CS 1704

Introduction to Data Structures and Software Engineering

Module 1 Outline

- What is Software Engineering?
- What is Requirements Engineering
- What is Design?
  - Structure Charts
- What is Testing?
- Intro to Pointers
- Intro to ADT's
- Intro to Classes

What does Correctness Mean?

- "Program correctness" is not easily defined.
  - The programmer and user of a program may interpret "correctness" quite differently and hence have different expectations of program performance. Various interpretations of correctness are listed below in order of increasing difficulty of achievement:

Correctness Continued

- 1. The program contains no syntax errors that can be detected during translation by the language processor.
- 2. The program contains no errors, either of syntax or invalid operation, that can be automatically detected during translation or execution of the program.

Correctness Continued

- 3. There exists some set of test data for which the program will yield the correct answer.
- 4. For a typical (reasonable or random) set of test data the program will yield the correct answer.

Correctness Continued

- 5. For deliberately difficult sets of test data the program will yield the correct answers.
- 6. For all possible sets of data that are valid with respect to the problem specification, the program yields the correct answers.

Correctness Continued

- 7. For all possible sets of valid test data, and for all likely conditions of erroneous input, the program gives a correct (or at least reasonable) answer.
- 8. For all possible input, the program gives correct or reasonable answers.

Correctness Continued

- Some programmers never mature beyond level 3 in their attitude toward correctness.
- From the user's point of view a reasonable definition of correctness is certainly not less than level 6. Level 7 is better and level 8 is what he would really like.
- The programmer may maintain that a literal interpretation of problem specifications cannot demand more than level 6, while the user will maintain that certain implied requirements do not have to be explicitly stated.

Obtaining Correctness

May be Incorrect

- Deskchecking
  - Code traces
- Walkthrough
  - Expert Review Board:
    - Mental execution of text cases before the review board
  - Code Inspections
    - Programmer explains logic of code (statement by statement) formally to a peer group

Demonstrated Some Correctness

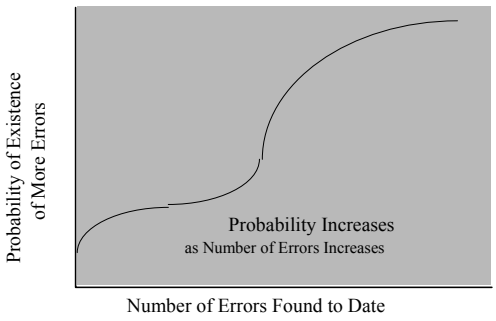
Correctness is Unobtainable

Demonstrated Some Correctness

- Testing
  - Case data
  - Computer aided
- Proof
  - Formal (mathematical)

Proven Correct

Discovered vs Undiscovered Errors

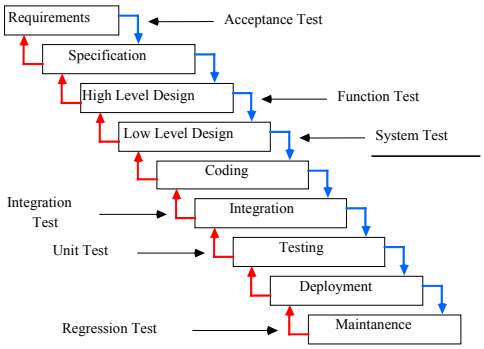


Testing

- 40-50% of all development time is spent in the testing process
- Humans (programmers) are NOT good at testing. The process of testing admits that one has produced code with errors.
- Successful testing can be thought of as successfully finding errors and testing failure implies not discovering any errors

"Testing can establish the presence of errors, but never their absence." **-Edsger Dijkstra**

Testing Phases



Testing Phases Defined

- Regression Testing involves fixing errors during testing and the re-execution of all previous passed tests.
- Unit Testing utilizes module testing techniques (white-box / black-box techniques).
- Integration Testing involves checking subsets of the system.
- Acceptance, Function and System testing is performed upon the entire system.

Integration Testing

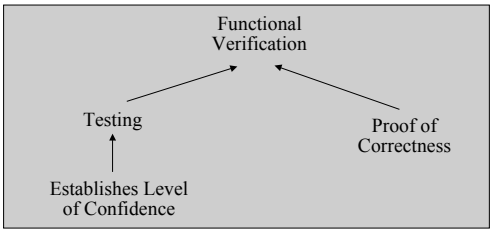
- Bottom-Up Testing
  - Unit Test (Black & White box techniques)
  - discovers errors in individual modules
  - requires coding (& testing) of driver routines
- Top-Down Testing
  - Main module & immediate subordinate routines are tested first
  - requires coding of routine stubs to simulate lower level routines
  - system developed as a skeleton

Integration Testing cont'd

- Sandwich Integration
  - combination of top-down & bottom-up testing
- Big Bang
  - No integration testing
  - modules developed alone
  - All modules are connected together at once

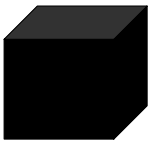
Function Testing

- Proof of Correctness vs. Testing



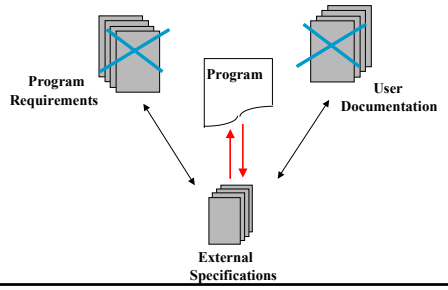
Function Testing cont'd

- Checks that the system satisfies its external specification
- Entire system is viewed as a "Black Box"
- Techniques:
  - Boundary-value Analysis
  - Cause-Effect Graphing



Function Testing cont'd

- Doesn't verify that the requirements or user documentation are correct

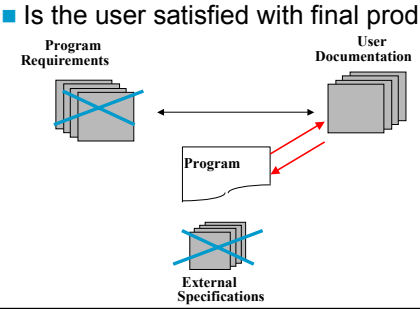


Acceptance Testing

- Tests the program against the current needs of the users and its original objectives.
- Usually performed by the end user (customer)
- Contract may require, as part of acceptance test:
  - performance tests (throughput, statistics collection, ...)
  - stress tests (system limits)
- If performed by system developers may consist of α (alpha), β (beta) testing

Acceptance Testing cont'd

- Is the user satisfied with final product?

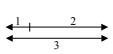


Testing Example

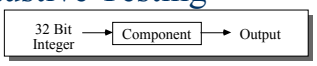
- Program reads 3 integer values from a line.
- The 3 values represent the lengths of the sides of a triangle.
- The program outputs whether the triangle is equilateral, isosceles, or scalene.
- Write a set of test cases which would **adequately** test this program!

Test cases

- Valid scalene triangle.
- Valid equilateral triangle.
- Valid Isosceles triangle.
- All possible permutations of Isosceles triangles (e.g. (3,3,4) (3,4,3) (4,3,3))
- One side having a zero value.
- One side having a negative value.
- Degenerate Triangle (e.g. 1-Dim Δ (1,2,3)
- All possible permutations of Degenerate Triangles (e.g. (1,2,3) (3,1,2) (1,3,2))
- Invalid Triangle (e.g. (1,2,4))
- All possible permutations of invalid triangles.
- All sides = 0.
- Non-integer values.
- Incorrect number of sides
- …



Exhaustive Testing



- How long will it take to try all possible inputs at a rate of one test/second?

2^{32} tests * 1 second / test

$= 2^{32}$ seconds

$= 2^{32}$ / (60 * 60 * 24 * 365) years

$> 2^{32} / (2^6 * 2^6 * 2^5 * 2^9)$ years

$= 2^{32} / 2^{26}$ years

$= 2^6$ years = 64 years

Impractical!!!

Goal of Testing

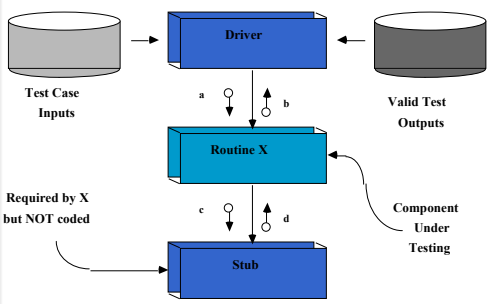
- No method (Black/White Box, etc.) can be used to detect all errors.
- Errors may exist due to a testing error instead of a program error.
- A finite number of test cases must be chosen to maximize the probability of locating errors.

Perform testing to ensure that the probability of program/system failure due to undiscovered errors is acceptably small.

Testing Principles

- The expected output for each test case should be defined **in advance** of the actual testing.
- The test output should be **thoroughly inspected**.
- Test cases must be written for **invalid & unexpected**, as well as valid and expected input conditions.
- Test cases should be **saved and documented** for use during the maintenance / modification phase of the life cycle.
- New test cases must be added as new errors are discovered.
- The test cases must be a **demanding exercise** of the component under test.
- Tests should be carried out by a third party independent tester, developer engineers should not privatize testing due to **conflict of interest**
- Testing must be planned as the system is being **developed**, NOT after coding.

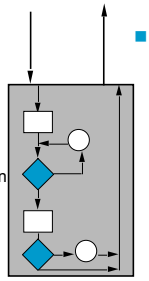
Drivers vs. Stubs



White Box Testing

Correct I/O relationships are verified using:

Functional Description and actual implementation



- Structural Testing
  - Exercise of Source code and internal data structures
  - Test cases are derived from analysis of internal module logic and external module specifications

White Box: Logic Testing

- Statement Coverage
  - Every statement is executed at least once.
- Decision Coverage
  - Each decision is tested for TRUE & FALSE.
  - Correctness of conditions within the decisions are NOT tested
- Condition Coverage
  - Each condition in a decision takes on all possible outcomes at least once.
  - Does not necessarily test all decision outcomes.
  - Test cases do not take into account how the conditions affect the decisions.

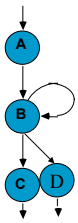
White Box: Logic Testing cont'd

- Decision/Condition Coverage
  - Satisfies both decision coverage and condition coverage.
  - Does NOT necessarily test all possible combinations of conditions in a decision.
- Multiple Condition Coverage
  - Test all possible combinations of conditions in a decision
  - Does not test all possible combinations of decision branches.

```
Eg: if (x >= 5 && y != 7 && z= =9)
    {     ;//2^3=8 tests!   }
```

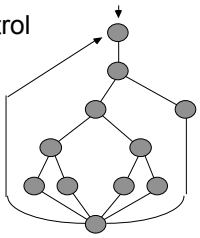
White Box: Path Testing

- Exercise a program by testing all possible execution paths through the code.
- Method
  - Enumerate the paths to be tested
  - Find the Input Domain of each
  - Select 1 or more test cases from domains
- Problem: Loops (∞ number of paths)
  Paths: ABC; ABBC; AB ... BC
- Solution:
  - Restrict loop to N iterations
- Small number of paths that yield reasonable testing



Control Flow Graph

- Node: sequence of statements ending in a branch
- Arc: transfer of control

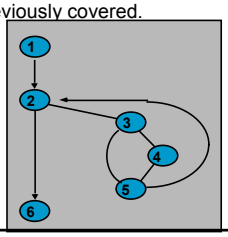


Exhaustive Path Testing

- (analogue of exhaustive input testing)
- requires executing the total number of ways of going from the top of the graph to the bottom
- approx. 100 trillion, where 5 = number of unique paths
- assuming all decisions are independent of each other
- specification errors could still exist
- does not detect missing paths
- does not check data-dependent errors

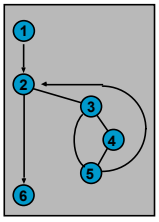
Test Path Determination

- Independent Path
  - any path that introduces at least one new set of processing statements (nodes), i.e. it must traverse an edge not previously covered.
- Independent Paths:
  - 1 - 2 - 6
  - 1 - 2 - 3 - 5 - 2 - 6
  - 1 - 2 - 3 - 4 - 5 - 2 – 6



Cyclomatic Complexity

- Upper bound on the number of independent paths, i.e. number of tests that must be executed in order to cover all statements.
- CC
  - edges - Nodes + 2
    = E - N + 2
    = 7 - 6 + 2 = 3

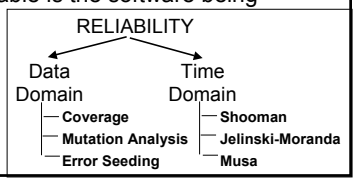


When to Stop Testing

- When no more errors exist. (Impossible to ascertain.)
  - (1) How reliable is the set of test cases?
    - Data Domain
  - (2) How reliable is the software being developed?
    - Time Domain

When to Stop Testing

- When no more errors exist. (Impossible to ascertain.)
  - (1) How reliable is the set of test cases?
    - Data Domain
  - (2) How reliable is the software being developed?
    - Time Domain

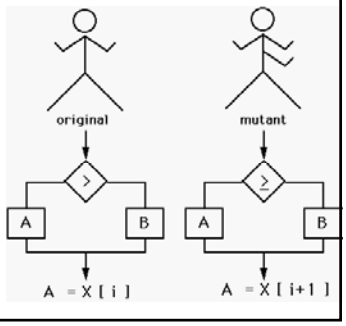


Reliability: Time Domain

- Time Domain Reliability
  - MTBF : mean time between failures
  - MTTF : mean time to failure
  - MTTR: mean time to repair
- MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR) * 100
- Estimate Methods:
  - Predictions based on calendar time
  - Predictions based on CPU time

Data Domain: Mutation Analysis

- Tests the test data
- Mutate Code to determine the adequacy of the test data.
- Determines whether all deliberately introduced (mutant) errors are detected by the original test cases.



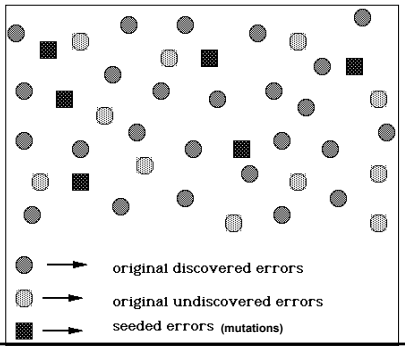
Mutation Analysis cont'd

- Mutation Testing Process
  - Program P is executed for test case T
  - If errors occur test case T has succeeded
    - Errors are corrected & retested until no errors with test case T are observed.
- Program is Mutated P'
  - Mutant P' is executed for test case T
  - IF no errors are found {
    - test case T is inadequate; further testing is required;
    - new test cases are added & step 3 is repeated until all mutations are discovered; entire process is started again at step 1 with the new test cases
  - } ELSE { all mutations located by tests T
    - T is adequate and no further testing is required}

Error Seeding

- Estimate of the number of original undiscovered errors remaining in a system.
  - Intentionally introduce (seed) errors into the source code.
  - Execute test cases upon source code.
  - Count the number of seeded errors & original errors (unseeded errors) discovered.
  - Estimate the total number of original errors

Error Scattergram Graph



Debug Error Checking

```
assert ( (index>=0) && (index < MAXDIM) );
```

- The assert macro pseudo-function defined in <assert.h>, and <cassert>, (new style header), is used to check a condition, (pre-condition, post-condition, etc.).
- If the condition is false, assert prints an error message containing the line number, the condition tested, and the file name containing the assert, calls the abort function in <stdlib.h> and <cstdlib> to halt program execution.
- If the condition is true execution continues normally.

Debug Error Checking

- Release builds and assertions
  - assert functions need not be removed after testing is complete.
  - Defining the preprocessor symbolic constant NDEBUG will force the preprocessor to ignore all of the assertions.

```
#define  NDEBUG
```

NDEBUG must be defined at the beginning of the program files.

Debug Error Checking

- Considerations
  - Assertions do not allow for programs to recover from errors.
  - It is good programming practice to precede all array accesses with assertions for bounds checking.

Command Line Parameters

`main(int argc, char* argv[])`

- When a program name is typed at the Operating System Line prompt, it is treated as a command. The name of the executable program file and any other strings typed on the line, (usually file names), are considered command line parameters.
- C/C++ compilers provide access to command line parameters.

Command Line Parameters Cont

`argc`

- gives the number of command line parameters (arguments) that the user typed on the command line.
- always equals at least 1 for the command itself

`argv[]`

- an array of pointers to C-style strings that holds the command line arguments.
- argv[0] holds the name of the program (command)
- may hold full pathname on some systems
- may be capitalized on DOS systems

Command Line Parameters Cont.

```
#include <iostream>
#include <cstdlib>     // for EXIT_SUCCESS
using namespace std;

int main (int argc, char* argv[]) {

   cout << "argc = " << argc << endl;

   for (int i =0 ; i < argc; i++) {
     cout << "argv[ " << i << " ] = "
          << argv[i] << endl;
   }

   return EXIT_SUCCESS;
}
```

Module 1 Outline

- What is Software Engineering?
- What is Requirements Engineering
- What is Design?
  - Structure Charts
- What is Testing?
- Intro to Pointers
- Intro to ADT's
- Intro to Classes

Static Variables

- Size is fixed throughout execution
- Size is known at compile time
- Space/memory is allocated at execution

Dynamic Variables

- Created during execution
  - "dynamic allocation"
- No space allocated at compilation time
- Size may vary
  - Structures are created and destroyed during execution.
- Knowledge of structure size not needed
- Memory is not wasted by non-used allocated space.
- Storage is required for addresses.

Example of Pointers

- Assume:
  - Houses represent data
  - Addresses represent the locations of the houses.
- Notice:
  - To get to a house you must have an address.
  - No houses can exist without addresses.
  - An address can exist without a house (vacant lot / **NULL** pointer)

Memory and Addresses

- On modern computers, memory is organized in a manner similar to a one-dimensional array:
  - memory is a sequence of bytes (8 bits)
  - each byte is assigned a numerical address, similar to array indexing
  - addresses are nonnegative integers; valid range is determined by physical system and OS memory management scheme

Memory and Addresses

- OS (should) keep track of which addresses each process (executing program) is allowed to access, and attempts to access addresses that are not allocated to a process should result in intervention by the OS
- OS usually reserves a block of memory starting at address 0 for its own use
- addresses are usually expressed in hexadecimal (base 16), typically indicated by use of a prefix: 0xF4240

Memory Organization

- run-time stack used for statically allocated storage
- heap used for dynamically allocated storage

Pointer Type

- Simple type of variables for storing the memory addresses of other memory locations

Pointer Declaration

Pointer Variables Declarations

- The asterisk '*' character is used for pointer variable declarations:

```
int*   iptr;
float* fptr,
       fptr2;
```

← *recommended form*

← *not a pointer*

- iptr is a pointer to an integer
- fptr is a pointer to a real

common declaration

```
int*  iptr1, iptr2;
```

↑ *not a pointer*

Pointer Declaration

- Given the equivalent declaration:

```
int*   iptr1;
int    iptr2;
```

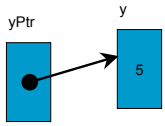
† Declares iptr1 to be a pointer variable, but iptr2 is a simple integer variable.

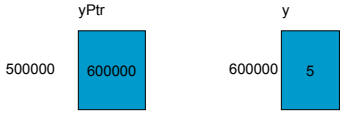
- Typedef declaration:

```
typedef  int* intPtr;
intPtr   iptr1;
```

† Declare all pointer variables in separate declaration statements.

Pointers in Memory





Pointer References

"Address Of" Operator: & (ampersand)

- Unary operator that returns the hardware memory location address of it's operand.

Given:

```
int*   iptr1;
int*   iptr2;
int    numa, numb;

numa = 1;
numb = 2;
```

- Address Assignment:

```
iptr1 = &numa;
iptr2 = &numb;
```

“Contents of” Operator

Dereference / Indirection Operator:

* (asterisk)

- unary ‘pointer’ operator that returns the memory contents at the address contained in the pointer variable.
- Pointer Output:

```
cout << iptr1 << *iptr1 << endl;
cout << iptr2 << *iptr2 << endl;
```

- (Possible) results:

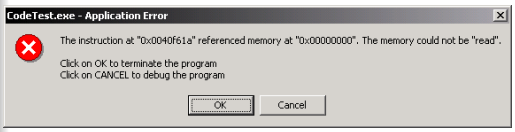
```
0xF4240      1
0x3B9ACA00   2
```

NULL Pointer

- Pointer constant, address 0
- Named constant in the `<cstddef>` include header (`<stddef.h>` old style header).
- Represents the empty pointer
  - points nowhere , unique pointer/address value
- Illegal: NEVER dereference a pointer that equals NULL

  `*NULL`

*NULL



Module 1 Outline

- What is Software Engineering?
- What is Requirements Engineering
- What is Design?
  - Structure Charts
- What is Testing?
- Intro to Pointers
- Intro to ADT’s
- Intro to Classes

Intro to ADT’s

Data Type

- a collection of related data elements plus operations that can be performed upon values of the data type.

Types

- Built-In (Language Defined)
  - Array, Structures (Records), Classes
- Programmer Defined, Abstract Data Types (ADT)
  - Lists, Stacks, Queues

ADT Views

- Application
  - usage in a particular program (variant box view)

- Abstract (Logical)
  - organization viewed by the user (black box view)

- Implementation (Physical)
  - coding methods used to represent the data and the operations (open box view)

Example

- Two-Dimensional Array
  - Application View: maze, surface points
  - Logical View: table, matrix
  - Physical View
    - Stored sequentially
      - (implies logical to physical mapping)

        Index Limits (L1 .. U1, L2 .. U2)

        Length = (U1 - L1 + 1)*(U2 - L2 + 1)

Accessing

- Column Major: all elements in a column are stored in sequence

  FORTRAN and Java - Column Major

- Row Major: all elements in a row are stored in sequence

  "C" , PASCAL - Row Major

Row Major Accessing:

- Location of [i] [j] element (Row Major)

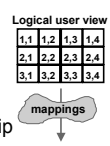
  β + [(U2-L2+1) * (i-L1) + (j-L2)] * size of element

  β = base address of array

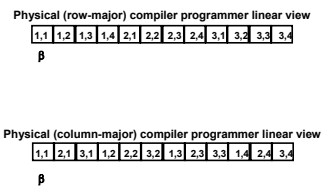
  (U2 - L2 + 1) = Size of Column

  (i - L1) = number of columns to skip

  (j - L2) = number of rows to skip

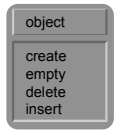


Row vs. Column Major



ADT

- New data type defined by programmer
- Includes:
  - † Set of Data Objects
  - † Set of Abstract Operations



Data Abstraction

- Design of abstract data objects and operations upon those objects.
- Abstraction in programming separates a data type's logical properties from its implementation.

Information Hiding

- Used in the design of functions and new data types.
- Each component should hide as much information as possible from the users of the component (function).
- Implementation details are hidden from the user by providing access through a well-defined communication interface.

Encapsulation

- the bundling of data and actions in such a way that the logical properties of the data and actions are separated from the implementation details [Dale].
- access to an ADT is **restricted** to a specified set of supplied operations
- Implies:
  - User has no "Need to Know"
  - User may **not** directly manipulate data elements
- Advantages
  - Changes to the underlying operators or representations does not affect code in a client of the data type
  - Extends programming languages

Info Hiding vs Encapsulation

Information Hiding
- a question of *program design*
- In many cases: Language Supported (functions, procedures)

Encapsulation
- a question of *language design*
- "...an abstraction is effectively encapsulated only when the language **prohibits** access to information hidden within the abstraction."
- Ada packages
- **C++** classes
  - "**C**" modules

Module 1 Outline

- What is Software Engineering?
- What is Requirements Engineering
- What is Design?
  - Structure Charts
- What is Testing?
- Intro to Pointers
- Intro to ADT's
- Intro to Classes

Rationale for Classes

- The aim of the C++ class construct is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types.
- A type is a concrete representation of a concept.
- For example, the C++ built-in type double with its operations +, -, *, etc., provides a concrete approximation of the mathematical concept of a real number. A class is a user-defined type.

Rationale for Classes

- We design a new type to provide a definition of a concept that has no counterpart among the built-in types.
- A program that provides types that closely match the concepts of the application tends to be easier to understand and to modify than a program that does not.
- A well-chosen set of user-defined types makes a program more concise. In addition, it makes many sorts of code analysis feasible. In particular, it enables the compiler to detect illegal uses of objects that would otherwise remain undetected until the program is thoroughly tested.

The C++ Class Type

- The C++ class type provides a means to encapsulate heterogeneous data elements and the operations that can be performed on them in a single entity.
- A class type is like a struct type in that it may contain data elements, called data members, of any simple or structured type.
- A class type may also contain functions, called function members or methods, that may be invoked to perform operations on the data members.

The C++ Class Type

- The class type also provides mechanisms for controlling access to members, both data and function, via the use of the keywords public, private and protected. (Default access mode is private.)
- A variable of a class type is referred to as an object, or as an instance of the class.

The struct language construct was extended to make it equivalent with a class, except its members are by default public. While structs may be used the same as classes they are rarely employed as such.

A Simple Date Class

```
class DateType {
public:
   void Initialize(int newMonth, int newDay,
                   int newYear);
   int YearIs( )  const;          // returns year
   int MonthIs( ) const;          // returns month
   int DayIs( )   const;          // returns day
private:
   int Year;
   int Month;
   int Day;
};
```

Indicates Fn is a const member and cannot change any data member values.

A Simple Date Class

- The DateType class incorporates three data members, Year, Month, and Day, and four function members.
- Note the class type declaration defines a data type — it does not declare a variable of that type.
- Also note that the class type declaration above includes only prototypes of the function members, not their definitions.

Simple Date Class Continued

- Given the class type declaration, a user may declare variables of that type in the usual way:

```
DateType Today, Tomorrow, AnotherDay;
```

- No default initializations are performed. It is up to the user to assign values to the data members of these variables.

Simple Date Class Continued

- The data members of the DateType class are declared as being private. The effect is that the data members cannot be accessed in the way fields of a struct variable are accessed:

```
Today.Month = 9;
```

will generate a compile-time error. A user of a DateType variable may access only those members which were declared public. So, the user could initialize Today by using the public member function Initialize( ):

```
Today.Initialize(9, 28, 1998);
```

Simple Date Class Continued

- Similarly, a user can only obtain the value of the Year member by using the public member function YearIs( ):

```
int ThisYear = Today.YearIs( );
```

- Note the use of the field selection operator '.'.

Module 1 Outline Completed

- What is Software Engineering?
- What is Requirements Engineering
- What is Design?
  - Structure Charts
- What is Testing?
- Intro to Pointers
- Intro to ADT's
- Intro to Classes