

605.404 Object-Oriented Programming with C++
Summer 2007
Lectures 07

7	06/27	Namespaces and Exceptions; Source files and Programs	S: Chaps 8-9
---	-------	--	--------------

Additional Resources Related to This Lecture:

Exception Handling in ANSI C, Colvin, C Users Journal, August 1991, pp 77-88.

Better Exception-Handling in Block-Structured Systems, Knudsen, IEEE Software, May 1987. pp 40-49.

LECTURE:

Chapters 8-9

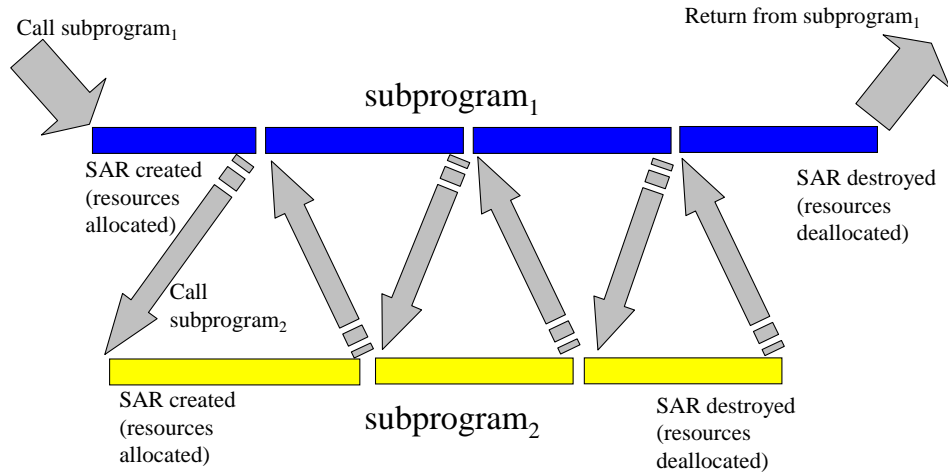
Notation: Dashed arrow between modules: "implements"

Solid arrow between modules: "relies on the interface provided by"

Chapter 8 Namespaces and Exceptions

- I. 8.1 Modularization and Interfaces
 - A. keep a distinction between a module and its interface
 1. interface should contain only the "advertised" facts about a module
 2. check out posted Ada reference materials (Package Specifications and Declarations, Package Bodies)
 - B. minimize dependencies between modules caused by error handling
 1. See additional materials posted at the end of the Stroustrup chapter summaries in this lecture
 2. remember to check formal notation information in lecture 01 this semester
 - C. concurrent subprograms have separate runtime memory managed by the operating system
 - D. objective is safe, convenient, and efficient communication across module boundaries

Coroutines



- E.
- II. 8.2 Namespaces
- A. a mechanism for expressing a logical grouping
 - B. see section 3.3.5 Namespace Scope of the ISO/IEC 14882 C++ standard
 - C. namespaces can be anonymous (see section 8.2.5.1 in our text)

```
#include<iostream>
namespace {
    int k=7;
}
int main() {
    std::cout << k;
    std::cout << ::k;
}
```

- D. unnamed namespaces in different translation units are different
- E. cannot declare a new member of a namespace outside a namespace definition using the qualifier syntax

F. example from 3.3.5 Namespace Scope of the ISO/IEC 14882 C++ standard

```
namespace N {
    int i;
    int g(int a) { return a; }
    int j;
    void q();
}
namespace {
    int k=7;
}
namespace N {
    int g(char a) //overloads N::g(int)
    {
        return k+a; //k is from an unnamed namespace
    }
    int i; //error: duplicate definition
    int j(); //OK: duplicate function declaration
    int j() //OK: definition of N::j()
    {
        return g(i);
    }
    int q(); //error different return type
}
```

- G. names declared outside all named or unnamed namespaces, blocks, function declarations, function definitions, and classes has global namespace scope
1. main() must be global for the run-time environment to recognize it as special

H. 8.2.1 Qualified Names

I.8.2.2 Using Declarations

1. introduces a local synonym (e.g., using Lexer::get_token)

J. 8.2.3 Using Directives

1. makes all names in a namespace available (e.g., using namespace Lexer)

K. 8.2.4 Multiple Interfaces

1. namespace provides
 - a) common environment for functions implementing the module
 - b) external interface offered by the module
2. 8.2.4.1 Interface Design Alternatives
 - a) express programs so that the set of potential dependencies is reduced to the set of actual dependencies
 - b) overkill solution to dependency minimization

L. 8.2.5 Avoiding Name Clashes

1. 8.2.5.1 Unnamed Namespaces

M. 8.2.6 Name Lookup

1. if a function is not found in the context of its use, we look in the namespaces of its arguments
2. when a class member invokes a named function, other members of the same class and its base class are preferred over functions potentially found based on argument types; operators differ though (see sections 11.2.1 and 11.2.4)

N. 8.2.7 Namespace Aliases

- O. 8.2.8 Namespace Composition
 - 1. A namespace should
 - a) Express a logically coherent set of features
 - b) Not give access to unrelated features
 - c) Not impose significant notational burden on users
 - 2. must use scope resolution operator when something is defined
 - 3. 8.2.8.1 Selection (via using declarations)
 - 4. 8.2.8.2 Composition and Selection
 - a) composition (using directives)
 - b) selection (using declarations)
 - c) names explicitly declared in a namespace including names declared by using-declarations take precedence over names made accessible in another scope by a using-directive
 - 5. 8.2.9 Namespaces and Old Code
 - a) 8.2.9.1 Namespaces and C
 - (1)cautions about namespace definition in libraries (especially std)
 - b) 8.2.9.2 Namespaces and Overloading
 - (1)overloading works across namespaces
 - c) 8.2.9.3 Namespaces are Open
 - (1)that is they are not closed (not completed and done); they are changeable

III. 8.3 Exceptions

- A. when a program is composed of separate modules and especially when those modules come from separately developed libraries, error handling needs to be separated into two distinct parts:
 - 1. the reporting of error conditions that cannot be resolved locally
 - 2. the handling of errors detected elsewhere
 - 3. library author can detect run-time errors but does not know what to do about them in the problem domain of the current program
 - 4. user of library may know how to cope with errors with respect to the problem domain of the current program, but cannot detect the errors in the library
 - 5. Chapter 14 has much more detail on exceptions
- B. 8.3.1 Throw and Catch
 - 1. the notion of an exception is provided to help deal with error reporting – says Stroustrup
 - 2. I believe exceptions are part of the postcondition actions of a module in addition to Stroustrup's definition
 - 3. `try { } catch () { }`
 - a) `catch ()` is the exception handler
 - b) must occur immediately after a try block or another exception handler
 - c) `()` specifies the type of the objects that can be caught by this exception handler and optionally names the object caught
 - 4. if an exception is thrown and no try-block catches it, the program terminates
- C. 8.3.2 Discrimination of Exceptions
 - 1. suggestion to define types with no other purpose than exception handling
 - 2. similarity of list of exception handlers to switch statement (no need for break though)
 - 3. an exception is considered handled immediately upon entry into its handler so that exceptions thrown while executing a handler must be dealt with by the

callers of the try-block

- D. 8.3.3 Exceptions in the Calculator (examples)
 - 1. 8.3.3.1 Alternative Error-Handling Strategies
 - a) use of static variables (not necessarily a good idea – see page 193)
 - b) exception handling intended for dealing with nonlocal problems (handle locally if possible)

Chapter 9 Source Files and Programs

IV. 9.1 Separate Compilation

- A. file is traditional unit of storage and of compilation
- B. compilation process
 - 1. source code -> preprocessing -> translation unit
 - 2. compilation proper
 - 3. assembly
 - 4. linking
- C. physical structure, dependency graph

V. 9.2 Linkage

- A. compiling with gpp (example bottom page 198)
 - 1. gpp -c file1.cpp
 - 2. gpp -o file2.exe file1.o file1.cpp
- B. a name that can be used in translation units different from the one in which it was defined is said to have external linkage
- C. an inline function must be defined – by identical definitions – in every translation unit in which it is used
- D. consts and typedefs have internal linkage
- E. global variables should be placed in header files only
- F. unnamed namespaces have similar effect to that of internal linkage
- G. 9.2.1 Header Files
 - 1. rule of thumb on header contents
- H. 9.2.2 Standard Library Headers
 - 1. importance of using <X> for old C standard headers <X>
 - 2. defines the std namespace
- I. 9.2.3 the One-Definition rule
 - 1. two definitions of a class, template, or inline function are accepted as examples of the same unique definition if and only if
 - a) they appear in different translation units
 - b) they are token-for-token identical
 - c) the meanings of those tokens are the same in both translation units
- J. 9.2.4 Linkage to Non-C++ Code
 - 1. specifying linkage conventions in extern declarations, as in extern “C”
 - 2. __cplusplus predefined macro
 - 3. “C”, “C++”
 - 4. note that bsearch and qsort generate extern references for both forms when used in code from stdlib.h

VI. 9.3 Using Header Files

- A. 9.3.1 Single Header File
- B. 9.3.2 Multiple Header Files
 - 1. 9.3.2.1 Other Calculator Modules
 - 2. 9.3.2.2 Use of Headers
- C. 9.3.3 Include Guards

1. #ifndef ... #define...#endif
- VII. 9.4 Programs
- A. a program is a collection of separately compiled units combined by a linker
 - B. 9.4.1 Initialization of nonlocal variables
 1. if a variable outside any function (global, namespace, and static) variables are initialized in their definition order
 2. default initializer for built-in types and enumerations is zero
 3. 9.4.1.1 Program Termination
 - a) return from main()
 - b) call exit() – calls destructors for constructed static objects; local variables of the calling function and its callers will not have their destructors invoked
 - c) call abort() – does not call constructors for static objects
 - d) throw an uncaught exception
 - e) atexit() allows code to be executed a program termination
- VIII. Meyer OOSC Chapter 7 Systematic approaches to program construction
- A. introductory comments
 1. key notion is programming by contract
 - B. the notation of assertion
 1. you may associate with an element of executable code an expression of the element's purpose (derived from the specification)
 2. Label: boolean expression ;...; Label: boolean expression
 - C. preconditions and postconditions
 1. precondition expresses the properties that must hold whenever the routine is called (require)
 2. postcondition describes the properties that the routine guarantees when it returns (ensure)
 - a) old attribute denotes the value the attribute had on entry
 - b) Nochange
 - D. contracting for software reliability
 1. preconditions and postconditions in a routine should be viewed as a contract that binds the routine and its callers
 2. "If you promise to call r with pre satisfied then I, in return, promise to deliver a final state in which post is satisfied
 3. precondition binds clients: defines conditions under which a call to the routine is legitimate
 4. postcondition binds the class: it defines the conditions that must be ensured by the routine on return
 5. page 117 discussion (where would checks be done?)
 - a) don't do redundant checking
 - b) assume that the precondition is satisfied when writing a routine body
 - c) by forcing a clear definition of whose responsibility it is to check every condition required for correct operation, the method emphasizes a systematic, rigorous approach to the construction of correct programs (page 119)
 6. how restrictive?
 - a) if you have direct control over the calling programs, you can afford to define strict preconditions in order to keep the code simple
 - b) if not loosen the preconditions and deal explicitly with errors

- 7. filter modules
 - a) new layers of software that serve as filters between possibly careless calls and unprotected data
 - b) calling module must check the filter's comments about the results
 - c) filters achieve the needed separation of concerns between algorithmic techniques to deal with normal cases and techniques for handling errors
- E. side-effects in functions
 - 1. commands versus queries
 - a) queries do not change the internal state of the machine (functions) but return a result
 - b) commands change the internal state of the machine (procedures) but do not return a result
 - 2. prohibition of side-effects in functions
 - 3. legitimate side-effects: an example
- F. other constructs involving assertions
 - 1. loop invariant and variants
 - a) a correct invariant for a loop is an assertion which is satisfied after loop initialization and preserved by every iteration of the loop body
 - b) a variant is an integer expression whose value is non-negative after loop initialization and is decreased by at least one by every execution of the loop body (when the exit condition is not satisfied) but never becomes negative
 - 2. the check instruction
- G. using assertions
 - 1. four main applications of assertions
 - a) help in writing correct software
 - b) documentation aid
 - c) debugging tool
 - d) support for software fault detection
- H. coping with failure: disciplined exceptions
 - 1. when assertions are checked at run-time a debugging mechanism and a mechanism for fault tolerance and fault recovery are introduced
 - 2. monitoring assertions at run-time
 - a) C behavior contrasted to Eiffel
 - 3. exceptions
 - a) role for exception mechanism
 - (1) account for errors that remain in the software and include mechanisms that will handle any resulting run-time failure either by exiting cleanly or attempting recovery
 - (2) when the failure is an abnormal condition detected by the hardware or operating system
 - b) definitions (page 147)
 - (1) an exception is the occurrence of an abnormal condition during execution of a software element
 - (2) a failure is the inability of a software element to satisfy its purpose
 - (3) an error is the presence in the software of some element not satisfying its specification

- c) what to do when an exception is detected
 - (1) organized panic
 - (2) retry
- d) the rescue clause in Eiffel
- e) the retry clause

```

routine is
  require ...
  local ...
  do
    body
  ensure ...
  rescue
    rescue_clause
  retry
end

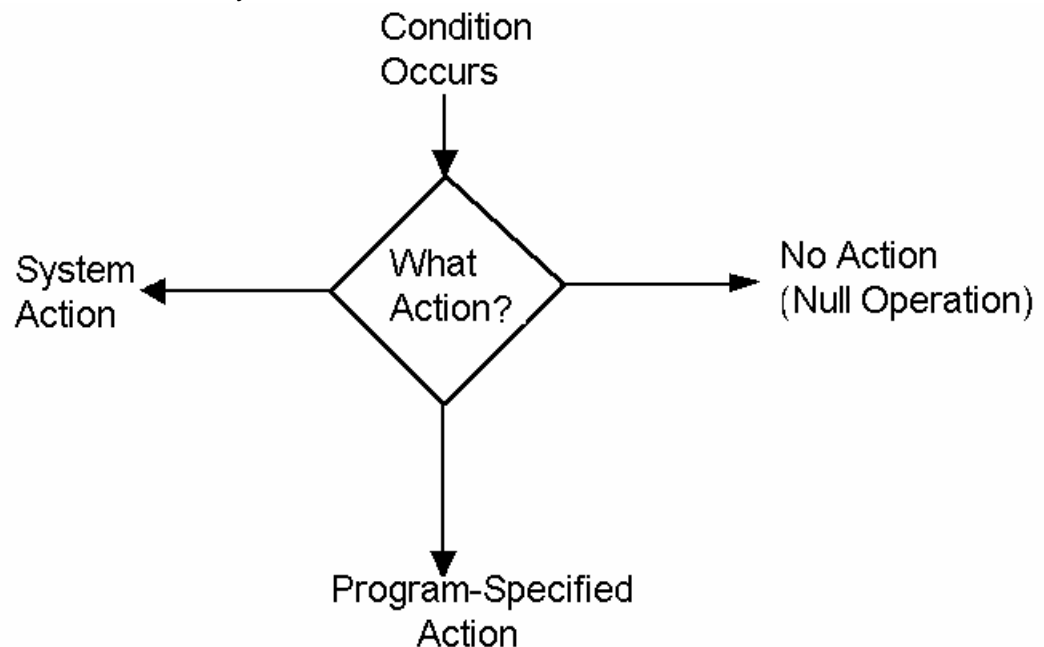
```

- f) Page 149 Eiffel definition of Exception
- g) page 150 Eiffel definition of Failure
- h) page 151 - the rescue clause may be viewed as a routine body whose postcondition would be the invariant of the class
- i) in practice the rescue clause should be a short sequence of simple instructions designed to bring the object to a stable state and to either retry the operation or terminate with failure
- j) processing hardware and operating system exceptions
- 4. summary discussion in the chapter
 - a) limitations on assertions in programming languages
 - (1) really need to directly manipulate sets, sequences, functions, relations, and first-order predicates with quantifiers such as "for all"
 - (2) remember to add comments explaining the assertion
 - (3) assertions may contain function calls
 - b) disciplined exceptions
 - (1) Ada raise cancels the routine that executed it and returns control to its caller
 - (2) the handler can ignore an exception or return an alternate result
 - (3) in programming by contract, though, a function may only succeed or fail
- IX. Meyer OOSC Chapter 18 object-oriented programming in Ada
 - A. exceptions
 - 1. separate the handling of errors from their detection
 - 2. raise command
 - 3. when an instruction raise exception is executed, control does not flow to the instruction that would normally follow, but is transferred to an exception handler
 - a) if no handler in the dynamic chain handles exc the program terminates and control is returned to the operating system
 - b) if a handler is found, the routine is executed and returns control to its caller or terminates if it is the main program

4. limit to three cases
 - a) abnormal cases leading to preemptive action by the hardware or operating system
 - b) abnormal cases that must lead to termination as early as possible to avoid catastrophic consequences
 - c) software fault tolerance
 5. First Law of Software Contracting: here are only two ways a routine call may terminate: either the routine fulfills its contract, or it fails to fulfil it
 6. Second Law of Software Contracting: If a routine fails to fulfil its contract, the current execution of its caller also fails to fulfil its own contract
 7. Ada Exception Rule - The execution of every exception handler should end by either executing a raise instruction or retrying the enclosing program unit
- B. tasks (information)
- X. Chapter 7, Fairley, Software Engineering Concepts, Modern Programming Languages
- A. Exception Handling
1. in PL/1 - resumption model
 - a) resume operation after action of an exception handler
 - b) only terminates if a return statement was at the end of the exception handler
 - c) makes code optimization and formal verification very difficult
 2. in Ada - termination model
 - a) program unit is automatically terminated following execution of the exception handler; the exception handler is viewed as replacing the remaining body of the program unit
 - b) if no exception handler is provided for a given exception condition, the program unit is terminated and the same exception is raised in the calling unit
 - c) calling program can correct the condition and re-invoke the calling routine
- XI. Chapter 12 Exception Handling (Sebesta)
- A. 12.1 Introduction to Exception Handling
1. hardware and operating system errors, except I/O may not be accessible to an application program
 2. I/O intercepts some errors
 - a) input errors
 - b) EOF
 - c) FORTRAN unconditional branch example
- B. 12.1.1 Basic Concepts
1. exception - any unusual event, erroneous or not, that is detectable either by hardware or software that may require special processing
 2. exception. An event that causes suspension of normal program execution. Types include addressing exception, data exception, operation exception, overflow exception, protection exception, underflow exception. [IEEE Std 610.12-1990]
 3. exception handling - special processing that may be required by the detection of an exception
 - a) exception handler code
 - b) an exception is raised when its associated event occurs
 4. can disable exception detection

5. handler types
 - a) handle by calling program; receives a status variable from the called program
 - b) pass a label parameter to the subprogram so the called unit can return to a different point in the caller unit if an exception occurs
 - c) handler as a separate subprogram; handler program parameter may have to be passed to each called unit
- C. 12.1.2 Design Issues
 1. handlers as complete program units versus code segments within complete program units
 - a) if separate unit can be in the same scope as the code that can cause the handler to be raised
 - b) can be outside the scope of the unit raising the exception and can communicate via parameters
 2. binding the exception occurrence to an exception handler
 - a) at the program unit level - it should be possible to bind the exceptions that can be raised by particular statements to particular handlers (same exception can be raised by many different statements)
 - b) above the program unit level - choose whether or not to propagate the exception to some other unit
 3. static or dynamic binding of exceptions to handlers
 - a) syntactic layout of the program
 - b) execution sequence
 4. what to do when the handler completes its execution
 - a) transfer to somewhere in a program outside of the handler code (continuation)
 - b) execution can terminate
 5. if users can define exception, how are they specified?
 - a) declare in specification parts of the programs in which they can be raised
 6. for languages with built-in exception handlers
 - a) should the language run-time system provide default handlers for built-in exceptions or should the user handle the exceptions?
 - b) can built-in exceptions explicitly be raised by a user program?
 7. can exceptions be temporarily or permanently disabled?
- D. 12.2 Exception Handling in PL/I
- E. 12.2.1 Exception Handlers
 1. 22 standard conditions cause exceptions in the execution of a PL/I program
 2. can specify user code or SYSTEM handler
 3. optional SNAP(shot) of dynamic chain of the program
- F. 12.2.2 Binding Exceptions to Handlers
 1. dynamic binding - there can be more than one statement for a given exception
 2. the ON binding stays in effect from the place it occurs until either a new ON occurs for the same exception or the block is exited
- G. 12.2.3 Continuation
 1. some handlers return to the statement causing the exception
 2. other conditions cause program termination
 3. user-defined handlers cannot access the address of the statement that

- caused the exception
- H. 12.2.4 Other Design Choices
 1. user-defined exceptions are called using SIGNAL
 2. built-in exceptions
 - a) those always enabled
 - b) those enabled by default but can be disabled by user code
 - c) those disabled by default that can be enabled by user code
 - I. 12.2.5 An Example
 - J. 12.2.6 Evaluation
 1. dynamic binding of exceptions to handlers causes a problem in writability and readability
 - K. Notes from PL/I Structured Programming, Second Edition, Joan Hughes, John Wiley, 1979.
 1. a condition is any occurrence within a PL/I program that could cause a program interrupt
 2. What action - if any - should be taken?



- 3.
4. Responses to Occurrence of Conditions
 - a) System Action - for most conditions, the standard system action is to print a message and then raise the ERROR condition; the standard system action for ERROR is to terminate the PL/I program and return control to the operating system
 - b) Program-Specified Action
 - (1) ON condition on-unit;
 - (2) when execution of an on-unit is successfully completed, control will normally return to the point of the interrupt or to a point immediately following it
 - c) Null action
 - (1) No action is to be taken for this on-unit
 - (2) ENDPAGE example for computer snoopy pictures

- (3) ON ENDPAGE (PRINTR);
- 5. Built-in functions to facilitate program debugging
 - a) ONCODE - integer that defines the type of interrupt that caused the unit to become active
 - b) ONLOC - character string indicating the name of the procedure in which the condition was raised
 - c) ONSOURCE - character string whose value is the contents of the field that was being processed when the CONVERSION condition was raised
 - d) ONCHAR - returns the character that caused the CONVERSION condition to be raised
- L. 12.3 Exception Handling in CLU
 - 1. procedure definition can include a list of exceptions that the procedure may raise
 - 2. exceptions can have parameters that are sent to the handler
- M. 12.3.1 Exception handlers
- N. 12.3.2 Binding of Exceptions to Handlers
 - 1. if no handler is found in the procedure, the built-in exception failure is raised
- O. 12.3.3 Continuations
 - 1. procedures that raise exceptions are normally terminated
 - 2. if handler is in the static environment of the calling statement it is executed
 - 3. if the handler is attached to a statement, control flows to the following statement after the handler is executed
 - 4. if handler appears at the end of a procedure, continuation is to the caller of the procedure that contains the exception handler
 - 5. can transfer control from statement to end of the procedure (exit)
- P. 12.3.4 Other Design Choices
 - 1. can explicitly raise exceptions with signal
- Q. 12.3.5 Example
- R. 12.3.6 Evaluation
- S. 12.4 Exception Handling in ADA
 - 1. CONSTRAINT_ERROR
 - 2. NUMERIC_ERROR
 - 3. PROGRAM_ERROR
 - 4. STORAGE_ERROR
 - 5. TASKING_ERROR
- T. 12.4.1 Exception Handlers
 - 1. usually local to the code in which the exception can be raised
 - 2. others keyword indicates that the handler is meant to handle exceptions not named elsewhere locally
 - 3. handlers are gathered together in an exception clause which must be placed at the end of the block or unit
- U. 12.4.2 Binding Exceptions to Handlers
 - 1. exception propagation in procedures
 - a) to the calling program unit at the point of the call if the called procedure has no handler for the exception
 - b) if it propagates to the main unit and no handler has been specified, the program terminates
 - 2. exception propagation in packages

- a) if no handler is in the package body, the exception is implicitly propagated to the declaration section of the unit containing the package declaration; if it's a library unit, the program is terminated
 - 3. exception propagation in tasks
 - a) if no handler, the task is marked as completed
 - 4. exception in the elaboration of the declarative sections of subprograms, blocks, packages, and tasks
 - V. 12.4.3 Continuation
 - 1. the block or unit that raises an exception, along with all units to which the exception was propagated but which did not handle it, is always terminated
 - 2. in a block, a statement can be retried after it raises an exception and that exception is handled
 - W. 12.4.4 Other Design Choices
 - 1. user defined exceptions
 - a) exception_name_list : exception
 - b) must be raised explicitly (raise)
 - 2. can suppress exception conditions
 - X. 12.4.5 example
 - Y. 12.4.6 Evaluation
- XII. Exception Handling in ANSI C, Colvin, C Users Journal, August 1991, pp 77-88.
 - A. five basic strategies for exception handling
 - 1. denial
 - 2. perfection
 - 3. paranoia
 - 4. truth
 - 5. communication
 - B. disciplined exceptions in Eiffel
 - C. disciplined exceptions in ANSI C
 - D. blocks of computation can be written as two clauses, similar to DO and RESCUE in Eiffel.
- XIII. Better Exception-Handling in Block-Structured Systems, Knudsen, IEEE Software, May 1987. pp 40-49.
 - A. exceptions and their handlers are defined separately and are associated with handlers either by binding imperatives or dynamic binding
 - B. proposes an exception-handling mechanism that declares the exception and its handler together
 - 1. uses static determination to decide which system parts should be destroyed when specific exceptions are raised
 - C. static exception handling
 - 1. the sequel concept - a sequel is an abstraction of the goto statement based on the procedure concept
 - 2. defines three aspects of exception handling
 - a) name of the exception
 - b) defines the handler associated with the exception
 - c) defines the termination level of the exception
 - 3. when the sequel body has been executed, control is transferred to the termination point of the block in which the sequel is declared
 - 4. consequences of the semantics of the sequel invocation
 - a) during the handling of an exception occurrence, it might be discovered

- that the exception cannot be handled locally, but it must be handled
by invoking a sequel in an outer block
- b) it might be discovered that the exception can be handled even more
locally, by a sequel declared in an inner block
- D. compares static and dynamic exception handling
 - E. termination - a hierarchical, cooperative exception-handling mechanism
 - 1. uses prefixed sequels
 - 2. uses default, virtual sequels also
 - F. unified terminology exception handling (sidebar)
 - 1. base definitions
 - 2. exception occurrences
 - 3. exception-handling model
 - a) resumption model
 - b) signalling model
 - c) termination model
 - 4. termination process
 - 5. association of handlers
 - 6. handler types
 - 7. parameterization
 - 8. default exception handling
 - 9. equivalence of user-defined and language-defined exceptions