Chapter 1

DESIGNING AND DEBUGGING PROGRAMS

Get on the Fast Track!

SYS-ED/ Computer Education Techniques, Inc.
Objectives

You will learn:

- Debugging process - principles and steps.
- How to recognize, isolate, and identify bugs.
- Determine a fix for the bug.
- Steps to reduce debugging.
- Steps in the debugging process.
- Program design and a well-designed program.
- Designing programs prior to coding.
- Coding guidelines.
- Logical control structures.
- Debugging guidelines.
- Syntax errors.
- Logic errors.
Bug and Debugging - Origin

The terms "bug" and "debugging" have been attributed to Admiral Grace Hopper in the 1940s. The terms are now part of the lexicon specific to information technology.

The term "bug" dates back at least to 1878 and Thomas Edison. Debugging also has been used in aeronautics. In an article in the Journal of the Royal Aeronautical Society circa 1945, the Oxford English Dictionary referred to the "debugging" used in testing airplane engines.
2 Software Tools

Debugging can be a challenging and tedious task. The debugging skill of the programmer is essential to problem resolution and producing code which operates efficiently and as intended. The difficulty of software debugging varies greatly with the programming language used and the availability and quality of debugger tools.

Debuggers are software tools which enable the programmer to monitor the execution of a program, stop and re-start the program, execute in a controlled step-wise fashion, change values in memory and if necessary revert to a prior code state. The term debugger also refers to the person doing the debugging.

High-level programming languages facilitate the debugging process with exception handling which makes it easier to identify sources of unexpected changes of flow of control and processing of data.
3 Debugging Process - Principles and Steps

Although each debugging experience will be unique, there are general principles which can be applied to the debugging process.

The fundamental steps in software debugging are:

1. Recognize that a bug exists.
2. Isolate the source of the bug.
3. Identify the cause of the bug.
4. Determine a fix for the bug.
5. Apply the fix and test it.

3.1 Recognizing Bugs

Detection of bugs can be done proactively or passively. An experienced programmer often will know where errors are most likely to occur. This knowledge of errors typically will be based on the complexity of sections of the program, as well as sources of data corruption.

- Any data obtained from a user needs to be examined and evaluated.
- Standardized testing needs to be applied ensure the integrity the format and content of the data.
- Caution needs to be applied to verify that the format and content of the data is correct.
- Data obtained from transmissions needs to be verified in order to ensure that the entire message and data has been received.
- Complex data must be parsed and processed in order to identify and remove unexpected combinations of values that the receiving program will not be able to handle correctly.

By inserting checks for common and anticipated error symptoms, a program will be able to detect when data has been corrupted or not handled correctly.
Isolation of the source of a bug typically is the most difficult step in debugging. The objective is to identify the portion of the system which is causing the error. Unfortunately, the source of the problem is not always the same as the source of the symptoms.

When an input record is corrupted, an error may not occur until the program is processing a different record or performing some action based on an erroneous instruction or improper data. This can transpire long after the record has been read.

This step often involves iterative testing. The programmer will need to verify that the input is correct, whether it was read correctly, processed correctly, etc. By iteratively testing inputs and outputs, the debugger can identify within a few lines of code where the error is occurring.

Skilled debuggers will be able to anticipate and establish a hypothesis where problems are likely to occur and test the inputs and outputs of the program. This form of debugging is based upon the application of the scientific method. Inexperienced programmers will often step sequentially through the program, looking for a place where the behavior of the program is different from that which is expected. Although a form of the scientific method; the programmer must make a decision on which variables to examine when looking for unusual behavior.

A binary search is another form of isolation process. By testing sections near the middle of the data and processing flow, the programmer can determine whether the error happens in earlier or later sections of the program. When no data problems are detected in the initial binary search, then the error is probably in the bottom portion of the process.

Once the location of the bug has been found, the next step is to determine its cause. This also may involve other sections of the program. When it has been determined that a program fault has resulted from a problem with a field, it will be necessary to identify the bad data and inability of a program to handle the data.

Knowledge of the system will be essential to successfully identifying the source of the bug. A skilled debugger will be able to isolate where a problem originates, familiarity with the system will provide the ability to identify the cause of the error. In certain situations, the bug might be external to the system such as with incorrect input data. In other cases it might be due to a logic error, where correct data has been handled incorrectly.

Unexpected values also can be a source of bugs. This occurs when the design of a program assumes that a given field can have only "n" values, when in fact, it can have additional values, as well as unexpected combinations of values in different fields: field x was only supposed to have a value when field y is something different.

Incorrect reference data can be a source of bugs; a lookup table which contains incorrect values relative to the record can result in corrupted data.

Once the cause of the bug has been determined, the recommended practice is to examine similar sections of the code in order to whether the same mistake has been repeated elsewhere.
3.4 Determine a Fix for the Bug

Once the source of the problem has been identified, it will be necessary to determine how the problem can be fixed. Implementing a bug fix will modify the existing behavior of the system and may produce unexpected results. Fixing an existing bug often will create additional bugs or expose bugs already present in the program, but that never have been exposed in the original bug.

In most situations, these problems are caused by the program executing a previously untested branch of code or untested conditions. In some cases, a fix is straightforward. This is especially true for logic errors where the original design has been implemented incorrectly.

If the problem uncovers a major design error which permeates a large portion of the system, then the fix may be difficult or impossible to resolve and require a total rewrite of the application. In some cases, it might be preferable to implement a "quick fix", followed by a more permanent fix.

This decision will be based upon the severity, visibility, frequency, and side effects of the problem. The nature of the fix and product schedules will have to be evaluated.

3.5 Fix and Test

After the fix has been applied, the system will have to be tested in order to determine whether the fix resolves the problem correctly.

Testing will need to:

1. Confirm whether the fix now handles the original problem correctly?
2. Ensure that the fix does not create any undesirable side effects.

For large systems, the recommended practice is to have regression tests, a series of test runs that exercise the system. After significant changes and bug fixes, these tests can be repeated at any time to verify that the system still executes as expected.
4   Steps to Reduce Debugging

4.1   Mindset

When starting to debug a program, it is important to recognize that:

• It will not be realistic to have a comprehensive understanding of the process.
• Programmers who are convinced that their program will work correctly are less likely to find errors.

When a program behaves the way it is expected to, it is not a recommended practice to bypass a formal debugging process. Proceed with the examination based on the assumption that there is at least one bug remaining and that there is a requirement to locate it. In this way, there will be a far higher likelihood that something will be found wrong with the program.

4.2   Start at the Source

A programmer will become aware of where problems are most likely to arise typically will be during the design phase and as the code is being written. By inserting integrity checks at various places within the program, problems can be detected and reported by the program itself.

In addition to detecting problems, consideration needs to be given on to handle and report each error:

• Set invalid fields to a default value, and continue.
• Discard the record associated with the invalid value, and continue.
• Transfer invalid record into a separate file/table in order that the user can examine and possibly correct the problem.
• Terminate the program.
4.3 User Input

Any data that originated from users, and this includes external systems, need to be examined. All input data needs to be validated through syntactical and semantic integrity checks.

Invalid data are a common source of programming errors. This includes not just data entered in error, but malicious data which results in buffer overflow.

• If data is entered interactively by users, appropriate error messages can be provided which allow the user to correct the invalid field.

• If data has not been entered from an interactive source, then the erroneous records also require a routine for correcting an error.

4.4 Log Files

Programs that write information to log files can provide valuable information for analyzing what was going on before, during, and after problems are encountered.

The number of entries to be searched can be reduced by creating specific log files, with a separate log for each major component of the system and one additional log file strictly for errors.

Each entry needs to be date/time stamped in order that entries from different logs can be correlated.

4.5 Test Suites

A standard set of tests can be run to perform tests and assist in finding errors before the code is placed into production. These test cases need to be automated to reduce the amount of effort required to perform these tests.

As new features are added to the system, additional tests will need to be created and automated to exercise those features.
4.6 Change One Thing at a Time

When making a high number of changes, apply them incrementally. Add one change and then test that change thoroughly before starting on the next change. This will reduce the number of possible sources of new bugs.

If several different changes are applied at the same time, then it is much more difficult to identify the source of the problem.

Minor errors in different areas can interact to produce errors that would not have happened if those changes had been applied one at a time.

4.7 Back Out Changes

If a change has been made to fix a problem, but the program still behaves the same way, back out those changes before proceeding.

The fact that the changes didn't do anything indicates one of several things:

- The problem is not where it originally was thought.
- The area that has been modified either is not being called or is not being called as assumed previously.
- Assuming the section that has been changed was not executed, there may have been new bugs introduced that will not appear until the current bug is fixed.
5 Debugging Essentials

There are six essential elements to the debugging process:

1. Reproducibility
2. Reduction
3. Deduction
4. Experimentation
5. Experience
6. Tenacity

5.1 Reproducibility

Find a way to reliably reproduce the error, this will often provide clues to the problem through deductive reasoning. The bug may happen in precisely one circumstance which can only happen one place in the code.

A bug that appears randomly is essentially unsolvable. There needs to be a guarantee of cause and effect in order to make inferences about changes that have been introduced. A change in the code that fixes the problem may not really fix it, the problem may randomly appear and disappear.

5.2 Reduction

Reduction is the process of reducing a problem in order to determine the smallest input that will cause the bug. The simpler the data or path to the bug, the easier it will be to deduce or track down the problem.

A large dataset can present problems in locating the cause of the trouble. When there is a large data input file associated with the problem, perform a binary search to reduce the problem.

- Divide the file in half, discard the bottom portion of the data.
  - If there still is a problem, then the bottom portion can be ignored.
  - If the problem goes away, then start whittling down the final half, as it must contain the input that causes the problem.
5.3 Deduction

Once a small input reliably causes a problem, then deduction will be important in resolution.

- What is the general path through the program used by the input?
- What components may be the problem or mangle the data in order that a future component fails?
- What is the difference between the input that does not work and some other input that does?
- Attempt to reduce the scope of possible errors by forming and eliminating hypotheses.

5.4 Experimentation

Psychologists study the human mind by testing it in varying situations with different stimuli. Brain scans, response times, and so on are used to support develop and support hypotheses about how human brains work.

Similarly, a programmer must change the conditions in order to test whether a bug disappears.

If the bug is corrected with a change, that change might indicate what the problem is or provide insightful information about what is going on in the system.

Hypotheses are developed based upon logic and deductive reasoning, and then tested and refined with experimentation and observation.

5.5 Experience

Experience is important. In order to become a productive debugger, it will be helpful to have an association with a capable experienced programmer. The alternative is to learn by doing and make mistakes.

Experience helps in the debugging process in two ways:

- Efficiently execute the debugging tools.
- Familiarity with a similar bug will help in addressing the current debugging challenge.
5.6 Tenacity

Computers perform exactly as they are instructed to by software. It will be necessary to have the attitude that giving up is not part of the feasible choice set; effort and time will need to be expended in order to find the problem and correct it. Confidence will increase each time a problem is solved.

Tenacity is a prerequisite attribute for successful debugging. There has never been a bug left unsolved that was not worth fixing.

A by product of solving insidious bugs, will be the knowledge and experience for anticipating errors and coding in a manner less likely to produce mysterious bugs.
6 Program Design

6.1 Preparation

Preparation is important in avoiding hours spent untangling complex code that has been typed directly into the computer system.

- Be clear about what the program is to do.
  - Know the inputs and outputs.
- Write a broad algorithm.
  - Carefully refine it using pseudo-code.

6.2 Procedure or Data Division First

The issue to be evaluated and decision that has to be made is whether to write the PROCEDURE DIVISION first and then go back and write the DATA DIVISION?

This would appear to be a sensible approach; except that in practice there typically will be a large number of variables that will not be declared. A significant degree of processing with the COBOL programming language involves performing calculation on data which relies on what has been declared in the DATA DIVISION.

The recommended practice is to prepare a high percentage of the data definitions on paper first. When writing a program, write as much DATA DIVISION as possible before starting on the PROCEDURE DIVISION. Then proceed through the code; return to the DATA DIVISION in order to update it as soon as possible.

6.3 Commenting

Include comments with the code which explains exactly what each fragment of code is meant to do.

This will help when examining what a program is doing; it will be needed in debugging and maintenance.
6.4 Variable Names

Use variable names that are meaningful and adhere to a standard format.

Example:
Use a prefix before variable names to indicate the general function of the variable.

```
01 PRINT-OUTPUT.
  03 P-NAME PIC X(20).
  03 P-ADDRESS PIC X(50).
  03 P-CUS-CODE PIC 9(6).
  03 P-PAGE-COUNT PIC 999.
```

6.5 Break it Up

Divide the code into smaller procedures, such as paragraphs.

It makes the program easier to read and identify areas where problems are emanating from.

6.6 Stubs

Stubs can be used to monitor what is going on when a program is run. Place "stubs" at important points in the logic.

Stubs are a DISPLAY statement which indicates a certain position in the logic has been executed:

```
MAIN-PARAGRAPH.
  DISPLAY 'IN MAIN PARAGRAPH'
```

6.7 Watching Variables

The placement "IN PARAGRAPH XYZ" of flags which can be extended to display the value of certain important variables:

```
DISPLAY "P-COUNTER = " P-COUNTER
```

When the viewing is complete, remove or comment the flags.
7 A Well-Designed Program

Program design is the development of a program in order that its elements fit together logically and in an integrated way. When programs are systematically planned prior to coding, they will be better designed.

Planning tools such as pseudocode, flowcharts, and hierarchy charts help in the organization and development of program logic.

COBOL programs need to be designed in order to efficiently utilize the logical control structures of sequence, selection, iteration, and case. There are techniques available to make programs easier to code, debug, maintain, and modify.

Programs need to be structured:
- Well-designed, structured programs have a series of logical programming constructs.
- The order in which these instructions are executed is standardized.
- Each set of instructions that performs a specific function is defined in a module or program segment; this is referred to as cohesion.

A module is referred to as a routine or, in COBOL, a paragraph:
- Each module is executed in its entirety from specific places in a program.
- In COBOL, modules are executed using a PERFORM statement.

Programs should use a top-down approach:
- All modules are independent of all other modules.
- There is only one entry point and one exit point.
- The modules are coded in a hierarchical order, with main modules written first followed by secondary modules that include the detailed code.
- This top-down approach provides stepwise refinement.
8  Designing Programs Prior to Coding

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>What are the required inputs and outputs?</td>
<td>system flowchart</td>
</tr>
<tr>
<td>What should the input and output look like?</td>
<td>record layouts</td>
</tr>
<tr>
<td></td>
<td>printer spacing charts</td>
</tr>
<tr>
<td>What steps need to be perform to accomplish the specifications?</td>
<td>hierarchy chart</td>
</tr>
<tr>
<td>How should the identified steps be implemented?</td>
<td>psuedo code</td>
</tr>
<tr>
<td></td>
<td>program flow charts</td>
</tr>
<tr>
<td></td>
<td>UML charts</td>
</tr>
</tbody>
</table>

In addition to learning syntax, programmers must learn how to design a program in order that it functions effectively as an integrated whole. The techniques for developing well-designed programs are applicable to all languages.

With knowledge of program design, it only will be necessary to learn the syntax rules of a specific language to implement these design elements.
9 Naming Modules or Paragraphs

When naming modules or paragraphs:

- Use a standard method for naming paragraphs in all programs.
- Choose meaningful names.
- Prefix with a numbering scheme.
- Preferred Numbering scheme.
- Paragraph's PERFORMed from more than 1 place should use the 900- paragraphs.
10 Coding Guidelines

Guidelines for coding programs include:

- Code each clause on a separate line.
  - Each module begins with a Paragraph Name.
  - Coding one clause per line makes programs easier to read and debug.
  - Words and clauses can be separated with any number of blank spaces.
  - Having only a single clause on each line helps to isolate errors.

- Indent Clauses within a statement.
  - Indentation makes programs easier to read.
  - By convention, there an indentation of four spaces on each line.
  - Indentation is used to clarify the logic.
  - Indentation does not affect the program logic at all.

Modularizing Programs

Programs are written top-down with main units or modules planned and coded first, followed by more detailed ones.

Structure or hierarchy charts illustrate the relationships among these modules as well as the Order in which they will be executed.

Statements that together achieve a given task should be designed as a module.
Structured programs use logical control structures to specify the order in which instructions are executed.

Logical control structures provide:

<table>
<thead>
<tr>
<th></th>
<th>sequence</th>
<th>selection</th>
<th>iteration</th>
<th>case structure</th>
</tr>
</thead>
</table>

These structures are the same for all languages.

The four logical control structures implement sequence:

- A sequence is instructions executed in the order they appear.
- Pseudocode represents a sequence.
- The ellipsis means that each statement has other components.
- Sequence is used to depict the logic when data is to be processed in a step-by-step order.
12 Scope Terminators

COBOL programmers should use scope terminators with the READ, inline PERFORM and IF statements.

- When scope terminators are coded, periods are not used to end statements except for the last statement in a paragraph.
- Scope terminators ensure that all clauses within a statement will be associated with the appropriate instruction, thereby minimizing logic errors.
- Programs must be fully tested to ensure that there are no errors.
13 Syntax Errors

After a program has been planned and coded, it is keyed into a computer. Then it is ready to be compiled or translated into machine language. During this translation and compilation process, the compiler will list any violations in programming rules that may have occurred.

These rule violations are syntax errors; they must be corrected before the program can be executed.

When diagnostics appear in a source listing either at the end or immediately after the line in question, they typically have the following format:

<table>
<thead>
<tr>
<th>Line No.</th>
<th>Error Code</th>
<th>Error Message</th>
</tr>
</thead>
</table>

Common syntax errors include:

- Misspelling data-names.
- Using the same data-name more than once.
- All file and record names must be unique.
- Field names must be unique, unless data-name is qualified when used in PROCEDURE DIVISION.
- Program-name and paragraph names must be unique.
- Using a reserved word for user-defined name.
- Use a reserved word only for its designated purpose.
- Misspelling reserved words.
- Using a nonnumeric field in arithmetic statement.
- Omitting scope terminators.
- Using letter "oh" instead of zero.
14 Logic Errors

Except for warnings, syntax errors will be detected by the compiler; they should all be corrected before a program is run. However, even after a program has been compiled, it is not fully debugged. The program must be executed with test data to ensure that there are no logic errors.

Some logic errors result in a program interrupt. These are run-time errors and must be corrected before execution can continue. Other logic errors result in erroneous output. These will be detected only if the test data is complete and the program is carefully checked by the programmer.

Common run-time logic errors include:
- Performing arithmetic operation with a field containing nonnumeric characters.
- Incorrect name, path or device-name for the input file in an ASSIGN clause.
- There is no way provided to stop a PERFORM UNTIL loop.

Common logic errors in output include:
- PERFORM UNTIL loop executed one too few or one too many times.
- Scope terminator is not in the correct location.

Detecting logic errors in output including:
- Preparing complete test data.
- Test data values.
- Detecting logic errors in output.
- Performing a structured walkthrough.
- Comparing computer-produced results to expected results.

Finding logic errors:
- Using the DISPLAY verb at key points in the program to determine program status and data values.
- Using the interactive debugging tools.