11 Exceptions

11.1 Exception handling requirements

A run-time error is called an *exception*. When programs were used only for offline computation, an appropriate response to an exception was simply to print an error message and terminate. However, the response to an exception in an interactive environment cannot be limited to notification, but must also include recovery to the point that the user can retry the computation or at least choose another option. Software that is used in embedded systems such as aircraft must perform error recovery without human intervention. Exception handling has not been routinely supported within programming languages until recently; rather, facilities supplied by the operating system have been used. This section will describe some of the exception handling mechanisms that exist in modern programming languages.

Error recovery does not come for free. There is always the cost of the extra data structures and algorithms needed to identify and handle the exceptions. In addition, it is often overlooked that exception handling code is itself a program component, and may contain bugs that can cause problems that are more serious than the original exception! It is also extremely difficult to identify error-causing situations and to test the exception handling code, because it may be difficult, if not impossible, to create the error situation.

What makes a good exception-handling facility?

- There should be very little overhead if no exception occurs.
- Exception handling should be both easy to use and safe.

The first requirement is more important than it may seem. Since we assume that exceptions generally *do not* occur, the overhead to the application should be minimal. Once an exception does occur, the overhead to process it is usually not an important consideration. The second requirement says that since exceptions occur infrequently, no great programming effort should be required to implement an exception handler; it goes without saying that the exception handler facility should not use constructs likely to cause bugs.

One warning to the programmer: an exception handler is *not* a replacement for an if-statement. If a situation is likely to arise, it is not an error and should be explicitly programmed. For example, a data structure such as a list or tree is very likely to be empty, and this situation should be explicitly checked using an if-statement:

Ada

```
if Ptr.Next = null then ... else ...
```

On the other hand, stack overflow or floating-point underflow are very rare and almost certainly indicate a bug in the computation.

Elementary exception handling is defined in some languages by allowing the user to define a block of code to be executed before program termination. This is useful for cleaning up (closing files, etc.) before exiting the program. In C, the setjmp/longjmp facility enables the user to define additional points within the program that the exception handler can return to. This type of exception handling is sufficient for ensuring that the program gracefully exits or restarts, but is not flexible enough to enable detailed exception handling to be programmed.

Note that according to our definition of an exception as an unexpected, run-time fault, there are fewer "exceptions" in C than in a language such as Ada. Firstly, errors such as exceeding array bounds are not defined in the C language; they are simply *bugs* that cannot be "handled". Secondly, since C does not have a flexible exception-handling facility, every language facility that is requested through a subprogram returns a code indicating if the request was successful or not. Thus an Ada memory allocation new can cause an exception if there is not enough memory, while in C, malloc returns a code that must be explicitly tested. The implications for programming style are: in Ada new can be used routinely and exception handling designed separately, while in C you should write an *envelope* subprogram around malloc so that exception handling can be centrally designed and programmed, rather than allowing each team member to test (or forget to test) for lack of memory:

C

```
void* get_memory(int n)
{
    void* p = malloc(n);
    if (p == 0)            /* Allocation failed */
        ...                /* Do something or abort gracefully */
    return p;
}
```

11.2 Exceptions in PL/I

PL/I was the first language to include a facility for exception handling within the language: the *on-unit*. An on-unit is a block of code that is executed when an exception occurs; upon termination of the on-unit, the computation is resumed. The problem with the PL/I on-unit is that it interferes with routine computation. Suppose that an on-unit for floating-point underflow is in force. Then *every* floating-point expression is potentially affected by the on-unit; more exactly, every floating-point expression includes an implicit call to and return from the on-unit. This makes it difficult to perform optimizations such as maintaining values in registers or extracting common subexpressions.

11.3 Exceptions in Ada

Ada defines a very elegant exception handling mechanism which has served as a model for other languages.

There are four predefined exceptions in Ada:

Constraint_Error Violation of a constraint such as an array indexing not in bounds or a variant field selection not matching the discriminant.

Storage_Error Not enough memory.

Program_Error Violation of a language rule, for example, leaving a function without executing a return-statement.

Tasking_Error Errors occurring during task communication (see Chapter 12).

Of course, Constraint_Error is by far the most common exception given the strong type checking of Ada. In addition, the programmer can declare exceptions which are treated exactly like predefined exceptions.

When an exception occurs, in Ada terminology when an exception is *raised*, a block of code called an *exception handler* is called. Unlike PL/I, executing an exception handler *terminates* the enclosing procedure. Since the handler does not return to the normal computation, there is no interference with optimization. Unlike global error handlers in C, Ada exception handling is extremely flexible since exception handlers can be associated with any subprogram:

```
procedure Main is
   procedure Proc is
      P: Node_Ptr;
   begin
      P := new Node;         -- Exception may be raised
      Statement_1;           -- Skip if exception is raised
   exception
      when Storage_Error =>
         . . .               -- Exception handler
   end Proc;
begin
   Proc;
   Statement_2;              -- Skip if exception is propagated
   exception
      when Storage_Error =>
         . . .               -- Exception handler
end Main;
```

After the last executable statement of a subprogram, the keyword exception introduces a sequence of exception handlers—one for each distinct exception. When the exception is raised, execution of the statements in the procedure is abandoned and the statements of the exception handler are executed instead. When the exception handler is completed, the procedure terminates *normally*. In the example: the allocator may raise the exception Storage_Error, in which case Statement_1 is skipped and the exception handler is executed. Upon completion of the exception handler, the procedure terminates normally and the main program continues with Statement_2.

The semantics of exception handling allow the programmer great flexibility in defining exception handling:

- If an exception is not handled within the procedure, the procedure is abandoned and the exception is raised again at the point of call. In the absence of an exception handler in Proc, the exception would be reraised in Main, Statement_2 would be skipped and the exception handler in Main executed.

- If an exception is raised *during* the execution of the handler, the handler is abandoned and the exception is raised again at the point of call.

- The programmer can choose to raise the same or another exception at the point of call rather than allowing the procedure to terminate normally. For example, we might want to translate a predefined exception such as Storage_Error to an application-defined exception. This is done by using an explicit raise-statement in the exception handler:

  ```
  exception
     when Storage_Error =>
        . . .                 -- Exception handler, then
        raise Overflow;       --    raise Overflow in caller
  ```

A handler for others can be used to handle all exceptions that are not mentioned in previous handlers.

If there is no handler for an exception even in the main program, the exception is handled by the run-time system which usually aborts the execution of the program with a message. It is good programming practice to ensure that all exceptions are handled, at least at the level of the main program.

The definition of exceptions in Ada 83 did not allow the exception handler access to any information about the exception. If more than one exception is handled identically, there is no way even to know which one occurred:

```
exception
   when Ex_1 | Ex_2 | Ex_3 =>
         -- Which exception has occurred ?
```

Ada 95 allows an exception handler to have a parameter:

```
exception
   when Ex: others =>
```

Whenever an exception occurs, the parameter Ex will contain the identity of the exception and predefined procedures enable the programmer to retrieve information about the exception. This information can also be programmer-defined (see the Language Reference Manual 11.4.1).

Implementation

The implementation of exception handlers is very efficient. A procedure that contains exception handlers will have an extra field in its activation record pointing to the handlers (Figure 11.1). It

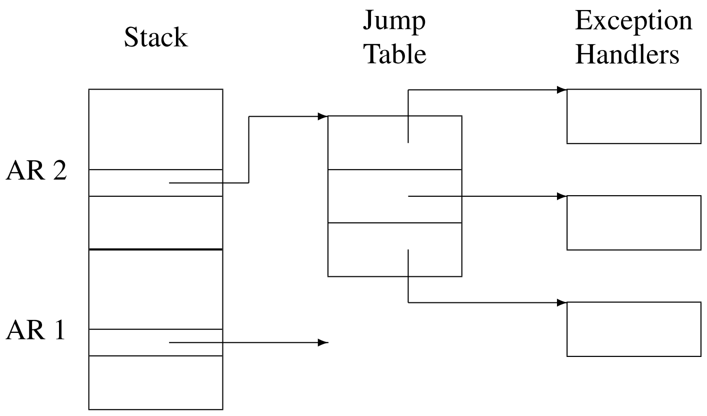

Figure 11.1: Implementation of exception handlers

takes just one instruction during procedure call to set up this field, and that is the only overhead there is if no exception occurs. If an exception is raised, finding a handler may require extensive computation to search the dynamic chain, but since exceptions rarely happen this is not a problem. Recall our warning not to use exception handlers as a replacement for a much more efficient if-statement.

11.4 Exceptions in C++

Exception handling in C++ is very similar to Ada in that an exception can be explicitly raised, handled by the appropriate handler (if any) and then the block (subprogram) is terminated. The differences are:

- Rather than attaching an exception handler to a subprogram, a special syntax is used to indicate a group of statements to which a handler applies.
- The exceptions are identified by a parameter type instead of by name. There is a special syntax equivalent to others for handling exceptions not explicitly mentioned.
- Exception families can be constructed using inheritance (Chapter 14).

- If an exception is not handled in Ada, a system-defined handler is invoked. In C++, the programmer can define a function terminate( ) which is called when an exception is not handled.

In the following example, the try block identifies the scope of a sequence of statements for which exception handlers (denoted by catch-blocks) are active. The throw-statement causes an exception to be raised; in this case it will be handled by the second catch-block since the string parameter of the throw-statement matches the char* parameter of the second catch-block:

```
void proc( )
{
    . . .                   // Exceptions not handled here
    try {
        . . .
        throw "Invalid data";// Raise exception
        . . .
    }
    catch (int i) {
        . . .               // Exception handler
    }
    catch (char *s) {
        . . .               // Exception handler
    }
    catch (...) {           // Other exceptions
        . . .               // Exception handler
    }
}
```

In both Ada and C++, it is possible that an exception handler is called upon to handle an exception which it cannot see, because the exception is declared in a package body (Ada), or the type is declared as private in a class (C++). If not handled by others (or ...), the exception will be reraised repeatedly until finally being processed as an unhandled exception. C++ has a way of preventing such unbounded behavior by specifying on a subprogram declaration exactly which exceptions it is willing to handle:

```
void proc( ) throw (t1, t2, t3);
```

Such *exception specifications* mean that an unlisted exception, which is raised but not handled in proc (or any subprogram called by proc), immediately calls the globally defined function unexpected( ) rather than continuing the search for a handler. In large systems, this construct is useful to document the full interface to subprograms, including the exceptions that it will propagate.

11.5 Error handling in Eiffel

Assertions

Eiffel's approach to exception handling is based on the concept that you shouldn't make errors in the first place. Of course all programmers strive for this, the difference being that Eiffel includes support within the language for specifying the correctness of a program. This is based on the concept of *assertions*, which are logical formulas commonly used for formalizing a program but not part of the program itself (Section 2.2).

Every subprogram (called a *routine* in Eiffel) can have assertions associated with it. For example, a subprogram for computing the result and the remainder of integer division would be written as follows:

```
integer_division(dividend, divisor, result, remainder: INTEGER) is
    require
        divisor > 0
    do
        from
            result = 0; remainder = dividend;
        invariant
            dividend = result*divisor + remainder
        variant
            remainder
        until
            remainder ¡ divisor
        loop
            remainder := remainder – divisor;
            result := result + 1;
        end
    ensure
        dividend = result*divisor + remainder;
        remainder ¡ divisor
end
```

The require-clause is called the *precondition* which specifies what the subprogram expects to be true about the input data. The do-clause contains the executable statements that comprise the subprogram. The ensure-clause, called the *postcondition*, specifies what the subprogram promises will be true if the do-clause is executed on data that satisfies the precondition. In this case it is trivial to see that the postcondition is satisfied because it follows immediately from the invariant (Section 6.6) and the until-clause.

On a larger scale, you can attach an invariant to a class (Section 15.5). For example, a class that implements a stack using an array would include an invariant of the form:

```
invariant
```

```
      top >= 0;
      top ¡ max;
```

All the subprograms of the class must ensure that the invariant is true when an object of the class is created and that every subprogram maintains the truth of the invariant. For example, the pop subprogram would have a precondition top¿0, otherwise executing the statement:

```
   top := top − 1
```

would falsify the invariant.

Enumeration types

Another use of invariants is to ensure type-safety that is achieved in other languages by the use of enumeration types. The Ada declarations:

Ada

```
   type Heat is (Off, Low, Medium, High);
   Dial: Heat;
```

would be written in Eiffel as an ordinary integer variable, with named constants:

```
   Dial:      Integer;
   Off:       Integer is 0;
   Low:       Integer is 1;
   Medium:    Integer is 2;
   High:      Integer is 3;
```

An invariant will ensure that meaningless assignments are not done:

```
   invariant
      Off <= Dial <= High
```

The latest version of Eiffel includes *unique constants* which are like enumeration names in that their actual values are assigned by the compiler. However, they are still integers so type safety must still be enforced using assertions: a postcondition should be attached to any subprogram that modifies variables whose values are to be limited to these constants.

Design by contract

Assertions are the basis of what Eiffel calls *design by contract*, meaning that the designer of the subprogram makes an implicit contract with the user of the subprogram: if you supply a state which satisfies the precondition, then I promise to transform the state so that it satisfies the postcondition. Similarly, a class maintains the truth of its invariants. If contracts are used throughout the system, then nothing can ever go wrong.

In practice, of course, the implementor of a subprogram may fail to fulfil the contract (either because the statements don't satisfy the assertions or because the wrong assertions were chosen). To support debugging and testing, an Eiffel implementation allows the user to request that assertions be checked upon subprogram entry and exit, so that the execution can be halted if an assertion is false.

Exceptions

Eiffel subprograms can have exception handlers:

```
proc is
    do
        . . .                    -- Exception may be raised
    rescue
        . . .                    -- Exception handler
end;
```

When an exception occurs, the subprogram is deemed to have failed and the statements following the rescue-clause are executed. Unlike Ada, when the exception handler terminates the exception is raised again in the calling program. This is equivalent to terminating an Ada exception handler by a raise-statement, which reraises in the calling subprogram the same exception that caused the handler to be entered.

The motivation behind this design is that a subprogram is supposed to satisfy a postcondition (and/or a class invariant). If it fails, you may want to cause a notification to be sent, but you certainly *cannot* satisfy the postcondition and you have failed in the task that the calling subprogram expects you to do. In other words, if you know how to fix the problem and satisfy the postcondition, then the fix should be part of the subprogram! This is similar to our advice not to use exceptions as replacements for an if-statement.

To assist in fixing a failed computation, the exception handler may make some modification, and then ask the subprogram to try again from the beginning by including the keyword retry as the last statement of the handler. The subsequent execution may then either succeed or fail. The principle is that a successful subprogram execution must terminate normally and fulfil its postcondition; otherwise, the execution fails.

An Ada exception handler can be simulated in Eiffel as follows, though this goes against the philosophy of the language:

```
proc is
    local
        tried: Boolean;          -- Initialized to false;
    do
        if not tried then
            -- Regular processing
            -- Raise exception
        else
            -- "Exception handler"
        end
    rescue
        if not tried then
            tried := true;      -- Don't retry twice
            retry
        end
end;
```

11.6 Exercises

1. The package Ada.Exceptions in Ada 95 defines types and subprograms for associating information with exceptions. Compare these constructs with the C++ constructs throw and catch.

2. Show that it is possible for an Ada exception to be raised outside the scope of the exception. (Hint: see Chapter 13.) How can you handle an exception whose declaration is not in scope?

3. Show how exception specifications in C++:

   ```
   void proc( ) throw (t1, t2, t2);
   ```

   can be simulated using multiple catch-blocks.

4. Study the class EXCEPTIONS in Eiffel and compare it with Ada exception handlers.

5. The exception construct in Java are quite similar to the exception construct in C++. One important difference is that a Java method must declare all exceptions that it can possibly throw. Justify this design decision and discuss its implications.