Exception Model

A car can be considered a correct piece of work. However, a car needs a driver to steer it in order to avoid disasters while performing its function. A software application could be a correct implementation of proven algorithms. However, during execution the program may face many practical challenges.

The set of linguistic mechanisms analogous to the steering wheel and pedals of a car are essential characteristics of a programming language. Paramount among such mechanisms is exception handling. A rudimentary mechanism for exception handling, as is the case with C++, deprives an engineer from controlling the execution of a program in a manner similar to asking a driver to operate a car without brakes.

In this note, we discuss the Z++ exception model and its advantages over the C++ model.

The disadvantages of C++ Model

The primary purpose of catching an exception is to handle it. There are two distinct notions of handling a none-fatal exception. In the context of Z++, the weak form of recovery from a handled exception is called **resumption**, and the stronger form is called **repetition**.

An exception model supporting resumption is useful in providing diagnostics, and then continuing the execution of the program at the statement immediately following the one that caused the exception.

In C++ model, one should avoid statements following a statement that may throw an exception. This is because the reason for having a throw is the possibility that an exception may occur. Should that happen, the statements following it in a C++ try block will be skipped, and there is no way to get back to them. The only reasonable thing to do is to put all those statements after the try-catch construct, effectively reducing the try block to statements that may throw exceptions.

Repetition allows repairing the cause of exception, perhaps by having user do something, and then trying the statement that caused the exception, over again. The C++ model is hopeless for this form of handling exceptions.

In the next section we illustrate the Z++ exception model. It is interesting to note that the C++ object-oriented model is quite costly. After all, exception objects must be created, copied, and destroyed.

Exception Layer

The notion of exception is associated with circles or layers, and handlers at each layer. C++ uses the terminology of try and catch for its object-oriented exception mechanism.

Z++ maintains the classic terminology while providing a more intuitive and expressive apparatus for exception mechanism.

The Z++ linguistic construct is “layer … handler … endlayer”, as shown below.

```
#include<exception.h>
using namespace exceptionSpace;

layer<exceptionType>

// statements that may raise exception are here

handler // (catching) handling exception

  case exceptionValue:
    // handle this exception

// more cases

endlayer;
```

The section between layer and handler corresponds to the C++ section between try and catch. The handler section extends until the closing tag endlayer.

Z++ allows extending an enumeration type somewhat like derivation of classes. The extended enumeration type will include the literals of its base. Z++ exceptions are of enumeration type. The header file “exception.h” defines the type exceptionType, which includes exceptions raised by the system, such as division by zero. Users define new exceptions by extending exceptionType.

The argument to layer is an exception type. The handler section permits only cases for exceptions of the type passed to the layer. The object-oriented version of C++ is neither more expressive, nor does it improve on this approach in any useful way.

Resumption

Z++ provides two notions of resumption (handling a none-fatal exception), namely repeat and resume. The statements repeat and resume can only appear in the handler section of a layer statement, and their target is the section between “layer … handler” of that same layer statement.

The Z++ repeat statement sends control back to the statement that caused the exception. **One uses the repeat statement when the cause of an exception can be repaired, and the statement that originally caused the exception can be re-executed**. Often times this is the scenario that one faces.

There are times that the cause of an exception either cannot be repaired, or the repair is not necessary. In such cases, possibly after logging some information, one wishes to continue executing the statements following the one that caused the exception. The Z++

resume statement sends control back to the statement following the one that caused the exception, precisely what one would like to happen.

Conclusion

The C++ object-oriented formulation of the notion of exception mechanism does not introduce any practically useful novelty. On the other hand, Z++ exception mechanism is complete for all practical purposes. Both languages allow extending exceptions. However, the C++ object-oriented model is expensive because exception objects must be created and destroyed, and perhaps copied.

In addition, Z++ compiler chases exceptions in order to ensure that uncaught exceptions will not result in unforeseen disasters.