Exception Handling

Jaiveer Singh & Raju Singh
Department of Information and Technology Dronacharya College of Engineering Gurgaon, India
Raju.16930@ggnindia.dronacharya.info; *Jaiveer.16915@ggnindia.dronacharya.info*

Abstract-

-In this paper, we address a new feature added to ANSI (American National Standard Institute) C++ named Exception Handling. We often come across some peculiar problems other than logic or syntax error, these anomalies or unusual condition that a program may encounter while executing is termed as exception. This paper will guide you to a procedural way or mechanism in order to deal with the exceptions. Beyond that it elaborates the two kinds of exceptions, namely, synchronousexceptions and asynchronous exceptions. The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken.

Keywords –

Syntax error; exception; anomalies; catch; try; throw; rethrow

1. Introduction

Exceptions are an important error handling aspect of many programming languages, especially object-oriented languages such as C++ and Java. This paper is written to stimulate discussion of exception handling in C++. The primary purpose of the exception handling mechanism described here is to cope with this problem for C++ programs; other uses of what has been called exception handling in the literature are considered secondary. Exceptions are often used to indicate unusual error conditions during the execution of an application (resource exhaustion, for instance) and provide a way to transfer control to special-purpose exception handling code [9]. The mechanism described is designed to handle only synchronous exceptions, such as array range checks. Asynchronous exceptions, such as keyboard interrupts, are not handled directly by this mechanism. A guiding principle is that exceptions are rare compared [10] to function calls and that exception handlers are rare compared to function definitions. The exception handling code deals with the unusual circumstance and either terminates the program or re- turns control to the non-exceptional part of the program, if possible. Therefore, exceptions introduce additional, and often complex, interprocedural control flow into the program, in addition to the standard non-exceptional control flow. When a program encounters an exception condition, it is important that it is identified and dealt with effectively. ANSI C++ provides built-in language features to detect and handle exceptions which are basically run time errors.

Exceptions handling was not a part of the original C++. It is a new feature added to ANSI C++. Today, almost all compliers support this feature. C++ exception handling provides a type-safe, integrated approach, coping with the unusual predictable problems that arise while executing a program.

2.1 Basics of Exception Handling

Exceptions are of two kinds, namely, synchronous and asynchronous exceptions. Errors such as "out-of-range index" and "over-flow" belong [3] to the synchronous type exceptions. The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:

1. Find the problem (Hit the exceptions).
2. Inform that an error has occurred (Throw the exception).
3. Receive the error information (Catch the exception).
4. Take corrective actions (Handle the exception).

The error handling code basically consists of two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.

2.2 Exception Handling Mechanism

C++ exceptions handling mechanism is basically built upon three keywords, namely, try, throw, catch. The keyword try is used to preface a bock of statements (surrounded by braces) which may generate exceptions. This bock of statements is known as try bock. When an exception is detected, it is thrown using a throw statement [4] in the try bock. A catch block defined by the keyword catch 'catches' the exceptions 'thrown' by the throw statement in the try bock, and handles it appropriatey. The relationship is shown in Fig. 2.2.(a)

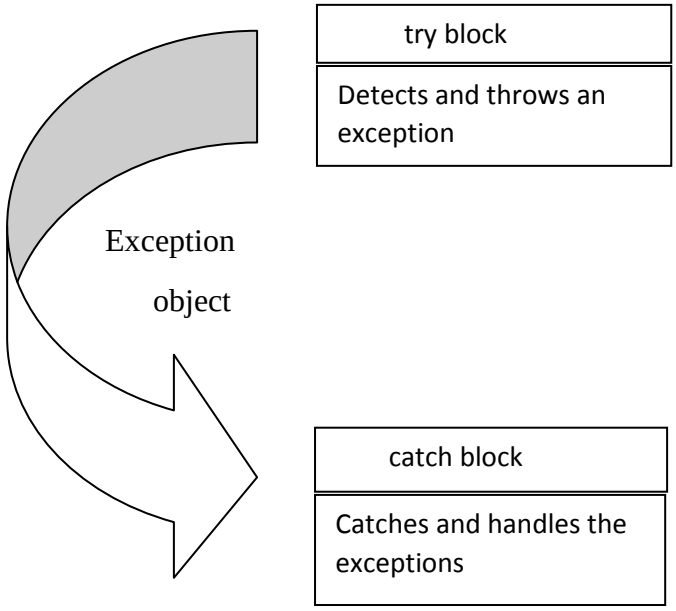

Fig.2.2.(a) The block throwing exception

2.2 An example

Suppose that an exception called xxii can occur in a function g() called by a function f(). How can the programmer off() gain control in that case? The wish to 'catch' the exception xxii when it occurs and g() doesn't handle it can be expressed like this:

```
Int f ()

{   Try {return g();}

Catch (xxii) {// we get here only if 'xxii' occurs error ("g () goofed: xxii return 22; ") ;}
```

The text from catch to the next close brace is called a handler for the kind of exception named xxii. A single try-block can have handlers for several distinct exceptions; a handler marked by the ellipsis, picks up every exception not previously mentioned. For example:

```
int f() { try {return g();}catch (xx)

{// we get here only if 'xx' occurs error("g() goofed: xx"); return 20;}

catch (xxii) { // we get here only if 'xxii' occurs  error("g() goofed: xxii");return 22; }

catch (...) {// we get here only if an exception // that isn't 'xxii' or 'xx' occurs error("g() goofed");  return 0; } }
```

The series of handlers is rather like a switch statement. The handler marked (...) is rather like a default. Note, however, that there is no 'fall through' from one handler to another as there is from one case to another. An alternative and more accurate analogy is that the set of handlers looks very much like a set of overloaded functions. However, unlike a set of overloaded functions, the try clauses [5] are checked in the sequence in which they appear. An exception handler is associated with a try-block and is invoked whenever its exception occurs in that block or in any function called directly or indirectly from it. For example, say in the example above that xxii didn't actually occur in g() but in a function h() called by g():

```
In t g(){ return h();}in t h(){ throw xxii(); // make exception 'xxii' occur }
```

The handler in f() would still handle the exception xxii. We will use the phrase 'throwing an exception' to denote the operation of causing an exception to occur. The reason we don't use the more common phrase 'raising an exception' is that raise() is a C standard library function and therefore not available [6] for our purpose. The word signal is similarly unavailable. Similarly, we chose catch in preference to handle because handle is a commonly used C identifier. A handler looks a lot like a function definition. A throw-expression looks somewhat like both a func- tion call and a return statement. We will see below that neither similarity is superficial. Because g() might be written in C or some other language that does not know about C++ exceptions, a fully general implementation of the C++ exception mechanism cannot rely on decorations of the stack frame, passing of hidden arguments to functions not in C++, or other techniques that require compiler coop- eration for every function called. Once a handler has caught an exception, that exception has been dealt with and other handlers thatmight exist for it become irrelevant. In other words, only the active handler most recently encountered by the thread of control will be invoked. For example, here xxii will still be caught by the handler in f ():

```
In t e()

{ Try {return f(); // f() handles xxii }
```

catch (xxii) {// so we will not get here //... } }

Another way to look at it is that if a statement or function handles a particular exception, then the fact that the exception [1] has been thrown and caught is invisible in the surrounding context – unless, of course, the exception handler itself notifies something in the surrounding context that the exception occurred.

3. Throwing mechanism

When a exception that is desired to be handled is detected, it is thrown using the throw statement in one of the following forms:

Throw (exception) ;

throw exception ;

throw;
// used for rethrowing an exception

The operand object exception may be of any type, including constants. It is also possible to throw objects not intended for error handling.

When an exception is thrown, it will be caught by the catch statement associated with try block. That is, the control exits [7] the current try block, and is transferred to the catch block after that try block.

Throw point can be in a deeply nested scope within a try block or in a deeply nested function call. In case, control is transferred to the catch statement.

4. Catching Mechanism

Code for handling exceptions is included in the catch blocks. A catch block looks like a function definition and is of the form

catch(type arg) { // statements for // managing exceptions }

The type indicates the type of exceptions that catch block handles. The parameter arg is an optional parameter name. Note that the exception-handling code is palced between two braces. The catch statement catches an exception whose type matches with the type of catch arguments. When it is caught, the code in the catch block is executed.

If the parameter in the catch statement is named, then the parameter can be used in the exception-handling code [8]. After executing the handler, the control goes to the statement immediately following the catch block.

Due to mismatch, if an exception is not caught, abnormal program termination will occur. Its important to note that the catch block is simply skipped if the catch statement does not catch an exception.

5. Rethrowing an Exception

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression (throw without assignment_expression) causes the originally thrown object to be rethrown.

Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next dynamically enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks for the dynamically enclosing try block have an opportunity to catch the exception.

The following example demonstrates rethrowing an exception:

#include <iostream>

using namespace std;

```
struct E {const char* message; E() : message("Class E") { }};

struct E1 : E {const char* message; E1() : message("Class E1") { }};

struct E2 : E {const char* message;E2() : message("Class E2") { }};

void f() {try {cout<< "In try block of f()" <<endl;

cout<< "Throwing exception of type E1" <<endl;

  E1 my Exception; throw my Exception; }

catch (E2& e) {cout<< "In handler of f(), catch (E2& e)" <<endl;

cout<< "Exception: " <<e.message<<endl; throw; }

catch (E1& e) {cout<< "In handler of f(), catch (E1& e)" <<endl;

cout<< "Exception: " <<e.message<<endl;

throw; }

catch (E& e) { cout<< "In handler of f(), catch (E& e)" <<endl;

cout<< "Exception: " <<e.message<<endl;

throw; }}

int main() {try { cout<< "In try block of main()" <<endl; f(); }

catch (E2& e) {cout<< "In handler of main(), catch (E2& e)" <<endl;

cout<< "Exception: " <<e.message<<endl; }

catch (...) {cout<< "In handler of main(), catch (...)" <<endl; }}
```

The following is the output of the above example:

```
In try block of main ()

In try block of f ()

Throwing exception of type E1

In handler of f (), catch (E1& e)

Exception: Class E1

In handler of main (), catch (...)
```

The try block in the main () function calls function f(). The try block in function f() throws an object of type E1 named myException [2]. The handler catch (E1 &e) catches myException. The handler then rethrows myException with the statement throw to the next dynamically enclosing try block: the try block in the main() function. The handler catch(...) catches myException.

6. Conclusions

The exception handling scheme described here is flexible enough to cope with most synchronous exceptional circumstances. Its semantics are independent of machine details and can be implemented in several ways optimized for different aspects. In particular, portable and run-time efficient implementations are both possible. The exception handling scheme presented here should make error handling easier and less error-prone

7. References

[1] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson: Modula-3 Report. DEC Systems Research Center. August 1988.

[2] Flaviu Cristian: Exception Handling. in Dependability of Resilient Computers, T. Andersen Editor, BSP Professional Books, Blackwell Scientific Publications, 1989.

[3] Margaret A. Ellis and BjarneStroustrup: The Annotated C++ Reference Manual. Addison Wes- ley 1990.

[4] J. Goodenough: Exception Handling: Issues and a Proposed Notation. CACM December 1975. [5] Steve C. Glassman and Michael J. Jordan: Safe Use of Exceptions. Personal communication. [6] Steve C. Glassman and Michael J. Jordan: Preventing Uncaught Exceptions. Olivetti Software Technology Laboratory. August 1989.

[7] Griswold, Poage, Polonsky: The SNOBOL4 Programming Language. Prentice-Hall 1971

[8] Andrew Koenig and BjarneStroustrup: Exception Handling for C++. Proc. C++ at Work Con- ference, SIGS Publications, November 1989.

[9] Andrew Koenig BjarneStroustrupAT&T Bell Laboratories Murray Hill, New Jersey 07974 ark@europa.att.com bs@research.att.com

[10] Prakash Prabhu1,2, Naoto Maeda1,3, Gogul Balakrishnan1, Franjo Ivanˇci´c1, and Aarti Gupta11 NEC Laboratories America, 4 Independence Way, Suite 200, Princeton, NJ 08540 2 Princeton University, Department of Computer Science, Princeton, NJ 08540 3 NEC Corporation, Kanagawa 211-8666, Japan