_Array_Kernel*, with the concrete template name *Static_Array_Kernel_1_C* substituted for the abstract template name *Static_Array_Kernel*.

To bring this component into the context you write:

```
#include "CT/Static_Array/Kernel_1_C.h"
```

Component Coupling Diagram



Descriptions

Static_Array_Kernel Type and Standard Operations

Formal Contract Specification

```

    /*!
      math subtype INDEXED_TABLE is finite set of (
        i: integer,
        x: Item
      )
      exemplar t
      constraint
        for all i1, i2: integer, x1, x2: Item
          where ((i1,x1) is in t and
                (i2,x2) is in t)
            (if i1 = i2 then x1 = x2)

      math subtype STATIC_ARRAY_MODEL is INDEXED_TABLE
      exemplar a
      constraint
        for all i: integer
          (there exists x: Item
            ((i,x) is in a) iff lower <= i <= upper))
    !*/

    standard_abstract_operations (Static_Array_Kernel);
    /*!
      Static_Array_Kernel is modeled by STATIC_ARRAY_MODEL
      initialization ensures
        self = {i: integer, x: Item
              where (lower <= i <= upper and
                    is_initial (x))
              (i,x)}
    !*/

```

Informal Description

The model for *Static_Array_Kernel* is a finite set of ordered pairs of type (integer, *Item*) that contains exactly one pair for each integer within the interval from *lower* to *upper*, inclusive. These bounds are template parameters that are just ordinary integer constants once *Static_Array_Kernel* is instantiated.

Like all Resolve/C++ types, *Static_Array_Kernel* comes with operator *&=* and *Clear*.

Note — The sample traces for this and the other operation descriptions refer to the type *Static_Array_Of_Text*, which is the result of the following instantiation:

```

concrete_instance
class Static_Array_Of_Text :
  instantiates
    Static_Array_Kernel_1_C <Text, 7, 9>
{};

```

Sample Traces

Statement	Object Values
	(a1 is not in scope)
object Static_Array_Of_Text a1;	
	a1 = { (7, ""), (8, ""), (9, "") }

Accessor

Formal Contract Specification

```
function Item& operator [] (
    preserves Integer i
) is_abstract;
/*!
    requires
        lower <= i <= upper
    ensures
        (i, self[i]) is in self
!*/
```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```
object Static_Array_Of_Text a1, a2;
object Text t;
object Integer i1, i2;
```

A call to the accessor operator [] has the form of an expression:

```
... a1[i1] ...;
```

This expression acts as the name of an object of type *Item* (i.e., *Text* in this case) whose value is that paired with *i1* in *a1*. You may make this call only if there is a pair in *a1* whose first component is *i1*, i.e., only if *i1* lies between *lower* and *upper* inclusive.

You may use *a1[i1]* wherever any other object of type *Item* (i.e., *Text* in this case) may appear.

The accessor operator [], like all functions, preserves its arguments. But it is important to realize that the accessor expression *a1[i1]* does not simply denote the value associated with *i1* in *a1*, it acts as the name of an object of type *Item* which you may consider to be in that particular pair in *a1*. This means that not only may you use the expression *a1[i1]* as a value of type *Item*; you may even change the value of the object called *a1[i1]*—but remember that this also changes the value of *a1*.

The sample traces help illustrate some important points regarding the convenience and danger of accessor expressions. The first and second sample statements show how an accessor expression such as *a1[i1]* acts like an object of type *Item* whose value may be changed, depending on the statement in which the expression occurs. The third sample statement's call to the swap operator involves other arguments (e.g., *a2*), but none of the other arguments may be *a1* or may involve an accessor expression for *a1* because this would violate the repeated argument rule. So, while you might be tempted to write something like:

```
a1[i1] &= a1[i2];
```

this statement violates the repeated argument rule. Be aware that the C++ compiler will not report this as an error. You need to avoid the pitfall by personal discipline.

Sample Traces

Statement	Object Values
	<pre> a1 = { (7, "abcd"), (8, "XYZ"), (9, "bucks") } i1 = 8 t = "go" </pre>
<code>a1[i1] &= t;</code>	
	<pre> a1 = { (7, "abcd"), (8, "go"), (9, "bucks") } i1 = 8 t = "XYZ" </pre>
<code>a1[8] = t;</code>	
	<pre> a1 = { (7, "abcd"), (8, "XYZ"), (9, "bucks") } i1 = 8 t = "XYZ" </pre>
<code>...</code>	
	<pre> a1 = { (7, "abcd"), (8, "go"), (9, "bucks") } a2 = { (7, ""), (8, "cd"), (9, "rom") } i1 = 8 i2 = 9 </pre>
<code>a1[i1] &= a2[i2];</code>	
	<pre> a1 = { (7, "abcd"), (8, "rom"), (9, "bucks") } a2 = { (7, ""), (8, "cd"), (9, "go") } i1 = 8 i2 = 9 </pre>

Lower_Bound

Formal Contract Specification

```

function Integer Lower_Bound () is_abstract;
/*!
    ensures
        Lower_Bound = lower
!*/

```

Informal Description

A call to the *Lower_Bound* operation has the form of an expression:

```
... a1.Lower_Bound () ...;
```

This operation returns an *Integer* value which is the lower bound on the interval of legal indices into *a1*, i.e., *lower*.

Sample Traces

Statement	Object Values
	<pre> a1 = {(7,"abcd"), (8,"go"), (9,"bucks")} i1 = -32 </pre>
<code>i1 = a1.Lower_Bound ();</code>	
	<pre> a1 = {(7,"abcd"), (8,"go"), (9,"bucks")} i1 = 7 </pre>

Upper_Bound

Formal Contract Specification

```

function Integer Upper_Bound () is_abstract;
/*!
    ensures
        Upper_Bound = upper
!*/

```

Informal Description

A call to the *Upper_Bound* operation has the form of an expression:

```
... a1.Upper_Bound () ...;
```

This operation returns an *Integer* value which is the upper bound on the interval of legal indices into *a1*, i.e., *upper*.

Sample Traces

Statement	Object Values
	<pre> a1 = { (7, "abcd"), (8, "go"), (9, "bucks") } i1 = -32 </pre>
<code>i = a1.Upper_Bound ();</code>	
	<pre> a1 = { (7, "abcd"), (8, "go"), (9, "bucks") } i1 = 9 </pre>

Text

Text is a built-in type (concrete instance) in Resolve/C++, which is included when you bring `RESOLVE_Foundation.h` into the global context. The name *Text* is therefore unusual: it is the name of a concrete template even though it does not end in “_2” or “_1a” or the like.

Motivation, Applicability, and Indications for Use

Resolve/C++ also includes *Boolean*, *Character*, *Integer*, and *Real* as built-in types. An object of any of these types has a very simple mathematical model, i.e., a simple mathematical type: boolean, character, integer, and real, respectively. While *Boolean*, *Integer*, and *Real* objects clearly are useful in a variety of situations, *Character* objects usually seem less valuable. The reason is that normally you want to combine individual *Character* objects into sequences or strings of characters. If you need to deal with a sequence of characters as a unit, then you should use a *Text* object, whose mathematical model is a string of characters.

RESOLVE/C++ has literals (constants) of the built-in types *Boolean*, *Character*, *Integer*, and *Real* — e.g., `true`, `'a'`, `13`, and `3.14159`, respectively. There also are literals of type *Text*, e.g., `"hello there"`. A *Text* literal comprises zero or more *Character* literals enclosed in double-quotes, without single-quotes around the individual *Character* literals that make it up. You may use a *Text* literal any place a *Text* object is required as an argument to an operation if the associated formal parameter has preserves mode. This makes *Text* a very useful type when you need to prompt a user for input, label an output, construct a menu of choices, etc. Such prompts and labels may be *Text* literals in your source code.¹

Related Components

- *Character* — the type of each of the individual items in a *Text* object
- *Sequence* — a generalization of *Text* in which the individual items may be of any type

Component Family Members

Abstract Components

- *Text_Kernel* — the programming type of interest, with the operations below
 - *Add*
 - *Remove*
 - *Accessor*
 - *Length*

¹ For flexibility, you might store such *Text* literals in “resource files”. This allows a knowledgeable user to customize certain aspects of an application program (e.g., to adapt it to a foreign language) without having access to the source code. Resource files are not further discussed here, but they are conventional features of many programs.

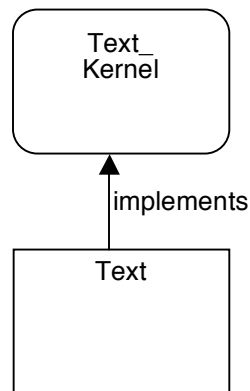
Concrete Components

- *Text* — This is the standard implementation of the abstract instance *Text_Kernel* that comes built-in to Resolve/C++. It is a concrete instance, i.e., you may declare objects of type *Text*. *Text* is a checking component, which means that it checks the preconditions of its operations. So, if you call any of the *Text* operations with a violated precondition then your program will immediately stop and report the violation.

To bring this component into the context you write:

```
#include "RESOLVE_Foundation.h"
```

Component Coupling Diagram



Descriptions

Text_Kernel Type and Standard Operations

Formal Contract Specification

```

standard_abstract_operations (Text_Kernel);
/*!
    Text_Kernel is modeled by string of character
    initialization ensures
        self = empty_string
!*/

```

Informal Description

The model for a *Text_Kernel* object is a string of characters which is initially empty. A *Text_Kernel* object may be an arbitrarily long string. Like all Resolve/C++ types, *Text_Kernel* comes with operator `&=` and *Clear*.

Sample Traces

Statement	Object Values
	(t1 is not in scope)
object Text t1;	
	t1 = ""

Add

Formal Contract Specification

```

procedure Add (
    preserves Integer pos,
    preserves Character c
) is_abstract;
/*!
    requires
        0 <= pos <= |self|
    ensures
        there exists a, b: string of character
            (|a| = pos and
             #self = a * b and
             self = a * <c> * b)
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Text t1, t2;
object Boolean b;
object Character c;
object Integer i;
object Character_IStream input;
object Character_OStream output;

```

A call to the *Add* operation has the form:

```
t1.Add (i, c);
```

This operation adds *c* in position *i* of string *t1*, i.e., in a position such that the number of characters before the newly added character is equal to *i*. You may make this call only if $0 \leq i \leq |t1|$.

You can see from this example that the numbering of positions in the string *t1* effectively starts at 0. For instance, to add 'X' at the beginning of *t1* (i.e., as the left-most character of *t1*) you might write:

```
t1.Add (0, 'X');
```


Sample Traces

Statement	Object Values
	t1 = "HeP" i = 2 c = 'L'
t1.Add (i, c);	
	t1 = "HeLp" i = 2 c = 'L'
t1.Add (4, '!');	
	t1 = "HeLp!" i = 2 c = 'L'

Remove

Formal Contract Specification

```

procedure Remove (
    preserves Integer pos,
    produces Character& c
) is_abstract;
/*!
    requires
        0 <= pos < |self|
    ensures
        there exists a, b: string of character
            (|a| = pos and
             #self = a * <c> * b and
             self = a * b)
!*/

```

Informal Description

A call to the *Remove* operation has the form:

```
t1.Remove (i, c);
```

This operation removes the character in position i of string $t1$ and returns it in c . You may make this call only if $0 \leq i < |t1|$.

The numbering of positions in the string $t1$ starts at 0. For example, to remove the left-most character of $t1$ and return it in c you might write:

```
t1.Remove (0, c);
```

Sample Traces

Statement	Object Values
	$t1 = \text{"HeLp"}$ $i = 2$ $c = \text{'z'}$
$t1.\text{Remove} (i, c);$	
	$t1 = \text{"Hep"}$ $i = 2$ $c = \text{'L'}$
$t1.\text{Remove} (2, c);$	
	$t1 = \text{"He"}$ $i = 2$ $c = \text{'p'}$

Accessor

Formal Contract Specification

```
function Character& operator [] (
    preserves Integer pos
) is_abstract;
/*!
    requires
        0 <= pos < |self|
    ensures
        there exists a, b: string of character
            (|a| = pos and
             #self = a * <self[pos]> * b)
!*/
```

Informal Description

A call to the accessor operator [] has the form of an expression:

```
... t1[i] ...;
```

This expression acts as the name of an object of type *Character* whose value is the character in position i of string $t1$. You may make this call only if $0 \leq i < |t1|$.

The numbering of positions in the string $t1$ starts at 0. For example, to set the left-most character of $t1$ to 'X' you might write (assuming $|t1| \geq 1$):

```
t1[0] = 'X';
```

The accessor operator [], like all functions, preserves its arguments ($t1$ and i in the example). But it is important to realize that the accessor expression $t1[i]$ does not simply denote the value of the character in position i of string $t1$; it acts as the name of a *Character* object which you may consider to lie in that position of the string. This means that not only may you use the expression $t1[i]$ as a value of type *Character*; you may even change the value of the object called $t1[i]$ —but remember that this also changes the value of $t1$.

The sample traces help illustrate some important points regarding the convenience and danger of accessor expressions. The first and second sample statements show an accessor expression such as $t1[i]$ acts like an object of type *Character* whose value may be changed, depending on the statement in which the expression occurs. The third sample statement shows that $t1[i]$ may appear as an argument in a call where a *Character* value (object or constant) is required. This call involves other arguments (e.g., $t2$), but none of the other arguments may be $t1$ or may involve an accessor expression for $t1$ because this would violate the repeated argument restriction.

So, while you might be tempted to write something like:

```
t1[i] &= t1[24];
```

this statement violates the repeated argument rule for the swap operator. This conclusion is especially clear if $i = 24$ at the time the statement is executed, but it is true even if i has some other value since both swap operator arguments are accessor expressions for the same object, namely $t1$. Be aware that the C++ compiler will not report this as an error. You need to avoid the pitfall by personal discipline.

Sample Traces

Statement	Object Values
	t1 = "HpLw" i = 1 c = 'e'
t1[i] &= c;	
	t1 = "HeLw" i = 1 c = 'p'
t1[3] = c;	
	t1 = "HeLp" i = 1 c = 'p'
...	
	t1 = "HeLp" i = 1 t2 = "MoV"
t2.Add (3, t1[i]);	
	t1 = "HeLp" i = 1 t2 = "MoVe"

Length

Formal Contract Specification

```
function Integer Length () is_abstract;
/*!
    ensures
        Length = |self|
!*/
```

Informal Description

A call to the *Length* operation has the form of an expression:

```
... t1.Length () ...;
```

This operation returns an *Integer* value which is the length of string *t1* , i.e., *|t1|*.

Sample Traces

Statement	Object Values
	t1 = "HeLp" i = 1
i = t1.Length ();	
	t1 = "HeLp" i = 4

Tree

Motivation, Applicability, and Indications for Use

The programming type *Tree* is very common in language processing applications (e.g., parsers, interpreters, compilers, etc.). The mathematics used to model general trees is similar to that used to model binary trees (see *Binary_Tree*). Here we describe only those aspects that differ from the explanation of binary trees.

The tree in Figure 1 has a **root** (*e*), **interior** items or nodes (*h*, *k*, and *f*), and **leaves** (*c*, *g*, *b*, *a*, *j*, *l*, and *d*), just like a binary tree. But note that item *e* has three children, and item *k* has four. This is what makes this a tree and *not* a binary tree. In a tree, there is no limitation on the number of children any given item can have. An item can have 0, 1, 2, 3, ... or any finite number of children.

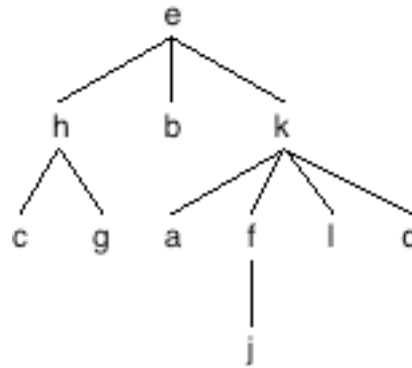


Figure 1: A tree *T*

Another important difference is that there is no empty tree. The smallest tree is a tree of size 1, i.e., a tree with exactly one item (the root) and no children.

Mathematically, we define “trees over a set of items”. This inductive definition is similar to the definition of binary trees. The set of “trees over set *A*” (or *tree of A*, which is the type name you use when writing the mathematics of trees) consists exactly of those trees that can be built up from the items in *A* by one or more applications of the **compose** function. The *compose* function for trees takes as arguments a root item from *A*, say *x*, and a string of *k* trees of *A*, say $\langle T1, T2, \dots, Tk \rangle$, and produces a tree of *A*, say *T*, with *x* as the root of *T*, and *T1*, *T2*, ..., *Tk* as the subtrees. This is pictured in Figure 2.

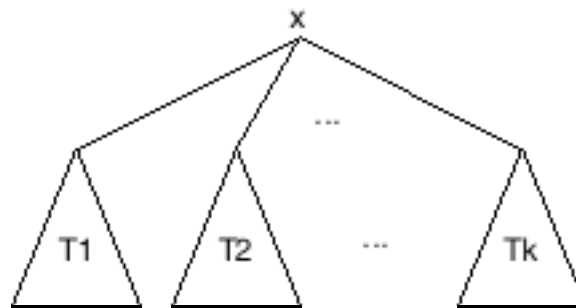


Figure 2: The tree $T = \text{compose}(x, \langle T1, T2, \dots, Tk \rangle)$

And here is the inductive definition of the set “tree over set *A*”:

- Base case: If x is an element of A , then $compose(x, empty_string)$ is an element of trees over set A .
- Inductive case: If x is an element of A , and for all $i: 1 \leq i \leq k$, T_i is an element of trees over set A , then $compose(x, \langle T_1, T_2, \dots, T_k \rangle)$ is an element of trees over set A .

In addition to the type *tree* and the operation *compose*, there are three other basic math operations you need to be familiar with. They are **size**, **root**, and **children**. Consider a tree T . Then $size(T)$ (also denoted $|T|$) is the number of items in T , e.g., in Figure 1, $size(T) = 11$; $root(T)$ is the root item of T , e.g., in Figure 2, $root(T) = x$; and $children(T)$ is the string of trees that are children of the root of T , e.g., in Figure 2, $children(T) = \langle T_1, T_2, \dots, T_k \rangle$.

Related Components

- *Binary_Tree* — a type that is similar to a *Tree* where each item has at most two children.

Component Family Members

Abstract Components

- *Tree_Kernel* — the programming type of interest, including the operations below
 - *Add*
 - *Remove*
 - *Accessor*
 - *Number_Of_Children*
 - *Size*

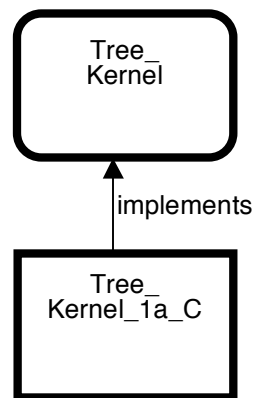
Concrete Components

- *Tree_Kernel_1a_C* — This is a checking implementation of *Tree_Kernel* in which the execution time for each of the operations constructor, *Number_Of_Children*, *Size*, and the accessor is constant, while the execution time for *Add* and *Remove* is proportional to the number of children of the root of the tree, and the execution time for the destructor is proportional to the size of the tree. All objects of this type have the interface of *Tree_Kernel*, with the concrete template name *Tree_Kernel_1a_C* substituted for the abstract template name *Tree_Kernel*.

To bring this component into the context you write:

```
#include "CT/Tree/Kernel_1a_C.h"
```


Component Coupling Diagram



Descriptions

Tree_Kernel Type and Standard Operations

Formal Contract Specification

```

standard_abstract_operations (Tree_Kernel);
/*!
    Tree_Kernel is modeled by tree of Item
    initialization ensures
        there exists x: Item
            (is_initial (x) and
              self = compose (x, empty_string))
!*/

```

Informal Description

The model for *Tree_Kernel* is a tree of items which initially has size 1. The root is an initial value for type *Item*, and the root has no children. A *Tree_Kernel* object may be arbitrarily large. Like all RESOLVE/C++ types, *Tree_Kernel* comes with operator *&=* and *Clear*.


Note — The sample traces for this and the other operation descriptions refer to the type *Tree_Of_Integer*, which is the result of the following instantiation:

```

concrete_instance
class Tree_Of_Integer :
    instantiates
        Tree_Kernel_1a_C <Integer>
{};

```

Sample Traces

Statement	Object Values
	(t1 is not in scope)
object Tree_Of_Integer t1;	
	t1 = 

Add

Formal Contract Specification

```

procedure Add (
    preserves Integer pos,
    consumes Subtree_Type& t
) is_abstract;
/*!
    requires
        0 <= pos and pos <= |children (self)|
    ensures
        there exists a, b: string of tree of Item
        (|a| = pos and
         children (#self) = a * b and
         self = compose (root (#self), a * <#t> * b))
!*/

```

Informal Description

Note — This and the other operation descriptions refer to these objects in examples:

```

object Tree_Of_Integer t1, t2, t3, t4;
object Integer i, n;

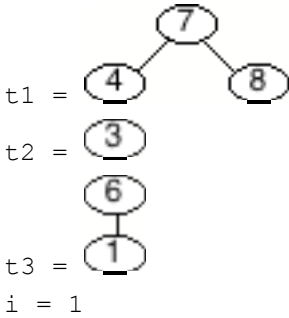
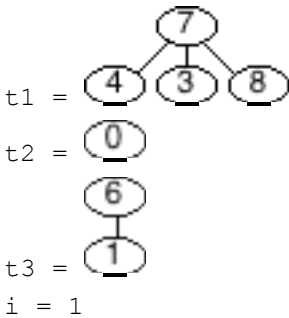
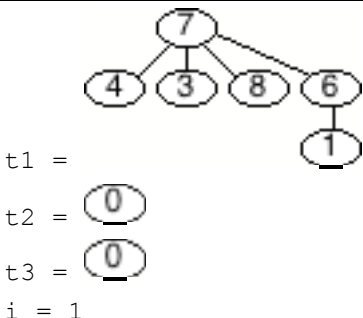
```

A call to the *Add* operation has the form:

```
t1.Add (i, t2);
```

This operation adds tree *t2* in position *i* of the string of children of the root of *t1*, i.e., in a position such that the number of children of the root of *t1* before the newly added one is equal to *i*. Notice that the value of *t2* is consumed, and that you may make this call only if $0 \leq i \leq |children(t1)|$, i.e., *i* is a valid position in the string of children of the root of *t1*.

Sample Traces

Statement	Object Values
	<div><p>t1 = 4 t2 = 3 t3 = 1 i = 1</p></div>
t1.Add (i, t2);	
	<div><p>t1 = 4 t2 = 0 t3 = 1 i = 1</p></div>
t1.Add (3, t3);	
	<div><p>t1 = 4 t2 = 0 t3 = 0 i = 1</p></div>

Remove

Formal Contract Specification

```

procedure Remove (
    preserves Integer pos,
    produces Subtree_Type& t
) is_abstract;
/*!
    requires
        0 <= pos and pos < |children (self)|
    ensures
        there exists a, b: string of tree of Item
            (|a| = pos and
             children (#self) = a * <t> * b and
             self = compose (root (#self), a * b))
!*/

```

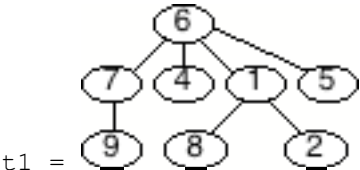
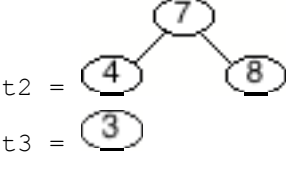
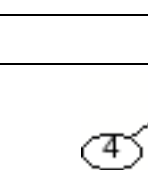
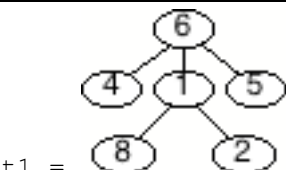
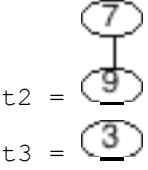

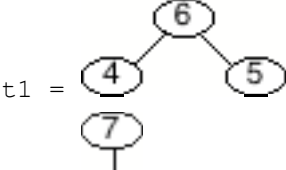
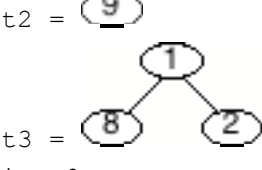
Informal Description

A call to the *Remove* operation has the form:

```
t1.Remove (i, t2);
```

This operation removes the tree in position i of the string of children of the root of $t1$, and returns it in $t2$. You may make this call only if $0 \leq i < |children(t1)|$, i.e., i is a valid position in the string of children of the root of $t1$.

Sample Traces

Statement	Object Values
	 <p>t1 = 9, t2 = 8, t3 = 2</p>  <p>t1 = 7, t2 = 4, t3 = 8</p>  <p>t1 = 7, t2 = 3, t3 = 3</p>
t1.Remove (i, t2);	
	 <p>t1 = 8, t2 = 2, t3 = 2</p>  <p>t1 = 7, t2 = 9, t3 = 9</p>  <p>t1 = 7, t2 = 3, t3 = 3</p>
t1.Remove (1, t3);	
	 <p>t1 = 4, t2 = 7, t3 = 9</p>  <p>t1 = 1, t2 = 8, t3 = 2</p>

Accessor

Formal Contract Specification

```

function Item& operator [ ] (
    preserves Accessor_Position& current
) is_abstract;
/*!
    ensures
        self = compose (self[current], children (self))
!*/

```

Informal Description

A call to the accessor operator [] has the form of an expression:

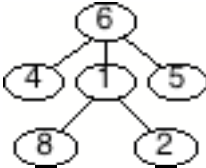
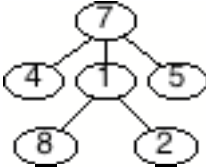
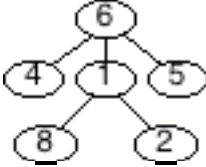
```
... t1[current] ...;
```

This expression acts as the name of an object of type *Item* whose value is the item at the root of *t1*.

The special word *current* is the only thing that can appear between [] in the accessor for type *Tree*. It always refers to the single item at the root. You may use *t1[current]* wherever any other object of type *Item* may appear.

The accessor operator [], like all functions, preserves its arguments. But it is important to realize that the accessor expression *t1[current]* does not simply denote the value of the root item in *t1*, it acts as the name of an object of type *Item* which you may consider to lie at the root of the tree *t1*. This means that not only may you use the expression *t1[current]* as a value of type *Item*; you may even change the value of the object called *t1[current]*—but remember that this also changes the value of *t1*. Note that using the accessor function is the only way to change the root in a tree.

Sample Traces

Statement	Object Values
	<div><p>t1 = i = 7</p></div>
t1[current] &= i;	
	<div><p>t1 = i = 6</p></div>
t1[current] = i;	
	<div><p>t1 = i = 6</p></div>

Number_Of_Children

Formal Contract Specification

```
function Integer Number_Of_Children () is_abstract;  
/*!  
    ensures  
        Number_Of_Children = |children (self)|  
!*/
```

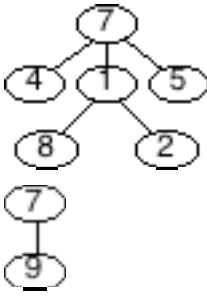
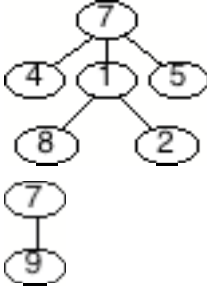
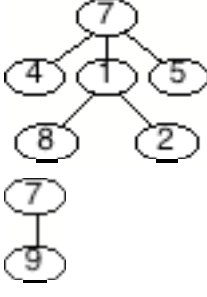
Informal Description

A call to the *Number_Of_Children* operation has the form of an expression:

```
... t1.Number_Of_Children () ...;
```

This operation returns an *Integer* value which is the length of the string of children of the root of *t1*.

Sample Traces

Statement	Object Values
	 t1 = t2 = i = 21
i = t1.Number_Of_Children ();	 t1 = t2 = i = 3
i = t2.Number_Of_Children ();	 t1 = t2 = i = 1

Size

Formal Contract Specification

```
function Integer Size () is_abstract;  
/*!  
    ensures  
        Size = |self|  
!*/
```

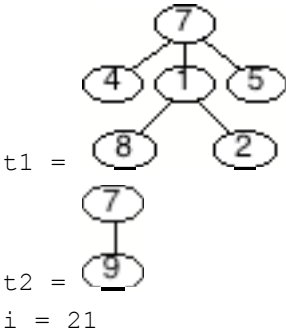
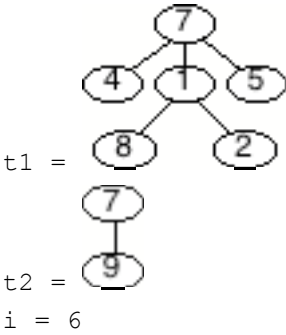
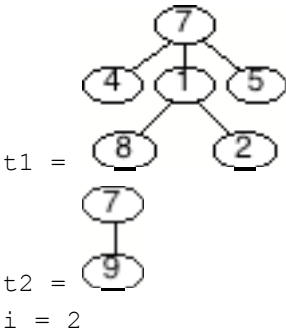
Informal Description

A call to the *Size* operation has the form of an expression:

```
... t1.Size () ...;
```

This operation returns an *Integer* value which is the size of the tree *t1*, i.e., $|t1|$.

Sample Traces

Statement	Object Values
	 <p>t1 = t2 = i = 21</p>
i = t1.Size ();	 <p>t1 = t2 = i = 6</p>
i = t2.Size ();	 <p>t1 = t2 = i = 2</p>