

605.404 Object-Oriented Programming with C++
Summer 2007
Lecture 01

1	06/06	Intro to semester; programming style and documentation conventions	S: Chapters 1-3, Sections 24.3.7.1-24.3.7.3 J: Chapters 1-3
---	-------	--	--

Additional Resources Related to This Lecture:

Concepts of Programming Languages, Robert W. Sebesta, Benjamin/Cummings, 1993.

Introduction to the Theory of Programming Languages, Bertrand Meyer, Prentice Hall, 1990.

lex & yacc, Second Edition, Levine, Mason, and Brown, O'Reilly & Associates, Inc., October 1992.

Compiler Design in C, Allen I. Holub, Prentice-Hall, 1990.

Introduction to Compiler Construction, Thomas Parsons, Computer Science Press, 1992.

"Applications of Formal Methods: Developing Virtuoso Software", Susan Gerhart, IEEE Software, September 1990. pp 7-10.

"Seven Myths of Formal Methods", Anthony Hall, IEEE Software, September 1990, pp 11-19.

"Integrating Formal Methods into the Development Process", Richard Kemmerer, IEEE Software, September 1990, pp 37-50.

"A Specifier's Introduction to Formal Methods", Jeannette Wing, IEEE Computer September 1990, pp 8-24.

"On Formalism in Specifications", Bertrand Meyer, IEEE Software, January 1985, pp 6-26.

"Formal Support Environments", Lafontaine, Ledru, and Schobbens, Comm ACM, May 1991, Vol 34, No 5, pp 63-71.

"An Overview of the ISO/VDM-SL Standard", Nico Plat and Peter Larsen, ACM SIGPLAN Notices, Vol 27, No 8, August 1992, pp 76-82.

"Using ABC to Prototype VDM Specifications", Aaron Kans and Clive Hayton, ACM SIGPLAN Notices, Vol 29, No 1, January 1994, pp 27-36.

Techniques for Formal program Specification, CS-TR-1980, James Gray, III, Univ. of MD, Computer Science Technical Report Series, January 1988.

Composing Specifications, Martin Abadia and Leslie Lamport, Digital Systems Research

Center, Research Report 66, October 10, 1990.

Applying "Design by Contract", Meyer, IEEE Computer, October 1992, pp 40-51.

Debugging in C - An Overview, Baldwin, C Users Journal, October 1991, pp 50-63.

Debugging with Assertions, Bates, C Users Journal, October 1992, pp 40-46

LECTURE NOTES:

- I. Review the syllabus
 - A. How to reach me this semester
 - B. course description from catalog
 - C. prerequisites
 1. C++
 2. Java
 - D. Computer Requirement
 1. C++ at JHU
 - a) Student account, Solaris, GNU port version 3.4.3 (command – g++)
 2. C++ on your resources
 - a) Cygwin port <http://www.cygwin.com>, GNU port version 3.4.4 (command – g++)
 - b) Delorie djgpp <http://www.delorie.com> , GNU port version 4.1.2
 - (1) Sample compile command: `gcc hello.c -o hello.exe`
 - (2) Use `gxx` (or append `-lstdcx` to your gcc command line) to compile and link C++ programs
 - c) GNU <http://www.gnu.org/software/gcc/gcc.html>
 - E. instructional objectives and relationship to exam this semester
 - F. texts and related resource texts
 - G. Internet resources reserve holdings for this class
 1. Johns Hopkins Enterprise Directory (JHED), <http://jhed.jhu.edu>, requirement to access full-text materials online
 2. Review WWW postings for this class
 - H. reading
 1. homework/programming
 - a) style & documentation (more on the header later)
 - (1) C++ header located on WWW page
 - (2) RCS/ CVS keyword substitution
 - (3) suggested usage `rsm -cef -H -O *.*`
 - (4) Print program listings in such a way that lines do not wrap
 - b) test cases – must exercise code with path coverage tests
 - I. Exam.
 - J. grading policies
 - K. Attendance
 - L. Academic standards
 - M. Tentative class Plan
- II. Stroustrup Chapter 1 Notes to the Reader
 - A. Be sure you are familiar with the first three Appendices
 1. A – The C++ Grammar
 2. B – Compatibility

- 3. C – Technicalities
 - B. 1.1 The Structure of this Book
 - 1. Exercises – definition of difficulty page 6
 - C. 1.2 Learning C++
 - 1. “focus on concepts and not get lost in language-technical details”
 - 2. must consider C++-specific compiler behaviors and programming practices necessary for robust implementation and maintenance with C++ (as in Coplien’s text or Scott Meyers texts)
 - D. 1.3 The Design of C++
 - 1. 1.3.1 Efficiency and Structure
 - a) importance and efficiency of type system to check function arguments, to protect data from accidental corruption, to provide new types, and to provide new operators,...
 - b) emphasis on structure; amount of direction expressed in each statement
 - c) C++ in a group environment
 - (1) Use of CVS to support work
 - (2) Modularity,
 - (3) Strongly typed interfaces
 - (4) Flexibility
 - 2. 1.3.2 Philosophical Note
 - a) close to the machine
 - b) close to the problem to be solved
 - E. 1.4 Historical Note
 - F. 1.5 Use of C++
 - 1. C++ contribution to maintainability and testability (note that the U.S. government argued broadly against C++ on these accounts in favor of Ada)
 - 2. Numerical work in C++
 - G. 1.6 C and C++
 - 1. take note of 1.6.1 if you are proficient in C and C++ is new to you
 - H. 1.7 Thinking about Programming in C++
 - 1. generic steps: analysis, design, programming
 - 2. create a class when you need a new type (a class is a type)
 - 3. overview of design issues in class identification
 - 4. complexities of class relationships
 - I. 1.8 Advice
- III. Stroustrup Chapter 2 A Tour of C++
- A. 2.1 What is C++?
 - B. 2.2 Programming Paradigms
 - 1. support a style of programming versus enable a technique to be used
 - 2. important issue is not so much what features a language possesses but that the features it does possess are sufficient to support the desired programming styles in the desired application areas
 - C. 2.3 Procedural Programming
 - 1. Decide which procedures you want; use the best algorithms you can find
 - 2. lgstar.cpp, posted on the class WWW page, is an example of this style
 - 3. 2.3.1 Variables and arithmetic
 - a) Every name and every expression has a type that determines the operations that may be performed on it

- b) Declaration is a statement that introduces a name into the program
 - (1) Specifies the type for that name
 - (2) Type defines the proper use of a name or an expression
 - c) fundamental types page 24
 - d) arithmetic operators page 24 (note there is no exponentiation)
 - e) comparison operators page 24
 - f) caution about assignment and equality operators
 - 4. 2.3.2 Tests and Loops
 - a) cout and cin standard streams page 25
 - b) << (put to)
 - c) >> (get from)
 - d) If
 - e) Switch
 - f) while
 - 5. 2.3.3 Pointers and arrays
 - a) [] array of
 - b) * pointer to
 - c) & address of
 - d) Also introduces “for” statement
 - D. 2.4 Modular Programming
 - 1. Decide which modules you want; partition the program so that data is hidden within modules
 - 2. (data-hiding principle)
 - 3. use of namespaces
 - a) grouping related data, functions, etc. into a module
 - b) scope resolution operator “::” introduced
 - c) C++ allows any declaration to be placed in a namespace
 - 4. 2.4.1 separate compilation (gcc instructions)
 - a) stack.h
 - b) \$ g++ -x c++ -c stack.c
 - c) \$ g++ -x c++ -o user.exe user.c stack.o
 - d) \$./user
 - e) -x c++ indicates language choice regardless of extension on file
 - f) -c indicates compile or assemble the source file but do not link
 - g) -o indicates place output in specified file name
 - h) -Wall for all common warning options
 - 5. delorie dgjpp notes:
 - a) Use gxx (or append -lstdcx to your gcc command line) to compile and link C++ programs (*.cc) with delorie port of gcc
 - b) See also <http://www.delorie.com/djgpp/doc/ug/> DJGPP User’s Guide online
 - 6. 2.4.2 exception handling via catch and throw blocks
 - a) Quick intro to the structure of the try – catch blocks and the handler class setup
 - E. 2.5 Data Abstraction
 - 1. modules defining types
 - a) leads to the centralization of all data of a type under the control of a type manager module
 - b) importance of the signature of the method

2. user-defined types
 - a) Decide which types you want; provide a full set of operations for each type
 - b) user-defined type is an ADT for Stroustrup in this text
 3. concrete types
 - a) (see section 25.2 page 766 ff for details on what is a concrete type
 - b) “each is the representation of a relatively simple concept with all the operations essential for the support of the concept”
 - c) (introduction of roles for constructors and destructors in class instance management)
 - d) Introduction of object lifetime page 34
 - e) Virtual keyword – means “may be redefined later in a class derived from this one”
 - f) A class that provides the interface to a variety of other classes is often called the polymorphic type
 - g) Introduction of class hierarchy bottom page 35
 4. abstract types
 - a) decouple the interface from the implementation
 - b) interface contains virtual function specifications
 - c) implementation defines details of virtual functions
 5. virtual functions
 - a) convert the name of a virtual function into an index into a table of pointers to functions that can be called at run-time (virtual function table (vtbl))
- F. 2.6 Object-oriented programming
1. problems with concrete types
 2. class hierarchies
 - a) Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance
- G. 2.7 Generic Programming
1. Decide which algorithms you want; parameterize them so that they work for a variety of suitable types and data structures
 2. templates introduced
 3. containers (not bags) - a class holding a collection of elements of some type
 4. 2.7.2 generic algorithms
 - a) Sequence – has a beginning and an end
 - b) Iterator refers to an element and provides an operation that makes the iterator refer to the next element of the sequence
 - c) End of a sequence may be a sentinel element, but does not have to be a sentinel element
 - d) Use dereference operator “*” to mean “access an element through an iterator
 - e) Use increment operator “++” to mean “make the iterator refer to the next element”

IV. Stroustrup Chapter 3 A Tour of the Standard Library

- A. 3.1 Introduction
- B. 3.2 Hello, world!
- C. 3.2 The Standard Library Namespace
 1. using namespace std;

- D. 3.4 Output
 - 1. character constants enclosed in single quotes
- E. 3.5 Strings
 - 1. C++ style
 - 2. C style – a zero terminated array of characters
- F. 3.6 Input
 - 1. default in switch statements
- G. 3.7 Containers
 - 1. a class with the main purpose of holding objects is called a container
 - 2. vector
 - 3. 3.7.2 range checking
 - a) NOT provided by class vector
 - b) Vec class used in subsequent chapters defined page 53
 - (1) Throws out of range error
 - 4. 3.7.3 list
 - a) Introduces begin() and end() methods for every container
 - 5. 3.7.4 map
 - 6. 3.7.5 standard containers
 - a) List page 56
- H. 3.8 algorithms
 - 1. 3.8.1 use of iterators
 - 2. 3.8.2 iterator types
 - 3. 3.8.3 iterators and I/O
 - 4. 3.8.4 traversals and predicates
 - a) A predicate is called for each element and returns a Boolean value, which the algorithm uses to perform its intended actions
 - 5. 3.8.5 algorithms using member functions
 - a) Mem_fun() template is used as a predicate and takes a pointer to a member function as its argument and produces something that can be called for a pointer to the member's class
 - 6. 3.8.6 standard library algorithms
 - a) List page 64
- I.3.9 Math
 - 1. 3.9.1 complex numbers
 - 2. 3.9.2 vector arithmetic (valarray)
 - 3. 3.9.3 basic numeric support
- J. 3.10 standard library facilities
- V. Stroustrup Chapter 24 Design and Programming
 - A. 24.3.7.1 Invariants
 - 1. objective is maintenance of well-defined states
 - B. 24.3.7.2 Assertions
 - C. 24.3.7.2 Preconditions and Postconditions
- VI. Josuttis Chapter 1
 - A. Text overview
- VII. Josuttis Chapter 2 Introduction to C++ and the Standard Library
 - A. 2.1 History
 - 1. Get the standard from ANSI for \$18 (see class link on WWW page)
 - 2. comparison of string class and STL pg 8
 - B. 2.2 New language features

1. 2.2.1 templates
 - a) Pg 10 You must have the implementation of a template function available when you call it, so that you can compile the function for your specific type
 - b) Therefore, the only portable way of using templates at the moment is to implement them in header files by using inline functions
 - c) Nontype template parameters
 - d) Default template parameters
 - e) Keyword “typename” – introduced to specify that the identifier that follows is a type; always necessary to qualify an identifier of a template as being a type, even if an interpretation that it is not a type would make no sense
 - f) Member templates (may not be virtual and may not have default parameters)
 - g) Template constructors defined page 13
 - h) Nested template classes allowed
 2. 2.2.2 explicit initialization for fundamental types
 - a) Fundamental types initialized with zeros iw you call an explicit constructor without arguments
 3. 2.2.3 Exception handling
 - a) Exception handling is not error handling
 4. 2.2.4 namespaces
 - a) Open for definitions and extensions in different modules
 - b) Koenig lookup (pg 17) – you don't have to qualify the namespace for functions if one or more argument types are defined in the namespace of the function
 5. 2.2.5 type bool
 6. 2.2.6 keyword explicit
 - a) You can prohibit a single argument constructor from defining an automatic type conversion
 7. 2.2.7 new operators for type conversion
 - a) Static_cast
 - b) Dynamic cast
 - c) Const_cast
 - d) Reinterpret_cast
 8. 2.2.8 initialization of constant static members
 9. 2.2.9 definition of main()
 - C. 2.3 complexity and the Big-O Notation
- VIII. Josuttis Chapter 3 General Concepts
- A. 3.1 Namespace std
 - B. 3.2 Header Files
 1. compatibility with “old” C-style headers through namespaces
 - C. 3.3 Error and Exception Handling
 1. 3.3.1 Standard Exception Classes
 - a) Exception Classes for Language Support
 - b) Exception Classes for the Standard Library
 - c) Exception Classes for Errors Outside the Scope of a Program
 - d) Exceptions Thrown by the Standard Library
 - e) Header Files for Exception Classes

2. 3.3.2 Members of Exception Classes
 3. 3.3.3 Throwing Standard Exceptions
 4. 3.3.4 Deriving Standard Exception Classes
- D. 3.4 Allocators
1. special objects to handle the allocation and deallocation of memory
 2. translates need to use memory into a raw call for memory

- IX. Seven Myths of Formal Methods, IEEE Software, Sept. 1990
 - A. Myth 1 - Formal Methods can guarantee that software is perfect
 - 1. some things can never be proved
 - 2. proofs can contain mistakes
 - B. Myth 2 - Formal methods are all about program proving
 - 1. really about specification
 - 2. specifications are abstract - does not use computer specific structures for objects, defines what not how, and gives only some detail, other left for later
 - C. Myth 3 - Formal methods are only useful for safety-critical systems
 - D. Myth 4 - Formal methods require highly trained mathematicians
 - E. Myth 5 - Formal methods increase the cost of development
 - F. Myth 6 - Formal methods are unacceptable to users
 - G. Myth 7 - Formal methods are not used on real, large-scale software
- X. Sebesta Chapter 3 Describing Syntax and Semantics
 - A. 3.5 Describing the Meanings of Programs
 - 1. 3.5.2 Axiomatic Semantics
 - a) structure of axiomatic semantics was defined with the primary goal of developing a method to prove the correctness of programs
 - b) each statement of a program is preceded and followed by a logical expression that specifies the constraints on and relationship among data values
 - c) the logical expressions are used to specify the meaning of the statement
 - d) notation used is predicate calculus
 - e) 3.5.2.1 Assertions
 - (1) precondition and postcondition
 - f) 3.5.2.2 Weakest Preconditions
 - (1) the weakest precondition is the least restrictive precondition that will guarantee the validity of the associated postcondition
 - (2) if the weakest precondition can be computed from the given postcondition for each statement of a language, then correctness proofs can be constructed for programs in the language
 - (3) axiom is a statement that is assumed to be true
 - (4) inference rule is a method of inferring the truth of one statement on the basis of other true statements
 - (5) an axiom or inference rule must be defined for each kind of statement in the language
- XI. Basic Concepts (Meyer OOSC Chapter 1)
 - A. Why or why not use formal descriptions
 - 1. support for program verification and software reliability
 - a) correctness and robustness of programs
 - b) proving program correctness
 - (1) associate with a program a mathematical transform
 - (2) use the mathematical proof techniques to check that this transform achieves the program's stated purpose
 - c) three kinds of problems involved in proving correctness of programs

- (1) software specification - state precisely the purpose of each program or program element
 - (2) develop the right kind of mathematical theories to reason about programs and prove properties of their behavior
 - (3) need efficient tools to assist in this process because it is tedious and error prone
- B. limitations of formal specifications
- 1. difficult - hard to learn, write, read
 - 2. need some mathematical ability and substantial effort
 - 3. seem to apply only to toy problems and languages
 - 4. hard to formally specify concurrency, floating point arithmetic, complex data structures
- C. programs versus mathematics
- 1. imperative language focus (instruction, sequencing, assignment)
 - a) imperative or operations style of programming characterized by the presence of constructs describing commands issued to a machine
 - b) the construct that makes programming languages prominently imperative is the instruction
 - (1) note on instruction versus statement
 - (2) like wrong and incorrect
 - c) explicit sequencing - explicit control structures to specify precisely the order their instructions are executed
 - d) assignment - this instruction is a command; it does not assert a property, but orders the machine to change something in its internal state
 - (1) instruction which works by side effect on the program state
 - 2. referential transparency
 - a) this is a property of mathematical notation of substitutivity of equals for equals (allows substitution)
 - b) programming languages do not have this property if they permit side effects
 - (1) example assumes variable y is global (you can tell because its not declared in the parameter list or list of local variables)
 - (2) aliasing - whenever a given object becomes accessible through more than one name
 - (a) argument passing by reference between a global variable and a subprogram argument
 - c) denotational semantics provides a model of instruction sequencing based on the mathematical notation of function composition
 - d) axiomatic semantics describes the effect of an assignment through the mathematical notion of substitution

XII. Syntax [Meyer OOSC Chapter 3]

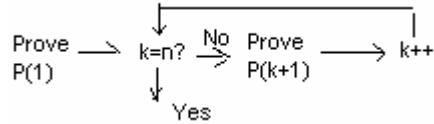
- A. Need for static semantics
 - 1. look at conditional and loop
 - a) need a test role tag which is defined as an expression
 - b) but expressions can be constant, variable, or boolean; boolean is the only one that is valid
 - c) some constraints cannot be expressed as part of a grammar

- (1) no variable should be declared twice in a program
 - d) could add Boolean_expression and Integer_expression; but duplicates productions and makes the grammar longer
 - 2. static semantics is the description of the structural constraints that cannot be adequately captured by the syntax descriptions
 - a) static semantics cover the gap between formalisms and the structural description needs of actual programming languages
 - 3. dynamic semantics the description of the effects of program execution
- XIII. Semantics: the main approaches [Meyer OOSC Chapter 4]
 - A. Major grammars studied in this book
 - 1. axiomatic semantics
 - a) views the definition of a programming language as the language's semantics a theory of programs written in the language
 - b) formulae and theorem
 - c) three components
 - (1) syntactic rules
 - (2) axioms
 - (3) inference rules
 - d) importance of pre-post formulae (pages 110-111)
- XIV. Axiomatic Semantics [Meyer OOSC Chapter 9]
 - A. Overview
 - 1. expresses the semantics of a programming language by associating with the language a mathematical theory for proving properties of programs written in the language
 - 2. denotational model is more an abstract implementation than a pure specification
 - 3. axiomatic method defines proof rules which make it possible to reason about properties of programs
 - 4. comparison of denotational and axiomatic semantics and erroneous computations
 - a) denotational must state what the error inducing conditions are and account for them in the denotations
 - b) axiomatic makes sure no proof rule applies to them (unobtrusive approach to erroneous cases and undefinedness)
 - 5. what are axiomatic semantics good for?
 - a) program verification
 - b) understanding and standardizing languages
 - c) providing help in program construction
 - B. The notion of a theory
 - 1. a theory about a particular set of objects is a set of rules to express statements about those objects and to determine whether any such statement is true or false
 - 2. Forms of Theories
 - a) grammar defines the meaningful statements of the theory (well-formed formulae)
 - b) semantic rules of the theory (axioms and inference rules) apply only to well-formed formulae determine which formulae are theorems
 - 3. axioms
 - a) an axiom is a rule which states that a certain formula is a theorem

- b) refers to actual expressions
- 4. rule (axiom) schemata
 - a) an axiom schema refers to arbitrary expressions
- 5. inference rules
 - a) inference rules are mechanisms for deriving new theorems from others
 - (1) if f_1, f_2, \dots, f_n are theorems, then f_0 is a theorem
 - (2) formulae above the horizontal bar are antecedents (premises)
 - (3) formula below the line is its consequent (conclusions)
 - b) modus ponens - makes possible use of implication in inferences
 - (1) means method of affirming
 - (2) (if p then q ; p) therefore q
 - c) modus tollens
 - (1) means method of denying
 - (2) (if p then q ; not p) therefore not q
 - d) induction
- 6. Loop invariants and the correctness of insertion sort (From Chapter 4 of Discrete Mathematics with Applications, Susanna S. Epp, 1990, Wadsworth)
- 7. [pre-condition for loop]
 - a) while (G)
 - (1) [Statements in loop body. None contain branching
 - (2) statements that leap outside the loop]
 - b) end while
 - c) [post-condition for loop]
- 8. A loop is defined to be correct with respect to its pre- and post-conditions if, and only if, whenever the algorithm variables satisfy the pre-condition for the loop and the loop is executed, then the algorithm variables satisfy the post-condition for the loop.
- 9. G is the guard, which restricts entry to the loop
- 10. Loop Invariant Theorem
 - a) Let a while loop with guard G be given, together with pre- and post-conditions that are predicates in the algorithm variables. Also let a predicate $I(n)$, called the loop invariant, be given. If the following four properties are true, then the loop is correct with respect to its pre- and post-conditions
 - b) Basis Property: The pre-condition for the loop implies that $I(0)$ is true before the first iteration of the loop.
 - c) Inductive Property: If the G and the loop invariant $I(k)$ are both true for an integer $k \geq 0$ before an iteration of the loop, then $I(k + 1)$ is true after iteration of the loop.
 - d) Eventual Falsity of Guard: After a finite number of iterations of the loop, the guard G becomes false.
 - e) Correctness of the Post-Condition: If for some nonnegative integer N , G is false and $I(N)$ is true, then the values of the algorithm variables will be specified in the post-condition of the loop
- 11. Principle of Strong Mathematical Induction (Epps page 249)
 - a) Let $P(n)$ be a predicate that is defined for integers n , and let a and b be fixed integers with $a \leq b$. Suppose the following two statements are true:

- (1) $P(a), P(a+1), \dots$ & $P(b)$ are all true [BASIS STEP]
- (2) For any integer $k > b$, if $P(l)$ is true for all integers l with $a \leq l < k$, then $P(k)$ is true [INDUCTIVE STEP]
- (3) Then the statement

- (a) For all integers $n \geq a$, $P(n)$ is true
- b) (The supposition the $P(l)$ is true for all integers l with $a \leq l < k$ is called the inductive hypothesis)



- c)
- 12. At the beginning of each iteration of the “outer” for loop, which is indexed by j , the subarray of $A[1 \dots j-1]$ constitute the sorted hand and the elements $A[j+1 \dots n]$ correspond to the cards on the table.
 - a) Properties of $A[1 \dots j-1]$ as a loop invariant
 - (1) At the start of each iteration of the for loop in lines 1-8, the subarray $A[1 \dots j-1]$ consists of the elements originally in $A[1 \dots j-1]$ but in sorted order
 - b) Must show three things
 - (1) Initialization: it is true prior to the first iteration of the loop
 - (2) Maintenance: if it is true before an iteration of the loop, it remains true before the next iteration
 - (3) Termination: when the loop terminates, the invariant gives us a useful property that helps show the algorithm is correct
- 13. proofs
 - a) definition (Theorem): A theorem t in a theory is a well-formed formula of the theory, such that t may be derived from the axioms by zero or more applications of the inference rules
 - b) structure of proof governed by these rules:
 - (1) proof is a sequence of lines
 - (2) each line is numbered
 - (3) each line contains a formula, which the line asserts to be a theorem (each formula preceded by an implied $\{$)
 - (4) each line contains an argument showing unambiguously that the formula is indeed a theorem (justification of the line)
 - (a) the name of an axiom or axiom schemata of the theory in which case the formula must be the axiom itself or an instance of the axiom schemata
 - (b) a list of references to previous lines, followed by a semicolon and the name of the inference rule or inference schemata of the theory
- 14. interpretations and models
 - a) an interpretation associates a member of some mathematical domain with every element of the theory's vocabulary in such a way that a boolean property of the domain is associated with every well-formed formula
 - b) a model is an interpretation which associates a true property with every theorem of the theory

C. axiomatizing programming languages

1. assertions
 - a) a property of the program's objects which may or may not be satisfied by a state of the program during execution
 - b) (will be expressed as boolean expressions in concrete syntax for now)
 2. preconditions and postconditions (programming by contract)
 - a) preconditions - assumed to be satisfied before the fragment is executed
 - (1) should be as broad as possible
 - (2) it is the responsibility of the environment to invoke the program or program fragment only for cases that fall within the precondition
 - b) postconditions - guaranteed to be satisfied after the fragment has been executed
 - c) a program or program fragment will be said to be correct with respect to a certain precondition P and a certain postcondition Q IFF when executed in a state in which P is satisfied it yields a state in which Q is satisfied
 - d) precondition obligates the environment, postcondition obligates the program
 - e) programming by contract;
 - f) defined this way program correctness is only a relative concept...
 3. definitions of total and partial correctness
 - a) definition (Total Correctness): A program fragment a is totally correct for P and Q IFF the following holds: whenever a is executed in any state in which P is satisfied, the execution terminates and the resulting state satisfies Q.
 - (1) achieves the postcondition
 - (2) and terminates
 - b) definition (Partial Correctness): A program fragment a is partially correct for P and Q IFF the following holds: whenever a is executed in any state in which P is satisfied and this execution terminates, the resulting state satisfies Q.
 - (1) achieves the postcondition
 - (2) if it terminates
 - c) but programs must terminate
 4. varieties of axiomatic semantics
 - a) [Floyd is a big name in algorithm development for graphs and networks]
 - b) pre-post formulae (Hoare)
 - (1) set up this way to match comments in Pascal
 - (2) proves partial correctness; termination proved separately
 - c) a Hoare theory of programming language consists of axioms and inference rules for deriving certain pre-post formulae (pre-post semantics)
 - d) Dijkstra and a calculus of programs (wp-semantics); wp stands for weakest precondition
- D. a closer look at assertions
1. assertion defined as a property of program objects, which given a state of program execution may or may not be satisfied
 2. differences between assertions and boolean expressions
 - a) boolean expressions appear in programs; they belong to the

- programming language
 - b) assertions express properties about programs; they belong to the formulae of the axiomatic theory
- 3. implication
 - a) any state that satisfies P satisfies Q
 - b) when P implies Q holds but Q implies P does not, P will be said to be stronger than Q and Q weaker than P
- E. fundamentals of pre-post semantics
 - 1. formulae of interest in pre-post semantics
 - a) interpretation of pre-post formulae: a pre-post formula $\{P\} a \{Q\}$ expresses that a is partially correct with respect to precondition P and postcondition Q
 - 2. rule of consequence
 - a) less informative formulae may be deduced from ones that carry more information
 - 3. facts from elementary mathematics
 - a) axiomatic theories for programming languages may have to embed other non-programming language specific theories
 - b) remember that numbers have representational limits on computers
 - c) EM - axioms and inference rules from elementary math and properties of mathematical implication
- F. the calculus of weakest preconditions (complementary viewpoint to pre-post semantics)
 - 1. introduction
 - a) weakest precondition approach based on these observations
 - (1) it defines the semantics of a programming language through a set of rules which associate with every construct an assertion transformer (a mechanism that associates a with given precondition of postcondition the most interesting postcondition or precondition (respectively) which corresponds to it through the instruction)
 - (2) yields for any postcondition the weakest corresponding precondition
 - b) three other differences from pre-post theory
 - (1) it does not require, at least in principle, the invention of a variant and invariant
 - (2) it directly handles total correctness
 - (3) it deals with non-deterministic constructs
- G. Review nondeterministic algorithm definition
 - 1. deterministic algorithms - property that the result of every operation is uniquely defined
 - 2. non-deterministic algorithms
 - a) contain operations whose outcome is not uniquely defined but is limited to a specified set of possibilities. the machine using the algorithm is allowed to choose any one of these outcomes subject to a termination condition
 - b) a non-deterministic algorithm terminates unsuccessfully iff there exists no set of choices leading to a successful signal
 - c) non-determinism (a non-deterministic instruction is one whose effect in

- not entirely characterized by the state in which it is executed)
- 3. guarded conditional
 - a) keeps you from dumping all the other condition processing in the last else
 - b) "in the guarded conditional, on the other hand, every branch is explicitly preceded by its guard and executed only if the guard is true. If no guard is satisfied, a good implementation will produce an error message and stop execution, or raise an exception, or loop forever; this is better than proceeding silently with a wrong computation"
 - c) has three distinct features
 - (1) it treats the various possible cases in a more symmetric way than the if..then..else conditional
 - (2) it is non-deterministic
 - (3) it may fail - produce an undefined result
- 4. guarded loop
- 5. role of demons in non-determinism
- 6. rule of disjunction in non-determinism
- H. routines and recursion
 - 1. routines with no arguments and no result (!)
 - 2. introducing arguments
 - a) in, out, in-out arguments
 - b) focus on arguments and results (in and out) though
 - c) use one argument list and one result list
 - d) procedure call
 - (1) stands for an instruction
 - (2) does not return results
 - e) function call
 - (1) stands for an expression
 - (2) returns a result
 - f) see page 349
 - 3. simultaneous substitution
 - a) substitution defined for a list
 - 4. conditions on arguments and results
 - a) no identifier may occur twice in the formal argument list
 - b) no identifier may occur twice in the formal result list
 - c) no identifier may occur in both the formal argument and formal result lists
 - d) no variable local to the body of the routine may have the same name as a variable accessible in the calling program unit, unless it occurs in neither the precondition P or postcondition Q
 - e) no element of the result list may appear in P
 - f) if the postcondition Q involves any element of the argument list, then the corresponding element of the actual input list may not occur in the output list
- I. assertion-guided program construction (constructive approach to program correctness)
 - 1. unless you start this way its hard to produce probably correct programs
 - 2. the assertions required to prove correctness are best made during design
 - 3. even if you can't prove correctness its a better start

- J. assertions in programming languages
 - 1. if language support for assertions exists
 - a) path from spec to design becomes smoother; first phase produces assertions, next ones yield instructions which satisfy the corresponding pre-post formulae
 - b) if the language includes a formal assertion sublanguage, software tools can extract the assertions and produce automatic high level documentation about the element
 - c) compiler may have an option of generating assertion checking code at runtime
 - d) assertions also have a direct connection with the important issue of exception handling
 - 2. I think you can do all the Eiffel type items in C or C++ if you choose to stick to a format designed to place pre and post condition statements in the code
 - 3. Eiffel
 - a) variant, invariant and initialization clauses for loops
 - b) require and ensure clauses in routines
 - c) classes have class invariants which express global properties of the class's instances (pre-post conditions of every exported routine of a class)
 - d) check instruction to ensure assertion at other times
 - e) old which refers to the value an expression had on routine entry
- K. advantages of assertions on programming languages
 - 1. need
 - a) syntactic inclusion of invariant, variant, and initialization clauses in loops
 - b) require and ensure clauses in routines to support the principle of "programming by contract"
- XV. Meyer OOSC Chapter 7 Systematic approaches to program construction
 - A. 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
 - B. 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
 - C. 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
- D. 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
- E. 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
- F. 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
- XVI. Applying "Design by Contract", Meyer, IEEE Computer, October 1992, pp 40-51.
 - A. introduction
 1. methodology for Object-oriented software construction whose main goal is to help improve the reliability of software systems
 2. contributions of the work
 - a) a coherent set of methodological principles helping to produce correct and robust software
 - b) a systematic approach to the delicate problem of how to deal with

- abnormal cases, leading to a simple and powerful exception-handling mechanism
 - c) a better understanding of inheritance and of the other associated techniques through the notion of subcontract, allowing a systematic approach to using these powerful, but sometimes dangerous mechanisms
 - 3. defensive programming revisited
 - 4. the notion of contract
 - 5. assertions: contracting for software
 - 6. the role of assertions
 - 7. observations on software contracts
 - 8. who should check?
 - 9. class invariants
 - 10. assertion language requirements (sidebar)
 - 11. documenting a software contract
 - 12. monitoring assertions
 - 13. why monitor?
 - 14. introducing inheritance
 - 15. concurrency issue (sidebar)
 - 16. invariants and dynamic binding
 - 17. dealing with abnormal situations
 - 18. a disciplined exception-handling mechanism
 - a) when a routine includes a rescue clause, any exception occurring during the routine's execution interrupts the body (the Do clause) and starts execution of the rescue clause
 - b) rescue clause
 - (1) contains zero or more instructions
 - (2) may include a Retry
 - c) execution terminates
 - (1) if the rescue clause terminates without executing a retry (organized panic case)
 - (2) if the rescue clause executes a Retry, the body (Do clause) is executed again
 - d) in a typical system, only a handful of routines will have an explicit rescue clause
 - (1) an absent rescue clause is considered equivalent to an implicit clause of the form `rescue default_rescue`
 - (2) defined this way to restore the invariant; a null action would not achieve this for a class with a nontrivial invariant
 - e) differences between Do and rescue clauses
 - (1) body must implement the contract, or ensure the postcondition
 - (2) rescue clause clause has no precondition and is required on exit to restore the invariant

XVII. Debugging in C - An Overview, Baldwin, C Users Journal, October 1991, pp 50-63.

XVIII. Debugging with Assertions, Bates, C Users Journal, October 1992, pp 40-46

A. introduces Euclid tie to C assert

- 1. an automatic theorem prover could be integrated with a compiler to derive some assertions from others

2. the theorem prover would delete any assertions that it could prove before code generation
 3. compiler could generate assertions that were not in the source code at all, instead of traditional runtime checks
- B. ensurer code and relyer code
- C. invariant uses some portion of code as both relyer and ensurer
- D. where to locate assertions
1. look out for things that could cause serious damage if not detected
 2. check for things that are not easily verified by examining a localized section of code
 3. put assertions at the site of relyer code, not after ensurer code
 4. look for simple consistency checks
 5. at the start of a function body
 6. don't confuse assertions with error checking
 7. parameter validity checks should be error checks rather than assertions if you think of the calling code as your user
 8. programs producing files are the ensurers of invariants about their internal structure
 9. treat consistency checks on special files as a user error
- E. comments about leaving assertions active in delivered code