MODULE 4
Exception Handling

Exception Handling

- Introduction
- Basic Concepts
- Design Issues

Introduction

In a language without exception handling:

- computer hardware systems are not capable of detecting certain run-time error conditions, such as floating-point overflow.
- In these languages, the occurrence of such an error simply causes the program to be terminated and control to be transferred to the operating system.
- When an exception occurs, control goes to the operating system, where a diagnostic message is displayed and the program is terminated

In a language with exception handling:

- Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing
- Many languages allow programs to trap input/ output errors (including EOF)
- For example, a Fortran **Read** statement can intercept input errors and end-of-file conditions, both of which are detected by the input device hardware.
- The end-of-file and events that are always errors, such as a failed input process, Fortran handles both situations with the same mechanism.
- Consider the following Fortran Read statement:

  Read(Unit=5, Fmt=1000, Err=100, End=999) Weight
- The Err clause specifies that control is to be transferred to the statement labeled 100 if an error occurs in the read operation.
- The End clause specifies that control is to be transferred to the statement labeled 999 if the read operation encounters the end of the file.

Basic Concepts

- An **exception** is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing
  - We consider both the errors detected by hardware, such as disk read errors, and unusual conditions, such as end-of-file (which is also detected by hardware), to be exceptions.
  - We further extend the concept of an exception to include errors or unusual conditions that are software-detectable (by either a software interpreter or the user code itself).
- The special processing that may be required after the detection of an exception is called **exception handling**
- The exception handling code unit is called an **exception handler**

- An exception is *raised* when its associated event occurs. In some C-based languages, exceptions are said to be *thrown*, rather than *raised*.
- If it is desirable to handle an exception in the unit in which it is detected, the handler is included as a segment of code in that unit.

- Alternatives:

1. Send an auxiliary parameter or use the return value to indicate the return status of a subprogram
   - e.g., C standard library functions
2. Pass a label parameter to all subprograms (error return is to the passed label)
   - e.g., FORTRAN
3. Pass an exception handling subprogram to all subprograms

- **Advantages of Built-in Exception Handling:**

1. Error detection code is tedious to write and it clutters the program
2. Exception propagation allows a high level of reuse of exception handling code
   - Exception propagation allows an exception raised in one program unit to be handled in some other unit in its dynamic or static ancestry.
   - This allows a single exception handler to be used for any number of different program units.
   - This reuse can result in significant savings in development cost, program size, and program complexity.

Design Issues

- A system might allow both predefined and user-defined exceptions and exception handlers.
- predefined exceptions are implicitly raised, whereas user-defined exceptions must be explicitly raised by user code.
- Consider the following skeletal subprogram that includes an exception-handling mechanism for an implicitly raised exception:

```
void example() {
  ...
  average = sum / total;
  ...
  return;
/* Exception handlers */
  when zero_divide {
    average = 0;
    printf("Error-divisor (total) is zero\n");
  }
  ...
}
```

- The exception of division by zero, which is implicitly raised, causes control to transfer to the appropriate handler, which is then executed.

The exception-handling design issues can be summarized as follows:

- How and where are exception handlers specified, and what is their scope?
- How is an exception occurrence bound to an exception handler?
- Can information about an exception be passed to the handler?
- Where does execution continue, if at all, after an exception handler completes its execution? (This is the question of continuation or resumption.)
- Is some form of finalization provided?
- How are user-defined exceptions specified?
- If there are predefined exceptions, should there be default exception handlers for programs that do not provide their own?
- Can predefined exceptions be explicitly raised?
- Are hardware-detectable errors treated as exceptions that may be handled?
- Are there any predefined exceptions?
- Should it be possible to disable predefined exceptions?

- The first design issue for exception handling is **how an exception occurrence is bound to an exception handler**.
- This issue occurs on two different levels.
- On the unit level, there is the question of how the same exception being raised at different points in a unit can be bound to different handlers within the unit.
  - For example, in the example subprogram, there is a handler for a division-by-zero exception that appears to be written to deal with an occurrence of division by zero in a particular statement (the one shown).
  - But suppose the function includes several other expressions with division operators. For those operators, this handler would probably not be appropriate.
  - So, it should be possible to bind the exceptions that can be raised by particular statements to particular handlers, even though the same exception can be raised by many different statements.
- At a higher level, the binding question arises when there is no exception handler local to the unit in which the exception is raised.
- In this case, the language designer must decide whether to propagate the exception to some other unit and, if so, where.

- For example, if handlers must be local, then many handlers must be written, which complicates both the writing and reading of the program.
- On the other hand, if exceptions are propagated, a single handler might handle the same exception raised in several program units, which may require the handler to be more general than one would prefer.
- An issue that is related to the binding of an exception to an exception handler is **whether information about the exception is made available to the handler**.
- After an exception handler executes, either control can transfer to somewhere in the program outside of the handler code or program execution can simply terminate.
- We term this the question of control continuation after handler execution, or simply **continuation.** Termination is obviously the simplest choice, and in many error exception conditions, the best.
- However, in other situations, particularly those associated with unusual but not erroneous events, the choice of continuing execution is best. This design is called **resumption.**

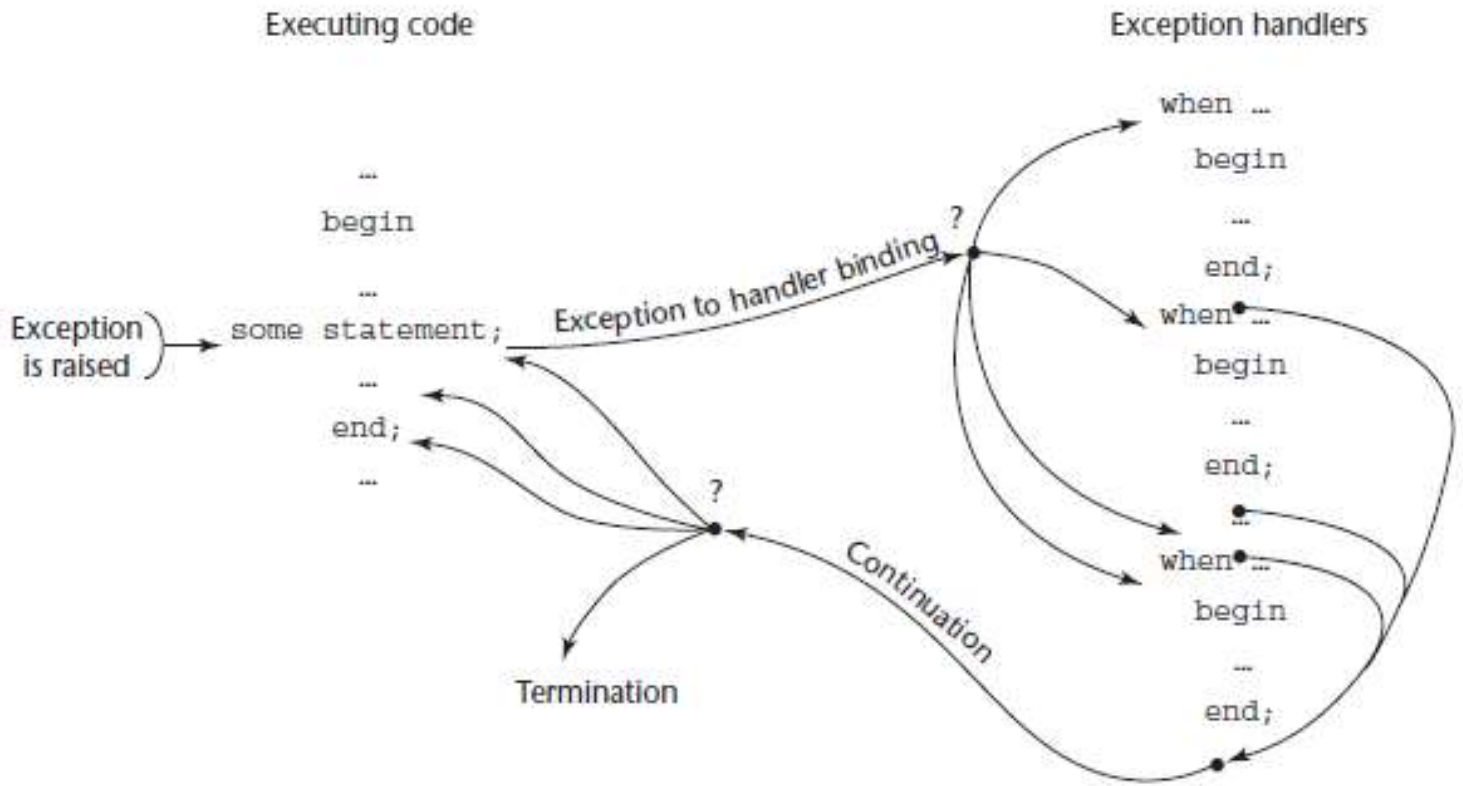

Figure 14.1

Exception-handling control flow

- When exception handling is included, a subprogram's execution can terminate in two ways: when its execution is complete or when it encounters an exception.
- In some situations, it is necessary to complete some computation regardless of how subprogram execution terminates. The ability to specify such a computation is called **finalization**.
- **The choice of whether to support finalization** is obviously a design issue for exception handling.

- Another design issue is the following: **If users are allowed to define exceptions, how are these exceptions specified?**
- The usual answer is to require that they be declared in the specification parts of the program units in which they can be raised.
- The scope of a declared exception is usually the scope of the program unit that contains the declaration.

- **In the case where a language provides predefined exceptions**, several other design issues follow:
- For example, should the language run-time system provide default handlers for the built-in exceptions, or should the user be required to write handlers for all exceptions?
- Another question is whether predefined exceptions can be raised explicitly by the user program. This usage can be convenient if there are software-detectable situations in which the user would like to use a predefined handler.
- Another issue is whether hardware-detectable errors can be handled by user programs.
- If not, all exceptions obviously are software detectable. A related question is whether there should be any predefined exceptions. Predefined exceptions are implicitly raised by either hardware or system software.
- Finally, there is the question of whether exceptions, either predefined or user defined, can be temporarily or permanently disabled.