บทที่ 4
การตรวจสอบโปรแกรม
(Program Debugging)

โปรแกรมที่เขียนเสร็จแล้วนั้น จำเป็นจะต้องมีการตรวจสอบ ทั้งนี้เพราะอาจจะยังมีที่ผิดอยู่ ข้อผิดพลาดที่ปรากฏในโปรแกรมนั้นจะแบ่งได้เป็น 2 ลักษณะ คือ

1. Syntax Error ความผิดชนิดนี้นั้น เนื่องจากการผิดไวยากรณ์ของภาษานั้น หรืออาจจะไปใช้ผิดจากรูปแบบที่กำหนด (format) ของภาษา หรือการนำเอา reserved มาใช้ตั้งชื่อ variable name เป็นต้น ปกติความผิดพลาดชนิดนี้เครื่องจะมี OS ช่วย support ในการค้นหาความผิดพลาดชนิดนี้อยู่แล้ว ซึ่งจะช่วยให้โปรแกรมเมอร์สามารถค้นหาความผิดพลาดในลักษณะนี้ได้โดยเร็ว

2. Logic Error ความผิดพลาดชนิดนี้เป็นความผิดพลาดที่ตรวจหาค่อนข้างจะยาก ทั้งนี้เพราะเครื่องไม่สามารถบอกได้ จึงเป็นหน้าที่ของโปรแกรมเมอร์เองที่จะต้องตรวจสอบเองโดยละเอียด โดยอาจจะดูจากผังโปรแกรมประกอบการค้นหาก็ได้ ตัวอย่างของ Logic Error เช่น การใช้สูตรในการคำนวณผิดพลาด เช่น สูตรการจ่ายเงินเดือนสุทธิ = (เงินรายได้ – ภาษี + เงินสวัสดิการ) แต่เรากลับไปเขียนสูตรว่า เงินเดือนสุทธิ = (เงินรายได้ + ภาษี + เงินสวัสดิการ) ซึ่งการใช้สูตรผิดในลักษณะนี้เครื่องไม่สามารถจะตรวจจับได้

ความผิดพลาดทั้ง 2 ข้อ ที่กล่าวมานี้เราเรียกว่า bugs และกรรมวิธีในการแก้ไขข้อผิดพลาดนี้เรียกว่า debugging ดังนั้นถ้าเรารู้ว่าโปรแกรมที่เขียนมานั้นยังมีข้อผิดพลาดอยู่เราก็จะต้องดำเนินการ debugging กับโปรแกรมนั้น แต่ถ้าหากว่าโปรแกรมนั้นไม่มี bugs อีกแล้ว เราจะเรียกกรรมวิธีที่จะดำเนินการกับโปรแกรมก่อนที่จะนำไปปฏิบัติงานจริงว่าเป็นการ testing

โดยปกติแล้วเรามักจะพบเห็นว่า โปรแกรมเมอร์นั้นจะถูกฝึกฝนมาในการฝึกหัดเขียนโปรแกรม แต่ไม่ได้ฝึกมาในด้านของการ debugging ทั้ง ๆ ที่ความเป็นจริงแล้วการ debugging โปรแกรมนั้นมักจะใช้เวลามากกว่าการเขียนโปรแกรมและทั้งยังเป็นขั้นตอนที่สำคัญมากกว่า

การเขียนโปรแกรมเสียอีก เป็นที่คาดประมาณกันว่าเวลาที่ใช้ในการ debugging นั้นจะใช้เวลาประมาณ 50—90% ของเวลาทั้งหมดในการจะสร้างโปรแกรม ๆ หนึ่งให้ใช้ได้ เราจะเห็นได้ว่าการเขียนโปรแกรมนั้นจะต้องมีอยู่ 2 ขั้นตอนด้วยกันเหมือนกับการเขียนรายงาน คือ ต้องเขียนฉบับร่างขึ้นมาเสียก่อนในที่นี้เปรียบเทียบได้กับการเขียนโปรแกรมในครั้งแรกนั้นเอง และภายหลังเมื่อเรา debugging โปรแกรมนั้นเสร็จแล้ว เราก็จะได้โปรแกรมฉบับที่ 2 (final draft)

ข้อเท็จจริงอย่างหนึ่งซึ่งเป็นที่ยอมรับกันก็คือ การ debugging นั้นเป็นศิลป (art) ซึ่งมักจะสอนกันไม่ค่อยได้ ปัจจัยของการ debugging นั้น จะขึ้นกับสภาพแวดล้อมต่อไปนี้คือ อุปกรณ์ที่ใช้, ภาษาที่เขียนโปรแกรม, ระบบควบคุม (the operating system) ปัญหาที่ดำเนินการและลักษณะโปรแกรมที่เขียนขึ้น ทั้งนี้เพราะการใช้ภาษา, compiler ตลอดจนอุปกรณ์ที่ใช้นั้นแตกต่างกันก็มักจะส่งผลให้ bugs ที่เกิดขึ้นมีรูปแบบแตกต่างกันด้วย ที่เห็นได้ชัดเจนก็คือ Syntax Error ซึ่งจะขึ้นอยู่กับภาษาแต่ละภาษาที่ใช้งาน

เนื้อหาที่จะกล่าวต่อไปนี้จะเน้นเกี่ยวกับเรื่องของ source language debugging ทั้งนี้เพราะจะได้หลีกเลี่ยงการกล่าวถึงเรื่องของความรู้เกี่ยวกับภาษาเครื่อง ซึ่งผู้อ่านบางคนอาจจะมีความรู้สึกว่ายุ่งยาก การที่จะใช้กรรมวิธีของ machine language debugging นั้นไม่มีประโยชน์ ทั้งนี้เพราะกรรมวิธีนี้จะพึ่งพิงอยู่กับอุปกรณ์ที่ใช้อยู่ตลอดเวลา (machine dependent) นั่นหมายความถ้าเราใช้ machine language debugging แล้ว เมื่อเราเปลี่ยนไปใช้คอมพิวเตอร์เครื่องอื่น หรือเกิดมีการเปลี่ยนแปลงระบบคอมพิวเตอร์ใหม่ก็จำเป็นจะต้องมีการเปลี่ยนแปลงโปรแกรมนั้นใหม่ ดังนั้น กรรมวิธี machine language debugging จึงไม่ค่อยนิยมเท่าใดนัก แต่ก็มีข้อแม้ว่าเราจะใช้ machine language debugging และ memory dump debugging ในสถานการณ์ที่ทุกกรรมวิธีในการ debugging นอกเหนือจาก 2 วิธีนี้นั้นใช้ไม่สำเร็จ เราอาจสรุปข้อดีของกรรมวิธี source language debugging ก็คือ

1. เราไม่จำเป็นจะต้องเรียนรู้ภาษาเครื่อง
2. เราสามารถสั่งให้พิมพ์ output ในรูปแบบที่อ่านได้รู้เรื่อง โดยมี lable ที่เหมาะสมกำกับอยู่
3. เทคนิคทั้งหลายที่ใช้ได้ใน source program สามารถนำมาเป็นเทคนิคในการ debugging ได้

ข้อแตกต่างระหว่าง debugging และ testing

มีโปรแกรมเมอร์มากมายที่เข้าใจสับสนระหว่างคำว่า debugging และคำว่า testing เราอาจจะจำแนกคำ 2 คำนี้ได้ง่าย ๆ ดังนี้ว่า ถ้าหากว่าโปรแกรมที่เขียนขึ้นนั้นยังปฏิบัติการไม่ได้หรือปฏิบัติการแล้วยังได้ผลที่ผิดพลาดอยู่ โปรแกรมนั้นจำเป็นที่จะต้องนำไป debugging แต่ถ้าเมื่อไรที่โปรแกรมนั้นดูแล้วพบว่า ทำงานได้ถูกต้องเราก็จำเป็นจะต้องนำโปรแกรมนั้นไปทำการ testing ก่อนที่จะไปใช้ปฏิบัติงานจริงต่อไป และก็เป็นความจริงว่ามีอยู่บ่อย ๆ ครั้งที่พบว่า โปรแกรมที่นำไป testing นั้นเราต้องนำกลับมา debugging ใหม่อีก จุดประสงค์ของ testing ก็เพื่อที่จะตรวจดูว่ายังมี error หลงเหลืออีกหรือไม่ในโปรแกรมนั้นในขณะที่ debugging คือวิธีที่จะหาเหตุที่ทำให้เกิด error ดังนั้นในทั้ง 2 ขั้นตอนนี้จึงยังมีการ overlap กันอยู่

โดยปกติแล้วโปรแกรมเมอร์ควรจะไล่ตรวจสอบโปรแกรมของตัวเองพร้อมกับกำหนดชุดของข้อมูลที่จะให้คอมพิวเตอร์ทดสอบปฏิบัติงานเสียก่อน วิธีการนี้เรียกว่า manual debugging ซึ่งจะมีข้อดีในแง่ที่ว่าประหยัดเวลาเครื่อง แต่ในบางครั้งถ้าหากโปรแกรมเมอร์ขี้เกียจเสียเวลาไล่โปรแกรมเอง ก็อาจจะให้เครื่องช่วยตรวจ error ในส่วนที่ผิดไวยกรณ์ของภาษาก่อนก็ได้

สาเหตุที่เกิด error ในโปรแกรม

1. Error in Problem Definition

บ่อยครั้งที่เราพบว่าผลจากการทำงานของโปรแกรมนั้นออกมาไม่ถูกต้องตามความต้องการของผู้ใช้ สาเหตุที่เป็นเช่นนี้ก็เพราะว่าโปรแกรมเมอร์เข้าใจ หรือตีความต้องการของผู้ใช้ไม่ถูกต้องนั่นเอง เราอาจจะคิดว่าก็คงไม่มีการพูดคุยกันระหว่างโปรแกรมเมอร์กับ user นั่นเอง แต่อันนี้ก็ยังไม่ถูกต้อง 100% เสมอไป ทั้งนี้เพราะบางครั้งที่มีการพูดคุยระหว่างโปรแกรมเมอร์กับ user แล้วแต่งานที่ปรากฏออกมาก็ยังไม่ถูกต้องตามความประสงค์ของผู้ใช้ สาเหตุเช่นนี้เราอาจจะแก้ไขได้โดยการเขียนร่างความต้องการของผู้ใช้ออกมาเป็นผังภาพ (รายงานที่ต้องการ) ในขณะที่พูดคุยกันระหว่างโปรแกรมเมอร์และ user เพราะการที่ได้เขียนผลที่ต้องการออกมาเป็นไดอะแกรมคร่าว ๆ จะเป็นการช่วยตรวจสอบและยืนยันความต้องการของ user ต่อโปรแกรมเมอร์ได้ดีกว่าการจะพูดซึ่งเป็นนามธรรม

2. Incorrect Algorithm

เมื่อผ่านปัญหาที่ 1 มาแล้ว ขั้นตอนต่อไปที่โปรแกรมเมอร์จะต้องดำเนินการก็คือ ค้นหาขั้นตอนการดำเนินการ หรือวิธีการที่เหมาะสมเพื่อนำมาใช้แก้ปัญหานั้น เราจะพบว่าในขั้นตอน

ของการเลือกอัลกอลิทึมนั้น โปรแกรมเมอร์อาจจะเลือกอัลกอลิทึมซึ่งไม่ถูกต้องมาใช้งานก็ได้ หรือบางทีอัลกอลิทึมนั้นอาจจะถูกต้องแต่เป็นวิธีที่ไม่มีประสิทธิภาพก็ได้ ตัวอย่างเช่น เลือกกรรมวิธีการแก้ปัญหาในระบบสมการทางคณิตศาสตร์ ซึ่งเป็นวิธีที่ช้ามากในการหาคำตอบปัญหาที่เกิดขึ้นขณะนี้ก็คือ แล้วโปรแกรมเมอร์จะทราบได้อย่างไรว่าในบรรดาอัลกอลิทึมแต่ละวิธีนั้นวิธีไหนที่ดีที่สุดที่ควรจะใช้ คำตอบก็มีอยู่ 2 ทาง คือ หนึ่งทดลองเองทุกวิธีก็จะทราบว่าอะไรคือวิธีที่เราควรจะเลือกมาใช้ ซึ่งวิธีที่หนึ่งนี้คงจะไม่มีใครปฏิบัติตามแน่ ยกเว้นแต่ผู้ที่ทำงานในเรื่องการเสาะหาคำตอบเปรียบเทียบในงานนั้น ๆ อยู่แล้ว ดังนั้น โปรแกรมเมอร์ทั่ว ๆ ไปก็คงจะต้องไปพึ่งพิงแหล่งที่สองคือ ศึกษากรรมวิธีแต่ละวิธีจากเอกสารงานวิจัย หรือตำราต่าง ๆ ที่ว่าด้วยงานนั้นเพื่อดูว่า อัลกอลิทึมแต่ละวิธีมีจุดบกพร่องจุดเด่น อย่างไรบ้าง และวิธีใดที่จะเหมาะสมกับสภาพปัญหาของเรามากที่สุด

3. Errors in Analysis

Errors in Analysis นั้นจะประกอบด้วย ความผิดพลาดจากการมองข้ามปัญหาความเป็นไปได้บางอย่าง หรือการใช้วิธีการแก้ไขปัญหาที่ไม่ถูกต้อง ปัญหาของการมองข้ามบางสิ่งบางอย่างที่อาจเกิดขึ้นได้ในการปฏิบัติงานที่แท้จริง เช่น การไม่พิจารณาข้อมูลซึ่งอาจจะติดลบ, เป็นศูนย์ หรือเลขที่มีค่ามาก ๆ ได้

ความสำคัญของการไม่มองข้อมูลให้ครบถ้วนนั้น มักจะก่อให้เกิดปัญหาของ logic error

ปัญหาที่เกิดจากการละเลยในการปฏิบัติงานบางจุดนั้น จะประกอบด้วย

3.1 การไม่กำหนดข้อมูลให้ตัวแปรเริ่มต้น

3.2 การกำหนดจุดหยุดการทำงานใน loop ไม่ถูกต้อง

3.3 การกำหนดดัชนีควบคุม loop ไม่ถูกต้อง

3.4 การลืมกำหนดค่าเริ่มต้นที่ควบคุม loop

3.5 มีการเปลี่ยนแปลงทิศทางที่จะไปภายหลัง เมื่อผ่านขั้นตอนการตัดสินใจมาแล้ว

กรรมวิธีในการ debugging สิ่งที่จะเกิดดังกล่าวมานี้ก็คือ การออกแบบผังโปรแกรมอย่างรอบคอบรัดกุม ตรวจสอบให้ถี่ถ้วนในรายละเอียดของแต่ละขั้นตอนก็จะต้องแจงให้ชัดเจน ผังโปรแกรมจะเป็นอุปกรณ์ที่มีประโยชน์มากต่อการ coding และต่อการ debugging

4. Programming Errors

ภายหลังจากการดำเนินงานในขั้นตอนที่ 1–3 แล้วยังพบว่า มี error ปรากฏอยู่ในโปรแกรมได้ ทั้งนี้เนื่องจากสาเหตุต่อไปนี้คือ

4.1 Lack of knowledge ความหมายก็คือ โปรแกรมเมอร์อาจจะยังขาดความรู้หรือไม่สันทัดกับภาษาที่เขียน ดังนั้นจึงอาจจะทำให้เกิดความผิดพลาด หรือบางครั้งถ้าเกิดระบบเครื่องเปลี่ยนไป คำสั่งบางอันที่เคยใช้อาจจะปฏิบัติการแตกต่างไปจากที่เคยใช้ก็ได้

4.2 An error in programming the algorithm ปัญหาที่เกิดขึ้นในกรณีนี้ก็คือ การเขียนคำสั่งจาก algorithm ไม่ถูกต้อง ซึ่งมักจะเกิดในกรณีของการใช้สูตรคณิตศาสตร์ที่ค่อนข้างจะซับซ้อน ปัญหานี้ก็คือ logic errors นั่นเอง

4.3 Syntax errors คือการใช้ไวยากรณ์ของภาษาผิดจากข้อกำหนด

4.4 Syntactically correct statements may cause execution errors ตัวอย่างของปัญหานี้เช่นการนำเลขศูนย์ไปหารเลขอื่น หรือการถอดรูทของเลขติดลบ เป็นต้น

4.5 Data error ตัวอย่างเช่น นำข้อมูลสติงค์ไปปฏิบัติงานในสูตรคณิตศาสตร์

ในบรรดา error ทั้ง 5 ประเภทนี้ยกเว้น error ประเภทที่ 3 คือ Syntax error นั้น จะเป็น error ประเภททั่วไปที่พบได้ในการ testing ซึ่งจะทำให้ต้องกลับมา debugging ใหม่

ตารางต่อไปนี้จะสรุปดัง error ต่าง ๆ ที่กล่าวมาแล้วพร้อมทั้งวิธีการแก้ปัญหา

Common Programming Errors

1. Error in problem definition.	Correctly solving the wrong problem.
2. Incorrect algorithm.	Selecting an algorithm that solves the problem incorrectly or badly.
3. Errors in analysis.	Incorrect programming of the algorithm.
4. Semantic error.	Failure to understand how a command works.
5. Syntax error.	Failure to follow the rules of the programming language.
6. Execution error.	Failure to predict the possible ranges in calculations (i.e., division by zero, tec.)
7. Data error.	Failure to anticipate the ranges of data.
8. Documentation error.	User documentation does not match the program.

หมายเหตุ ในบรรดา error ประเภทที่ 4.1—4.5 นั้น บางคนยังแยกออกเป็น error อีกประเภทหนึ่งออกไปต่างหากโดยเรียกว่า **glich** โดยที่ glitch นั้นอาจจะนับว่าเป็นทั้ง programming

error และ error in documentation ด้วย ก็ได้ error ประเภทนี้จะหมายถึง จุดอ่อนหรือจุดโหว่ของโปรแกรมนั้น โดยปกติเราจะใช้คำนี้ในความหมายที่แสดงถึงโปรแกรมที่มีลักษณะ **awkward, clumsy,** or **does not apply** ดังนั้นถึงแม้ว่าโปรแกรมประเภทนี้จะยังมี glitch อยู่ แต่โปรแกรมนั้นก็สามารถที่จะทำงานได้ตรงตามคุณสมบัติที่ตั้งไว้ แต่อาจจะไม่ตรงตามคุณสมบัติดั้งเดิมที่ตั้งไว้โดยสมบูรณ์

5. **Physical Errors**

Physical Errors ซึ่งก่อให้เกิด program bug นั้นจะแยกรายละเอียดได้เป็นประเภทต่าง ๆ ดังนี้คือ

5.1 Missing program cards or lines

5.2 Interchanging of program cards or lines

5.3 Additional program cards or lines (ie failure to remove corrected lines)

5.4 Missing Data

5.5 Data out of order

5.6 Incorrect data format

5.7 Missing job control (monitor) statements

5.8 Refering to the wrong program listing

การดูแลกับ card decks (กรณีใช้บัตร) จะช่วยลด error ประเภทนี้ ส่วนในเรื่องของคำสั่งถ้าหากเกิดกรณีของ missing หรือ out-of-sequence นั้น จะไม่สามารถตรวจสอบได้โดย syntax error โดยเฉพาะถ้าเป็นโปรแกรมยาว ๆ การจะตรวจดูว่าคำสั่งใดหายไปจะเป็นเรื่องที่ค่อนข้างยากลำบาก ดังนั้นถ้าเป็นไปได้ในภาษานั้น ก็ควรจะใส่หมายเลขประจำคำสั่งเรียงลำดับกันเพื่อจะได้ง่ายแก่การตรวจสอบ

ในเรื่องของข้อมูลนั้น จำเป็นจะต้องมีการนำ edit-checked และ echo-printed เพื่อป้องกันข้อมูลผิดพลาด ความหมายของ edit-checked ก็คือการตรวจสอบข้อมูลก่อนเข้าเครื่อง ซึ่งมีวิธีการซับซ้อนหลายประการ ถ้าหากนักศึกษาสนใจให้ไปอ่านได้จากหนังสือเรื่องการวิจัยเบื้องต้น (ST 436) ในบทที่ว่าด้วยการบรรณาธิการข้อมูล ส่วนการทำ echo-printed ก็คือการพิมพ์ข้อมูลของแต่ละรายการในแต่ละระเบียบข้อมูลออกมาทุกระเบียบข้อมูลของแฟ้มข้อมูลนั้น เพื่อตรวจสอบด้วยสายตามนุษย์

โดยปกติถ้าเป็นการเจาะโปรแกรมลงในบัตรอย่างวิธีที่ใช้กันในสมัยก่อนนั้น เรามักจะมีการทำเครื่องหมายใน program decks ดังนี้คือ ถ้าหากโปรแกรมนั้นมี subroutine อยู่ด้วยก็ให้ทำเครื่องหมายแยกความแตกต่างในลักษณะนี้คือ กำหนดเครื่องหมายของ subroutine ดังนี้

1. Mark fist card on the face with FC
2. Mark last card on the back LC
3. Mark program name on the tops of the deck
4. Mark diagonal or cross strips on the top of the deck

การทำเครื่องหมายเช่นนี้จะช่วยเรารู้ตำแหน่งของแต่ละ subroutine ใน source deck เพื่อความสะดวกในการเรียงสลับใหม่หรือเพื่อในการ debugging

ดังที่ได้กล่าวแล้วในตอนแรก ๆ ว่า เราอาจจะสรุป error ได้เป็น 2 ประเภทใหญ่คือ syntax errors กับ logical errors นั้น syntax error จะถูกตรวจสอบโดย compiler มีอยู่มากมายหลายประเภทดังตัวอย่างจะนำมาแสดง คือ

ประเภทที่พบที่ผิดในคำสั่งนั้น เช่น

1. Required punctuation missing
2. Unmatched parenthesis
3. Missing parenthesis
4. Incorrectly formed statements
5. Incorrect variable names
6. Misspelling of reserved words

syntax error ประเภทที่เกิดจากผลกระทบ (interaction) ของสองคำสั่ง หรือมากกว่าขึ้นไป ดังตัวอย่างเช่น

1. Conflicting instructions
2. Nontermination of loops
3. Duplicate or missing labels
4. Not declaring arrays
5. Illegal transfer

นอกจากนี้ compiler ยังมีความสามารถในการหา syntax errors จากลักษณะของ errors ต่อไปนี้ได้ด้วย คือ

1. Undeclared or incorrectly declared variables
2. Typing errors
3. Use of illegal characters

มีข้อที่น่าสังเกตว่า syntax error บางประเภทที่เกิดขึ้นนั้น โดยตัวของมันเอง (ในคำสั่งนั้น) จะไม่มี syntax error แต่ที่เกิด error ก็เพราะผลกระทบจากคำสั่งอื่น ดังนั้นเวลาดูว่าคำสั่งใดมี syntax error แล้วจะต้องพิจารณาอย่างรอบคอบเสียก่อน มิฉะนั้นแล้วเราอาจจะแก้ไขจากคำสั่งที่ถูกไปเป็นคำสั่งที่ผิดได้ ทางที่ดีขอให้อ่านข้อความซึ่งเป็น code ของ syntax error นั้นว่ามีความหมายอย่างไร แล้วพิจารณาดูเสียก่อนที่จะมีการแก้ไข error ต่อไป

ในกรณีที่เรามี debugging compiler ที่มีความสามารถแล้ว เราจะสามารถลดเวลาที่เสียไปในการ debugging เฉพาะในส่วนของ syntax error ได้สูงถึง 50 กว่าเปอร์เซ็นต์ แต่การที่จะทำงานในส่วนนี้ได้ดีเท่าใดนั้นจะขึ้นอยู่กับความสามารถของ compiler เป็นสำคัญ ตัวอย่างของ compiler ที่เก่ง ๆ ที่ติดตั้งในที่ต่าง ๆ เช่น ที่ University of Waterloo จะมี debugging compiler ของภาษาโคบอลถึง 2 ประเภท คือ COBOL และ WATBOL สำหรับภาษาฟอร์แทรนก็มี FORTRAN และ WATFIV ในขณะที่ Cornell University มี PL/1 compiler ซึ่งเรียกว่า PL/C ที่ Standford University มี ALGOL W

ถึงแม้ว่า compiler จะมีความสามารถมากเพียงใดก็ตาม แต่ยังมี errors บางประเภทที่ compiler ตรวจสอบไม่ได้ดังตัวอย่างเช่น

1. Omission of part of the program
2. Branching the wrong‘way on a decision statement
3. Using wrong format for reaching data
4. Incorrect values in loops, such as the mitial value, increment, or terminal value
5. Arrays to small or incorrect array subscripting
6. Failure to consider all possibilities that may occur in the data or in calculations

ตัวอย่างเช่น ในโปรแกรมหนึ่งกำหนดอะเรย์ A ไว้มีขนาดเท่ากับ 10 และภายในโปรแกรมมีส่วนหนึ่งของคำสั่งมีลักษณะดังนี้คือ

```
I  =  4*K
:
:
A(I)  =  ...
```

จะเห็นได้ว่าโปรแกรมส่วนนี้จะปฏิบัติงานได้ตามเงื่อนไขของการกำหนดค่าของอะเรย์ คือ เมื่อ K = 1 หรือ 2 เท่านั้น ถ้าเมื่อใดที่ K มีค่าเท่ากับ 3 หรือสูงกว่าขึ้นไป โปรแกรมนี้จะเกิด error ขึ้นทันที ลักษณะดังตัวอย่างนี้นั้น compiler ไม่สามารถที่จะตรวจพบได้ ทั้งนี้เพราะ error ประเภทนี้จะเกิดขึ้นก็ต่อเมื่อมีการประมวลผลและพบค่า K มีค่ามากกว่า หรือเท่ากับ 3 เป็นต้นไป นอกจากนั้นค่า K จะต้องไม่เป็นเลขลบ หรือทศนิยม อีกด้วย

errors ทั้งหลายที่ตรวจสอบพบนั้น อาจจะพบในช่วงระยะเวลาต่าง ๆ กัน เช่น อาจจะพบในช่วง compilation หรืออาจจะพบในช่วงของ execution ก็ได้

ประเภทของการ debugging

ภายหลังเมื่อเราแก้ไข syntax error ที่พบเสร็จเรียบร้อยแล้ว ก็ลองรันโปรแกรมนั้นกับ simple test data ที่กำหนดขึ้น ถ้าหากว่า ผลที่ได้จากการรัน test data ออกมาแล้วถูกต้องกับคำตอบที่เราตั้งไว้ ก็แปลว่าเรานำโปรแกรมนั้นไป testing ได้ แต่ถ้าหากว่าผลจากการปฏิบัติกับ test data ยังมีข้อผิดพลาดอยู่ หรือส่งรันแล้วไม่มีผลใด ๆ ปรากฏออกมา ก็แปลว่าปัญหาที่เราประสบอยู่จะต้องอยู่ในประเด็นใดประเด็นหนึ่งดังต่อไปนี้ คือ

1. The program did not compile, but there are no syntax errors
2. The program compiles, executes, but produces no output.
3. The program compiles, executes, but terminates prematurely.
4, The program compiles, executes, but produces incorrect output.
5. The program does not stop running (or infinite loop)

กรณีที่ 1 Compilation Not Completed

เหตุการณ์นี้ค่อนข้างจะเกิดยาก แต่ถ้าเกิดขึ้นก็หมายความว่าเกิด catastrophic error ในโปรแกรมนั้น ในกรณีเช่นนี้จะเกิด system error message ขึ้นมา เพื่อชี้ตำแหน่งของ error การเกิด system error ในลักษณะนี้นั้นเรียกว่า abend (ย่อมาจาก abnormal end) เมื่อเกิด abend ขึ้น เราจะต้องดู system ที่ใช้ ซึ่งอาจจะเป็นเรื่องยากสำหรับผู้ที่ไม่รู้จัก system นั้น ๆ ซึ่งเราอาจจะปรึกษาจากผู้รู้ หรือศึกษาจากคู่มือประจำเครื่องก็ได้ หรือถ้าลองแก้ไขแล้วก็ยังไม่สำเร็จ ก็ให้ใช้วิธีการ

แบ่งโปรแกรมเป็นส่วน ๆ แล้วให้เครื่อง compile เป็นลำดับเพิ่มขึ้นทีละส่วนจนกระทั่งพบว่าส่วนที่เพิ่มเข้าไปนั้นทำให้ไม่มีการ compile ก็จะทราบได้ว่าโปรแกรมส่วนที่เพิ่งเพิ่มเข้าไปเป็นส่วนที่ทำให้เกิด system error

กรณีที่ 2 Execution but No output

เมื่อเราพบว่าโปรแกรมที่ส่งเข้าไปรันนั้นผ่านการ compile แล้วแต่ไม่ปรากฏผลลัพธ์ใด ๆ จากการประมวลผลนั่น หมายความว่าอาจจะเกิดจาก logic error หรือ system error ก็ได้ ตัวอย่างของ logic error ก็เช่นภายหลังการประมวลผลแล้วก็กระโดดไปยังคำสั่งหยุดการทำงานโดยที่ข้ามคำสั่งในการแสดงผล output ออกมา ซึ่งลักษณะของ error ชนิดนี้นั้นจะตรวจจับตำแหน่งที่ผิดพลาดได้โดยเทคนิคที่เรียกว่า locating errors ซึ่งจะกล่าวถึงในตอนต่อไป

การที่ระบบเครื่องหยุดการทำงานนั้นอาจจะเป็นผลสืบเนื่องมาจากองค์ประกอบใดองค์ประกอบหนึ่งต่อไปนี้คือ computer hardware, operating system หรือไม่ก็เป็นโปรแกรมของเราที่ผ่านการ compile แล้ว กรณีที่โปรแกรมถูกขัดจังหวะการทำงานโดยสาเหตุของ system error นั้น ค่อนข้างจะเป็นเรื่องที่ยุ่งยากในการค้นหาที่ผิดพลาด สำหรับการที่โปรแกรมดำเนินไปได้บางส่วนแล้วถูกขัดจังหวะให้หยุดการทำงานนั้น เราอาจจะหาตำแหน่งของคำสั่งที่ผิดพลาดได้โดยวิธีใดวิธีหนึ่งต่อไปนี้คือ debugging traces หรือโดยวิธี debugging output ซึ่งกรรมวิธี debugging ทั้ง 2 ประเภทนี้จะได้กล่าวโดยละเอียดต่อไปภายหลัง ตัวอย่างกรณีที่เกิดผลทำให้เกิด system error คือ

1. Division by zero
2. Branching to a data area and attempting execution
3. Array subscripts incorrect
4. Numeric underflow or overflow

การหาที่ผิดพลาดในโปรแกรมเมื่อเกิดปัญหากรณีที่ 1 หรือที่ 2 นั้น เรามีวิธีการหาที่ผิดพลาดได้โดยการทำ reprogram to segment ดังที่อธิบายมาแล้ว หรือไม่ก็ลองใช้กรรมวิธีอื่นในการเขียนโปรแกรมเสียใหม่

กรณีที่ 3 Terminates Prematurely

กรณีที่โปรแกรมถูก compile เรียบร้อยแล้ว และเริ่มดำเนินปฏิบัติการ โดยให้ผล output ออกมาบางส่วน แล้วโปรแกรมนั้นก็ถูกขัดจังหวะและถูกหยุดการทำงาน ในกรณีเช่นนี้ ก็ให้ใช้วิธีการ debugging แบบธรรมดาที่เคยใช้กันมากเข้าช่วย

กรณีที่ 4 Incorrect Answers

หมายความว่า โปรแกรมรันแล้วปฏิบัติงานได้ให้ผลออกมาแต่ผลไม่ถูกต้อง ซึ่งกรณีเช่นนี้ไม่น่าหวาดวิตกอะไรมากนัก เพราะหมายความว่าทั้งตัวโปรแกรมเอง และ logic การทำงานก็เกือบจะได้ผลอยู่แล้วเพียงแต่จะต้องแก้ไขอีกนิดหน่อยก็จะได้รับผลสำเร็จ

กรณีที่ 5 An Infinite Loop

ความผิดกรณีเช่นนี้สามารถตรวจพบโดยไม่ยากลำบากนัก เพียงแต่เราพิจารณาเฉพาะส่วนของคำสั่งที่ทำงาน loop เท่านั้น ซึ่งเราอาจจะดำเนินการง่าย ๆ โดยการเพิ่มคำสั่ง PRINT อยู่ก่อนหน้าและหลัง loop ก็พอที่จะจับจุดได้ว่าเกิด infinite loop ขึ้นที่ใด

ในภาษาแต่ละภาษานั้นมีคำสั่งจะช่วยในการ debugging โดยจะแสดงขั้นตอนการปฏิบัติงานออกมาทุกคำสั่งในโปรแกรม ดังนั้น ถ้าไปหยุดที่คำสั่งใดแสดงว่าเกิด error ณ คำสั่งนั้น ๆ ตัวอย่างเช่น ภาษาฟอร์แทรนจะมีคำสั่ง DEBUG เช่นเดียวกับภาษา RPG ส่วนภาษาเบสิคจะมีคำสั่ง TRACE

ข้อแนะนำในการปฏิบัติสำหรับโปรแกรมที่มีซับรูทีนก็คือ เราไม่ควรจะเขียนซับรูทีนให้ยาวเกินไปนัก ตัวอย่างเช่น ไม่ควรให้ซับรูทีนยาวเกินกว่า 50 คำสั่ง ทั้งนี้เพราะถ้า ซับรูทีนยาวมาก ๆ จะมีปัญหาในการ debugging ในกรณีที่มีซับรูทีนยาวก็ควรจะแบ่งออกเป็น ซับรูทีนย่อยไปอีกซึ่งเรียกว่า ซับรูทีนซ้อน (nested subroutine) แต่การที่มีซับรูทีนซ้อนมาก ๆ ในหลายระดับก็ไม่ดีเช่นกัน เพราะเราจะต้องศึกษาถึงผลกระทบของซับรูทีนซ้อนอีก

ในช่วงที่เรา debugging โปรแกรมนั้น เราควรจะมีรายชื่อของตัวแปรของค่าคงที่ไว้ในมือเพื่อความสะดวกในการตรวจสอบ นอกจากนี้การ debugging ยังส่งผลให้เราต้องทำการรันโปรแกรมหลาย ๆ ครั้ง ตราบเท่าที่ยังมี bug อยู่ และในแต่ละครั้งนั้นก็จะมี output ปรากฏออกมา ดังนั้น เราควรจะระบุวันที่ เวลาที่รัน output นั้นออก เพื่อสะดวกกับการตรวจสอบในภายหลัง รวมทั้ง listing ของโปรแกรมที่ได้จากการรันในแต่ละครั้ง ทั้งนี้เพื่อป้องกันความสับสนในการตรวจสอบภายหลัง และถ้าจะให้ดีก็ไม่ควรให้ listing ของโปรแกรมเหล่านี้หายไปด้วย ในการแก้ไข bug แต่ละครั้ง เราควรจะต้องตระหนักให้ดีว่าจะไม่ก่อให้เกิด bug ชนิดอื่นเพิ่มเข้ามา

ปัญหาอันเกิดจากการไม่กำหนดตัวแปร

โดยปกติแล้ว error ประเภทที่พบบ่อย ๆ มักจะเป็นเรื่องเกี่ยวกับตัวแปรที่ใช้ เช่น กำหนดตัวแปรไม่ถูกต้อง หรือไม่กำหนดค่าเริ่มต้นให้กับตัวแปร การกำหนดตัวแปรในคำสั่ง output

หรือคำสั่งในการประมวลผล ซึ่งตัวแปรอาจจะปรากฏอยู่ทางด้านซ้ายหรือขวาของเครื่องหมาย = ในนิพจน์ของคณิตศาสตร์ หรืออาจจะเป็นตัวแปรที่ปรากฏในคำสั่ง input ก็ตาม เราอาจจะไปใช้ตัวแปรที่ไม่เคยกำหนดมาก่อนเลยก็ได้

ตัวอย่างของคำสั่งประมวลผลคณิตศาสตร์บางอัน เช่น

```
A = 1
B = B + A
```

ในคำสั่งที่ 2 B = B + A ทั้ง B ทางด้านขวามือของเครื่องหมายเท่ากับจะไม่เคยถูกกำหนดมาก่อน ดังนั้น compiler ในบางภาษาจะไม่ยอมปฏิบัติงานกับตัวแปรประเภทนี้ ซึ่งเราเรียกว่า undefined variable แต่สำหรับ compiler บางภาษาก็มีความสามารถในการปฏิบัติงานได้โดยทำการ difine ตัว undefined variable โดยการกำหนดให้มีข้อมูลเป็น 0 ปรากฏอยู่ การที่มีเหตุการณ์ของ undefined variable ปรากฏนั้น ขึ้นอยู่กับ 2 สาเหตุ คือ

By not initializing a variable before it is used

By typing error

สาเหตุประเภทแรกก็ได้กล่าวมาแล้ว ส่วนสาเหตุประเภทที่ 2 นั้นพวกโปรแกรมเมอร์คงจะเจอบ่อย ๆ ตัวอย่างเช่น

K0 มีความหมายเป็น Kศูนย์ หรือ Kโอ กันแน่

KI มีความหมายเป็น Kหนึ่ง หรือ Kไอ กันแน่

ดังนั้นเราควรจะแยกตัวโอ กับเลขศูนย์ให้เขียนแตกต่างกัน เพื่อสะดวกกับคนคีย์โปรแกรม

Storage Map

โดยทั่ว ๆ ไปแล้ว compilers มักจะมี option ซึ่งเรียกว่า storage map ความหมายของ storage map ก็คือ ตารางของชื่อตัวแปรทั้งหลายที่ปรากฏใน source program ดังนั้นเราจึงสามารถใช้ประโยชน์จาก storage map ได้โดยการตรวจสอบบรรดาตัวแปรทั้งหลาย เพื่อหาตัวแปรประเภท undefined variables ได้ โดยสร้างของ storage map นั้นจะเป็นตารางเรียงชื่อตามลำดับ เช่นในโปรแกรมของเราใช้ตัวแปรชื่อ VI แต่ปรากฏว่าใน storage map ปรากฏว่ามี VI แทนนั้น หมายความว่า เราคีย์ชื่อของตัวแปรผิดพลาด นอกจากนี้ compiler ในบางภาษายังมีความสามารถในการแยกแยะได้ว่า ตัวแปรใดเป็น explicit ตัวแปรใดเป็น implicit ได้อีกด้วย

Cross-Reference List

cross-reference list จะชี้ให้เห็นว่าตัวแปรตัวใดบ้างได้ถูกใช้ในโปรแกรม นอกจากนี้ cross-reference list จะชี้ถึงตำแหน่งของ label ทุกแห่ง, ฟังก์ชั่น, หรือ subroutine ที่ถูกอ้างอิงไปใช้ ซึ่งข้อมูลพวกนี้จะมีประโยชน์ต่อการ debugging เป็นอย่างมาก เพราะมีอยู่บ่อยครั้งเหมือนกันที่พบ bug จาก cross-reference list เช่นเรากำหนดตัวแปรไว้แต่ปรากฏว่าในโปรแกรมนั้นไม่ปรากฏชื่อของตัวแปรนั้นเลย ทั้งนี้เพราะเราอาจจะสะกดชื่อของตัวแปรผิดก็ได้ เราจะกำหนดให้มี cross-reference lists ได้จากการ request ใน job control language commonds

Typing Errors

การพิมพ์คำสั่ง หรือชื่อตัวแปรผิดใน source program นั้น อาจจะทำให้เกิด bug ชนิดที่หาตำแหน่งที่ผิดได้ยากลำบาก

มาตรการในการที่ลด error ชนิดนี้จะประกอบด้วย

1. การใช้แบบฟอร์มมาตรฐานในการ code โปรแกรม ทั้งนี้เพื่อลดความผิดพลาดจากการเขียนคำสั่งในคอลัมน์ที่ไม่ถูกต้อง

2. ให้เขียนโปรแกรมด้วยดินสอสีดำอย่างชัดเจน เพื่อสะดวกกับการอ่านและการแก้ไข

3. ให้ code โดยใช้ตัวอักษรตัวพิมพ์ใหญ่

4. ภายหลังการคีย์โปรแกรมเข้าเครื่อง หรือเจาะลงบัตรแล้ว จะต้องมีการตรวจสอบอีกครั้งว่าถูกต้องตาม coding form

ตัวอย่างของตัวอักษรที่มักจะคีย์ผิด

1 number	Z letter
I letter	7 seven
or	2 two
/ S l a s h	
' quotation mark	U Make the u round on the bottom plus a tail
1 not	V
7 seven	4 four (close the top of the four)
> greater than	

L letter
< less than

0 letter
Q letter
ϕ zero (strokes in opposite direction)

S letter (tails on the letter)
5 five

+ plus

D put tails on the letter D
0 letter

G letter
C letter
6 close the number

— break character
— minus

Desk Checking

เงื่อนไขนี้จำเป็นจะต้องดำเนินการเมื่อเราใช้บัตรเป็นโปรแกรม source desk ของเรา ทั้งนี้เพราะอาจจะเป็นได้ว่าการรันโปรแกรมในครั้งที่แล้วมีบัตรติดกับเครื่องอ่านแล้วฉีกขาดไป โดยที่ operator ไม่ได้แจ้งให้เราทราบ ดังนั้นทุกครั้งที่รับโปรแกรมกลับมาเพื่อจะรันใหม่ก็ควรจะต้องตรวจสอบว่า source deck ของเรายังอยู่ครบถ้วนดีหรือไม่

Input/Output Errors

ขั้นตอนแรกของงาน debugging ที่เราควรจะปฏิบัติก่อนอื่น ก็คือ การพิมพ์ input data ทั้งหมดออกมาตรวจสอบเสียก่อน ทั้งนี้เพราะเราพบว่า program errors นั้น มีสาเหตุมาจาก data error ในอัตราที่สูงมาก

data error นั้นอาจจะเกิดจากหลายสาเหตุด้วยกัน เช่น เจาะผิด, เข้าใจผิด, หรือกำหนดโครงสร้างของข้อมูลผิด ดังนั้น การพิมพ์ input data ทั้งหมดออกมาตรวจสอบด้วยสายตามนุษย์ก็เป็นหนทางอีกอันหนึ่งที่ช่วยลด error การปฏิบัติเช่นนี้เราจะใช้ศัพท์เรียกว่า echo checking (อ่าน input เข้า แล้วพิมพ์ผลที่อ่านออกมา)

การทำ echo checking นั้นถ้าจะให้ดีควรจะมีการพิมพ์ลาเบลของข้อมูลแต่ละรายการออกมาด้วย จะช่วยให้ผู้ตรวจสอบเข้าใจดียิ่งขึ้น โดยปกติแล้วภาษาบางภาษามีคำสั่งที่ทำงานในการพิมพ์ลาเบลของตัวแปรอยู่แล้ว เช่น ภาษา FORTRAN จะมี NAMELIST, ภาษา PL/I จะมี PUT DATA และภาษา COBOL จะมีคำสั่ง DISPLAY หรือ EXHIBIT ให้ใช้อยู่แล้ว ในกรณีที่ภาษาที่ท่านใช้อยู่นั้นไม่มีคำสั่งในลักษณะนี้ เราก็อาจจะเขียนคำสั่งอยู่ในรูปของโมดูล หรือซับโปรแกรมเพื่อทำงานนี้ก็ได้

Numerical Pathology

นิพจน์คณิตศาสตร์บางรูปนั้นอาจจะแฝง error ไว้โดยที่เราไม่สามารถตรวจพบได้ ดังตัวอย่างเช่น

$$X = 999.0,\ Y = -1000.,\ Z = .001$$

ดังนั้น นิพจน์คณิตศาสตร์

$$X + Y + Z + 1.0$$

ก็หมายถึง รูปแบบของการคำนวณดังนี้คือ

$$\begin{aligned}&((X + Y) + Z) + 1.0\\ &= ((999.0 - 1000.) + .001) + 1.0\\ &= (-1.0 + .001) + 1.0\\ &= -.999 + 1.0\\ &= .0001\end{aligned}$$

ในขณะที่ถ้าเราเขียนอยู่ในรูป

$$\begin{aligned}&X + (Y + Z) + 1.0\\ &= 999.0 + (-1000. + .001) + 1.0\\ &= 999.0 + (-1000.) + 1.0\end{aligned}$$

(loss of precision because of only four plrces.)

$$= -1.0 + 1.0 = 0.0$$

การที่สองนิพจน์นี้ ซึ่งความเป็นจริงตามสามัญสำนึกจะต้องได้ค่าออกมาเท่ากัน แต่ปรากฏว่าไม่เท่ากัน ทั้งนี้ก็เพราะขนาด precision ของเครื่องเป็นตัวกำหนด ดังนั้นปัญหาเรื่องนี้เราควรจะต้องคำนึงโดยถี่ถ้วนด้วย มิฉะนั้นจะเกิด error ได้

Locating Errors

การหาตำแหน่งที่ผิดภายในโปรแกรมนั้น เป็นกรรมวิธีที่ค่อนข้างจะยุ่งยาก เหตุผลที่เราต้องการจะทราบตำแหน่งของ error ก็เพื่อวัตถุประสงค์ที่ว่า

1. ไม่แน่ใจว่าโปรแกรมนั้นอยู่ในลักษณะที่เตรียมพร้อมจะปฏิบัติการหรือไม่

2. ขณะที่โปรแกรมนั้นอยู่ในลักษณะที่เตรียมจะปฏิบัติการ แต่เกิดการขัดจังหวะให้หยุดทำงานนั้นจะเกิดขึ้นเนื่องจาก system error หรือไม่

3. โปรแกรมได้มีการปฏิบัติงานแล้ว แต่ประสบปัญหาว่า การทำงานใน loop ไม่รู้จักจบ ทั้งนี้เนื่องจากโปรแกรมมีความยาวมากเกินไป

4. การทำงานของโปรแกรมให้ผลลัพธ์ที่ไม่ถูกต้อง

การ debugging กับโปรแกรมใดโปรแกรมหนึ่ง ถ้าหากผลที่เกิด error ในการรันโปรแกรมแต่ละครั้งนั้นให้ error ต่างกัน ทั้ง ๆ ที่เรายังไม่ได้แก้ไขคำสั่งใด ๆ ในโปรแกรมเลย อาจจะเป็นได้ว่าเกิดจากสาเหตุของ operator error, hardware error power fluctuation, หรือเกิดจากระบบควบคุมเครื่องผิดพลาดก็ได้ ในกรณีที่รันแต่ละครั้ง (โปรแกรมเดิม) แล้ว bug ออกมาต่างกัน เราควรจะนำ bug ในแต่ละครั้งมาเปรียบเทียบวิเคราะห์ บางครั้งอาจจะเป็นไปได้ว่าความผิดพลาดในกรณีนี้นั้นมีสาเหตุมาจาก undefined variable ก็ได้ แต่โดยปกติแล้วโปรแกรมเดียวกันถ้ายังมี bug ปรากฏอยู่ แล้วจะรันกี่ครั้งก็ยังคงให้ bug ประเภทเดียวกัน ยกเว้นกรณีที่กล่าวมาแล้ว

ความหมายของ locating errors ในโปรแกรมก็คือ การ search เพื่อหาตำแหน่งที่เกิด bug นั้นเอง ในบางกรณีเช่น system error การทำ locating errors อาจจะยุ่งยาก เพราะไม่มีข้อมูลใด ๆ มาช่วยวิเคราะห์ ลำดับแรกที่เราควรจะวิเคราะห์คร่าว ๆ ก่อนคือดูว่า bug ควรจะปรากฏที่ใด เช่น hardware bug, operating system bug, compiler bug หรือ program bug แต่โดยปกติแล้ว bug มักจะปรากฏอยู่ในโปรแกรมของเรามากกว่า ทั้งนี้เพราะปัจจุบันเทคโนโลยีการสร้างคอมพิวเตอร์ทั้งทางด้าน hardware และ software มีสูงมากจนความผิดพลาดในแหล่งต่าง ๆ ที่กล่าวมาเกิดขึ้นนั้นน้อยมาก ดังนั้นเมื่อท่านพบว่าแหล่งของ bug อยู่ที่ใด ก็จะช่วยร่นระยะเวลาในการตรวจสอบให้น้อยเข้า เช่น ถ้าเราพบว่า bug ปรากฏที่ program เราก็จะให้ความสนใจเฉพาะในโปรแกรมเท่านั้น

กรรมวิธีของการ debugging ในโปรแกรมนั้นก็อาจจะดำเนินการง่าย ๆ ในขั้นต้น โดยใช้แรงคนตรวจสอบ ก่อนอื่นให้พิจารณาดูว่าโปรแกรมของเรามีซับรูทีนหรือไม่ ถ้ามีก็ให้ตรวจสอบทีละซับรูทีนว่ามี bug ที่ใดหรือไม่ ลำดับถัดไปก็คือการแบ่งคำสั่งในโปรแกรมเป็นส่วน ๆ ที่เรียกว่า segment แล้วตรวจสอบดูว่ามี segment ใดมา ที่ทำให้เกิด bug กรรมวิธีในการตรวจสอบโปรแกรมลักษณะนี้เรียกว่า การ tracing ถ้าเราตรวจดูเฉย ๆ แล้วยังไม่พบ bug เราอาจจะเพิ่มคำสั่ง output เข้าไปทีละ 10—20 คำสั่ง โดยนำคำสั่ง output พวกนี้ไปแทรกยัง segment ที่แบ่งไว้เป็นตอน ๆ ไปตอนละคำสั่ง แล้วจึงนำโปรแกรมนี้ไปรัน แต่มีข้อแนะนำว่าไม่ควรนำ output statement เหล่านี้ไปใส่ภายใน loop เพราะผลที่ได้อาจจะสับสนจนหา bug ไม่พบก็ได้ การเพิ่ม

output statement ก็เหมือนกับการไล่วงจรไฟฟ้านั่นเอง เพื่อตรวจดูว่าวงจรส่วนใดขาด โดยการนำไฟฟ้าไปเสียบในแต่ละช่วงของวงจร ถ้าหากตรวจสอบช่วง 1 แล้วมีกระแสไหลผ่านก็แสดงว่าช่วงที่ 1 วงจรยังดีอยู่ ต่อไปก็ดูช่วงที่ 2 ช่วงที่ 3 ไปเรื่อย ๆ สมมุติว่า พบว่าช่วงที่ 6 กระแสไฟฟ้าไม่ไหลผ่านก็แสดงว่าวงจรไฟฟ้าในช่วงที่ 6 มีปัญหา ในลักษณะของการตรวจหา bug ในโปรแกรมก็มีลักษณะคล้ายคลึงกัน เพื่อจะหาช่วงของคำสั่ง (segment) มี bug อยู่ เช่นใน segment ที่มี bug อยู่นั้นประกอบด้วย 10 คำสั่ง เราก็มาพิจารณาทีละคำสั่งได้ แทนที่จะต้องดูทุกคำสั่งในโปรแกรม กรรมวิธีนี้จะลดระยะเวลาลงและทำให้เรากำหนดขอบเขตของ bug ได้ว่าอยู่ ณ ส่วนใดในโปรแกรม

คำสั่ง output ที่จะใส่ในโปรแกรมก็อาจจะใช้คำสั่งง่าย ๆ ว่าให้พิมพ์ข้อความว่า DEBUG 1, DEBUG 2, ..., DEBUG 10 ถ้าหากภายหลังการเพิ่ม output statement แล้วส่งโปรแกรมเข้ารันผลปรากฏออกมามีข้อความว่า

DEBUG 1

DEBUG 2

DEBUG 3

DEBUG 4

DEBUG 5

โดยมี DEBUG 5 เป็นข้อความสุดท้ายที่ออกมาก็แสดงว่าเกิด error ขึ้นระหว่างคำสั่งที่อยู่หลังคำสั่ง output statement DEBUG 5 เรื่อย ๆ มาจนถึงคำสั่งก่อนหน้า output statement DEBUG 6 ดังนั้นหน้าที่ต่อไปก็คือหาดูว่าคำสั่งใดในช่วงดังกล่าวเป็นคำศัพท์ที่ก่อให้เกิด error

ปัจจัยที่จะนำมาพิจารณาในการ locating an error ก็คือ point of detection and point of origin ความหมายของ point of detection ก็คือ ตำแหน่งที่ error จะเริ่มปรากฏ จุดนี้เป็นตำแหน่งแรกที่จะต้อง locate ต่อไป ดังตัวอย่างเช่น ถ้าเราพบคำสั่งใน segment ที่ 6 มีอยู่คำสั่ง คือ C = B/A ซึ่งคำสั่งนี้อาจจะผิดได้ถ้าหาก A มีค่าเป็นศูนย์ นั่นหมายความว่า คำสั่งนี้จะเป็น point of detection ส่วน point of origin ก็คือ ตำแหน่งที่เกิด error condition ถ้าพิจารณาตามตัวอย่างคำสั่ง C = B/A แล้ว ค่า A เริ่มมีค่าเป็น 0 ที่ใด ที่นั่นคือ point of origin เราจะเห็นว่า ection point จะถูกใช้เป็นจุดเริ่มต้นในการนำไปค้นหา error origin point เท่านั้นเอง

Debugging Output

กรรมวิธีนี้จะรวมถึงการทำ echo printing ที่กล่าวมาแล้วส่วนหนึ่ง และอีกส่วนหนึ่งก็คือการเพิ่มคำสั่ง output statement เพื่อแสดงผลลัพธ์ที่ได้จากการประมวลผลทีละส่วนในโปรแกรม กรรมวิธีนี้จะคล้าย ๆ กับการให้พิมพ์ข้อความออกมาในแต่ละ segment ของโปรแกรม แต่แทนที่จะให้พิมพ์คำว่า DEBUG เราก็ให้พิมพ์ผลจากการดำเนินงานในโปรแกรมเฉพาะส่วนของ segment นั้นเลย แล้วเราก็นำผลที่ได้จากการพิมพ์มาเปรียบเทียบกับค่าที่ถูกต้องจริง ๆ ของแต่ละขั้นตอน สิ่งที่พิมพ์อาจจะเป็นข้อมูลในตัวแปรที่ปรากฏ ถ้าหากพบว่าค่าใดผิดพลาดไปก็แสดงว่าการทำงานใน segment นั้นต้องมีคำสั่งหนึ่งคำสั่งใดผิดพลาด โดยปกติการดำเนินการเช่นนี้ควรจะทำพร้อม ๆ กับการเขียนโปรแกรม ไม่ใช่มาเสริมทีหลัง

กรรมวิธีของการเพิ่ม output statement ในโปรแกรมนั้น ภายหลังเมื่อเราแก้ไขโปรแกรมถูกต้องเรียบร้อยแล้ว แล้วเราไม่ประสงค์จะใช้ output statement พวกที่ช่วยในการ debugging อีกแล้ว ก็ให้เปลี่ยนคำสั่งเหล่านี้เป็น comment statement จะดีกว่าที่จะไปลบทิ้งไป เพราะเราอาจจะมีความจำเป็นจะต้องเรียกใช้ภายหลังอีกเมื่อไรก็ได้ โดยการแก้ไขง่าย ๆ และทั้งยังเป็น documentation statement ไปในตัวด้วย

ในกรณีของงานที่ใช้ระบบ terminal เราอาจจะกำหนดให้คำสั่งช่วยในการ debugging ดังกล่าวให้มีหมายเลขประจำคำสั่งให้แตกต่างไปจากคำสั่งอื่น ๆ ในโปรแกรม ตัวอย่างเช่น ถ้าเป็นภาษาเบสิคให้ลงท้ายด้วยเลข 9 จะได้เป็นที่สังเกตได้ง่าย

ในการพิมพ์เพื่อช่วย debuggin เราอาจจะใช้กระบวนการอื่น ๆ เข้ามาช่วย นอกจากที่กล่าวมานี้คือ

Selective Printout

การให้ output statement นี้เพื่อกำหนดการตรวจสอบเฉพาะเงื่อนไขบางอย่าง ตัวอย่างเช่น ตรวจสอบผลบางอย่างซึ่งเป็น error เช่น ตรวจสอบว่า ค่า X เป็นเลขลบหรือไม่ คือ

```
IF (X < = 0.0) THEN PRINT . . .
```

หรืออาจจะตรวจสอบว่า ค่า I เป็นเลขจำนวนเดิมหรือไม่ โดยใช้คำสั่ง

```
IF (I/5*5 – I = 0) THEN PRINT . . .
```

Logic Trace

ในกรณีของโปรแกรมชนิดที่มีซับรูทีน เราอาจจะใช้ output statement สำหรับกรณีของการตัดสินใจ, การกระโดดไปยังซับรูทีน และการกลับจากซับรูทีน ตัวอย่างของ output statement จะประกอบด้วยข้อความดังนี้คือ

```
ENTERED      SUBROUTINE      MAXNUM
EXITED       SUBROUTINE      MAXNUM
ENTEREO      SUBROUTINE      FIXNUM
LESS THAN ZERO BRANCH TAKEN.
ONE THOUSAND ITERATIONS
```

ข้อความที่ยกตัวอย่างมานี้จะชี้ให้เห็นว่า ขณะนั้นการทำงานจากโปรแกรมหลักจะกระโดดไปทำยังซับรูทีนใด ส่วนในข้อความที่ 4 นั้นจะแสดงว่าภายหลังการตัดสินใจแล้วจะกระโดดไปทำงานยังส่วนใดในโปรแกรมต่อไป ส่วนในข้อความสุดท้ายจะแสดงว่ามีอยู่ที่ iteration ซึ่งเกิดขึ้น เราอาจจะใช้ output statement เพื่อชี้สถานการณ์ที่พึงประสงค์ หรือไม่พึงประสงค์ก็ได้ขึ้นอยู่กับความต้องการ คำสั่ง output statement เหล่านี้จะถือได้เสมือน logic flow statement ซึ่งจะช่วยให้โปรแกรมเมอร์สามารถค้นหา bug ในโปรแกรมได้

โดยปกติแล้วถ้าหากว่าโปรแกรมนั้นทำงานไปตามปกติและจบด้วยตัวเองตามปกติ เรามักจะมี output statement ให้พิมพ์ข้อความว่า NORMAL END OF JOB ปรากฏอยู่ก่อนที่โปรแกรมจะหยุดการทำงานโดยปกติของมัน ทั้งนี้เพื่อช่วยให้โปรแกรมเมอร์ทราบว่า โปรแกรมนั้นจบด้วยการทำงานตามปกติ

Failure

ถ้าท่านพบว่ามี bug ปรากฏอยู่ในโปรแกรม แต่เราไม่สามารถตรวจพบชนิดของ bug และตำแหน่งที่เกิดได้ เราควรจะทำอย่างไรในกรณีที่ท่านคร่ำเคร่งอยู่กับการ debugging ในโปรแกรมนั้นเป็นเวลาหลาย ๆ วัน แต่ท่านก็ยังค้นหา bug ไม่พบ ท่านควรจะปฏิบัติตัวตามข้อแนะนำ 2 ทาง คือ ทางที่หนึ่งให้หยุดพักงานนั้นไว้ชั่วคราว โดยการหากิจกรรมอย่างอื่นที่จะช่วยผ่อนคลายสมองสักพักหนึ่ง เพราะในช่วงนี้ถึงท่านจะหยุดงาน debugging แต่สมองของท่านอาจจะยังคิดถึงเรื่องนี้อยู่ ภายหลังเมื่อร่างกายท่านสดชื่นแล้วจึงค่อยกลับไปทำงานต่อ ท่านอาจจะพบว่าในขณะที่ท่านกำลังพักผ่อนอยู่นั้น ท่านอาจจะหาคำตอบได้ว่า bug นั้นควรจะเกิดจากอะไร

สิ่งที่สองที่ท่านควรจะปฏิบัติก็คือ ให้หาผู้ร่วมคิด เช่น อาจจะเป็นผู้ร่วมงานคนใดก็ได้ที่เราสามารถปรึกษาหารือในงานนั้นได้ เพราะอาจจะเป็นไปได้ว่าภายหลังเมื่อเราได้เล่าถึงปัญหาและวิเคราะห์ถึงงานที่ทำแล้ว ตัวเราอาจจะเป็นผู้พบคำตอบเองก็ได้ หรืออาจจะหาคำตอบได้โดยฟังจากคำเสนอแนะของผู้ร่วมงาน

Defensive Programming

Defensive Programming หรืออาจจะเรียกว่า antibugging ก็ได้ หมายถึงกรรมวิธีของการเขียนโปรแกรมในรูปแบบที่จะทำให้ bug ปรากฏได้โดยการตรวจสอบ และยังทำให้ค้นหาตำแหน่งของ bug ได้โดยง่าย

เราสามารถจะขจัด bug ได้โดยการใช้เครื่องมือ debugging ในโปรแกรม โดยที่เราจะเรียก debugging aid ซึ่งสร้างในโปรแกรมเพื่อช่วยงาน debugging นี้ว่าเป็น arresting กรรมวิธีของการ debugging โดยวิธีนี้ก็คือการสร้าง bug-arresting statement ในโปรแกรมเพื่อตรวจสอบข้อมูลบางอย่าง ยกตัวอย่างเช่น การตรวจสอบพารามิเตอร์ก่อนที่จะส่งไปยังซับรูทีน หรือการตรวจสอบค่าของข้อมูลบางอย่างก่อนที่จะนำไปปฏิบัติงานด้วยฟังก์ชั่นทางคณิตศาสตร์บางอย่าง เช่น ฟังก์ชั่น square root, ฟังก์ชั่น logarithm เป็นต้น

defensive programming ประกอบด้วยหลักการดังนี้คือ

1. Mutual suspicion เป็นการตรวจสอบข้อมูลที่จะส่งไปปฏิบัติในโมดูลต่าง ๆ ว่าสามารถปฏิบัติงานได้หรือไม่ตามข้อกำหนดของโมดูลนั้น

2. Immediate detection เป็นการตรวจสอบหา error ให้เร็วที่สุดที่จะทำได้ เพื่อที่จะได้นำไปใช้ในการค้นหาตำแหน่งและแหล่งของความผิดพลาดที่เกิดขึ้น

3. Error isolation พยายามแยก error ที่เกิดขึ้นไม่ให้ไปมีผลกระทบต่อส่วนอื่น

โดยปกติแล้วเราก็มีความหวังว่า ข้อมูลที่พิมพ์จาก debugging aid จะช่วยในการตรวจสอบข้อมูลต่าง ๆ ที่จะส่งไปปฏิบัติในโมดูลต่าง ๆ และยังช่วยตรวจสอบผลของข้อมูลที่ได้จากการปฏิบัติงานในโมดูลด้วย การตรวจสอบข้อมูลว่ามีค่าเป็นไปได้หรือไม่ที่จะนำไปปฏิบัติการ เช่น มีค่าไม่เกินหรือไม่ต่ำกว่าขอบเขตที่จำกัดไว้จะช่วยให้การ debugging มาก เพราะ bug ที่เกิดจากข้อมูลนั้นมักจะปรากฏอยู่เสมอ โดยเฉพาะในงานที่มีข้อมูลมาก ๆ เราอาจจะเรียกกรรมวิธีที่กล่าวมานี้ว่าเป็น data filters ในเครื่องคอมพิวเตอร์บางระบบได้มีการสร้าง operating system ให้ทำหน้าที่เป็น data filter เพื่อตรวจสอบสถานภาพของ operating system ด้วยในตัว

กรรมวิธีในการจะตรวจสอบข้อมูลนั้นจะแบ่งออกเป็น 8 ประเภทดังนี้คือ

1. Data Type เป็นการตรวจสอบว่า รายการข้อมูลชนิดนั้นเป็นข้อมูลประเภท Alpha หรือ Numeric ทั้งนี้เพื่อความถูกต้องของการนำไปใช้งาน ตัวอย่างเช่น รายการเงินเดือนจะต้องเป็น numeric field

2. Range checks เพื่อตรวจสอบข้อมูลที่จะนำไปคำนวณ

3. Reasonability checks ตรวจสอบความสมเหตุสมผลของผลที่ได้จากการคำนวณตัวอย่างเช่น ภาษีที่คิดคำนวณได้นั้นไม่ควรจะสูงกว่าเงินเดือนที่ได้รับ

4. Total checks ตรวจสอบรายกลุ่มย่อย เพื่อดูความเป็นไปได้ของข้อมูลว่ามีอะไรผิดพลาดหรือไม่

5. Automatic checks อาจจะใช้ automatic check ที่กำหนดไว้เข้าช่วย เช่น overflow, underflow หรือ file label checking เข้าช่วย

6. Length check ถ้าหากเราทราบว่าขอบเขตสูงสุด หรือต่ำสุดควรจะเป็นเท่าใด ก็สามารถตรวจสอบได้ เช่น เราทราบว่ารหัสจังหวัดของประเทศไทยสูงสุดคือ 74 เราก็สามารถตรวจได้

7. Dog tags คำว่า dog tags จะหมายถึง รายการข้อมูลซึ่งปรากฏอยู่ใน field หรือ record ของข้อมูล ตัวอย่างเช่น ถ้าเรากำหนดว่าทุกระเบียบข้อมูล (record) จะต้องมีตัว MST ปรากฏอยู่ที่สดมภ์ที่ 73—75 เราก็จะต้องตรวจดูว่าระเบียบข้อมูลนั้นมีเครื่องหมายนี้ปรากฏอยู่จริงหรือไม่ ณ ที่ตั้งดังกล่าว

8. check digits ก็คือตัวเลขที่สร้างขึ้นมาเพื่อทำหน้าที่ควบคุมข้อมูลในรายการหนึ่งว่ามีความถูกต้องหรือไม่ ลักษณะของ check digits จะทำหน้าที่เสมือน parity bit ในระบบ hardware เราอาจจะเรียกกรรมวิธีการ coding ที่สร้าง check digits ว่าเป็น redundancy code

ขอให้จำไว้ว่า ข้อมูลเปรียบเสมือนวัตถุดิบที่โปรแกรมและคอมพิวเตอร์จะนำไปผลิตเป็นสินค้า ดังนั้นถ้าหากข้อมูลยังอยู่ในสภาพที่ไม่ถูกต้อง การประมวลผลข้อมูลนั้นจะได้รับผลผิดพลาดไปด้วย ดังคำพูดที่ว่า GIGO (Garbage In, Garbage Out)

Assertions

ในยุคปัจจุบันนี้ ภาษาหลาย ๆ ภาษาของคอมพิวเตอร์มีความสามารถในด้าน assertion ความหมายของ assertions ก็คือความสามารถในการสร้างเงื่อนไขกำกับสถานการณ์ในการปฏิบัติการของโปรแกรมไว้ โดยทั่วไปแล้วเราอาจจะแบ่งประเภทของ assertion ได้เป็น 2 แบบ คือ

1. global assertions ซึ่งจะหมายถึงการกำหนดเงื่อนไขของสถานการณ์เอง โดยโปรแกรมเมอร์ ตัวอย่างเช่น การกำหนดให้ N เป็นเลขจำนวนเต็มบวก นั่นหมายความค่า N จะต้องเป็นบวก และเป็นจำนวนเต็มอยู่ตลอดไป

2. local assertions หมายถึงการกำหนดเงื่อนไขในรูปของรหัสโดยอนุญาตให้ผู้ใช้สามารถใส่ค่าใด ๆ ที่ต้องการได้ในแต่ละส่วนของปฏิบัติการในโปรแกรมนั้นเอง ตัวอย่างเช่น เราอาจจะกำหนดว่าเงินค่าจ้างต่อสัปดาห์จะต้องมีค่าไม่เกิน $2,000 เป็นต้น ดังนั้นเมื่อใดก็ตามที่มีปัญหาว่าข้อมูลไม่สอดคล้องกับกติกาที่เราตั้งไว้นี้ โปรแกรมจะหยุดการประมวลผล แล้วพิมพ์ข้อความที่เกิดปัญหานั้นออกมา การกำหนด assertion ประเภทที่ 2 นี้ทำได้โดยง่าย โดยการใช้คำสั่ง IF เข้าช่วย นอกจากนี้ในภาษาบางภาษายังเอื้อให้กับโปรแกรมเมอร์ได้สามารถเลือกกำหนด assertions ในลักษณะที่จะส่งผลเป็น checking code ในขณะที่ compiler อยู่ใน debug mode ออกมาอีกด้วย

A **Catalog** of **Bugs** (A Classification **of bugs** by type)

These are not syntax errors but bugs that would still be present after syntax checking is complete.

Logic

1. Taking the wrong path at a logic decision.
2. Failure to consider one or more conditions.
3. Omission of coding one or more flowchart boxes.
4. Branching to the wrong label.

Loops

1. Not initializing the loop properly.
2. Not terminating the loop properly.
3. Wrong number of loop cycles.
4. Incorrect indexing of the loop
5. infinite loops (sometimes called closed loops).

Data

1. Failure to consider all possible data types.
2. Failure to edit out incorrect data.
3. Trying to read less or more data than there are.
4. Editing data incorrectly or mismatching of editing fields with data fields.

Variables.

1. Using an uninitialized variable.
2. Not resetting a counter or accumulator.
3. Failure to set a program switch correctly,
4. Using an incorrect variable name (that is, spelling error using wrong variable).

Arrays

1. Failure to clear the array.
2. Failure to declare arrays large enough.
3. Transpose the subscript order.

Arithmetic Operations (see also Variables)

1. Using wrong mode (i.e., using integer when real was needed).
2. Overflow and underflow.
3. Using incorrect constant.
4. Evaluation order incorrect.,
5. Division by zero.
6. Square root of a negative value.
7. Truncation.

Subroutines

1. Incorrect attributes of function.
2. Incorrect attributes of subroutine parameters
3. Incorrect number of parameters.
4. Parameters out of order.

Input/Output (see also Data)

1. Incorrect mode of I/O format specifications.
2. Failure to rewind (or position) a tape before reading or writing.
3. Using wrong size records or incorrect formats.

Character Strings

1. Declare character string the wrong size.
2. Attempting to reference a character outside the range of the string length.

Logico! Operations

1. Using the wrong logical operator.
2. Comparing variables that do not have compatible attributes.
3. Failure to provide ELSE clause in multiple IF statements.

Machine Operations

1. Incorrect shifting.
2. Using an incorrect machine constant (i.e.; using decimal when hexadecimal was needed).

Terminators

1. Failure to terminate a statement.
2. Failure to terminate a comment.
3. Using " instead of ', or vice versa.
4. Incorrectly matched quote.
5. Terminate prematurely.

Miscellaneous

1. Not abiding by statement margin restrictions.
2. Using wrong function.

There is another category of bugs that will be called special bugs here. They are sophisticated errors (i.e.. difficult to locate).

Semantic Error

These errors are caused by the failure to understand exactly how a command works. An example is to assume that arithmetic operations are rounded. Another example is to assume that a loop will be skipped if the ending value is smaller than the initial value. In IBM FORTRAN DO, loops are always executed once.

Semaphore Bug

This type of bug becomes evident when process A is waiting on a process B while process B is waiting upon process A. This type of bug usually emerges when running large complicated systems, such as operating systems. This is called the ***deadly embrace.***

Timing ***Bug***

A ***timing bug*** can develop when two operations depend on each other in a time sense. That is, operation A must be completed before operation B can start. If operation B starts too soon, a timing bug can appear. Both timing bugs and semaphore bugs are called situation ***bugs.***

Operation Irregularity Bugs

These bugs are the result of machine operations. Sometimes unsuspecting programmers do not understand that the machine does arithmetic in binary; so the innocent expression

```
1.0/5.0*5.0
```

does not equal one. This error shows up when this test is made.

```
A = 5.0

B = 5.0
  .
IF (1.0/A*B .EQ. 1.0) . . .
```

Error Checklist

มีข้อที่น่าสังเกตว่า โปรแกรมเมอร์บางคนมักจะทำ error ในลักษณะเดียวกันปรากฏขึ้นมาเสมอ ๆ ในสไตล์เดียวกัน เช่น ชอบเขียนตัวแปรลำดับผิด, ชอบใช้ conditional jump ผิด ฯลฯ ดังนั้นถ้าหากโปรแกรมเมอร์ได้มีการตรวจสอบโปรแกรมของตนกับ check list error ก็จะเป็นส่วนหนึ่งในการขจัด bug ในขั้นต้น

Program Dimensions

การ debugging นั้นมีอยู่ 2 มิติที่เราจะต้องตรวจสอบคือ space และ time

มิติของ space หมายถึง storage space ของคอมพิวเตอร์นั่นเอง

มิติของ time หมายถึง ช่วงระยะเวลาในการปฏิบัติการเริ่มแต่แรกจนจบการปฏิบัติการ

เราจะใช้ debugging aid เพื่อตรวจสอบมิติทั้งสองในการแยกแยะปัญหา และตำแหน่งที่เกิด error

Debugging Aids

นอกเหนือจาก debugging aids ที่โปรแกรมเมอร์จะคิดสร้างขึ้นมาใช้เองในโปรแกรมแล้ว ยังมีผู้รายงานสรุปและข้อเสนอของการใช้ debugging aids ไว้หลายประการด้วยกัน เราอาจจะสรุป debugging aids ที่มีประสิทธิภาพไว้ได้ 6 วิธีดังนี้คือ

1. Dumps
2. Flow trace
3. Variable trace
4. Subroutine Trace
5. Subscript check
6. Display

Dump จะหมายถึง ระเบียนข้อมูลหนึ่งซึ่งปรากฏในช่วงระยะเวลาหนึ่ง ซึ่งโปรแกรมปฏิบัติงานอยู่ ปกติแล้วจะออกมาในรูปของภาษาเครื่อง ซึ่งทำความเข้าใจลำบาก ดังนั้น วิธีนี้จึงเหมาะกับผู้ที่คุ้นเคยอยู่กับภาษาเครื่อง

Trace หมายถึง ข้อมูลซึ่งอยู่ในส่วนของการดำเนินงานของโปรแกรม การดูจาก trace ก็เพื่อช่วยตรวจสอบว่า โปรแกรมนั้น ๆ มีการปฏิบัติงานเรียงลำดับกิจกรรมต่าง ๆ อยู่ในลักษณะ

ที่ตรงกับความต้องการของโปรแกรมเมอร์หรือไม่ และตัวแปรต่าง ๆ ที่ใช้เก็บข้อมูลนั้นได้ผลตรงกับวัตถุประสงค์หรือไม่ เราอาจจะแบ่ง trace ออกเป็น 3 ประเภทคือ

ประเภทที่ 1 แสดงทิศทางการปฏิบัติงานภายใต้การควบคุมของโปรแกรม ดังนั้นจะมีการพิมพ์ข้อความในแต่ละขั้นตอนที่มีการปฏิบัติงานผ่านลำดับขั้นตอนนั้น

ประเภทที่ 2 แสดงผลของชื่อตัวแปร และข้อมูลของตัวแปรที่ได้จากการปฏิบัติงานในโปรแกรม โดยที่แต่ละครั้งที่มีการเปลี่ยนแปลงค่าของข้อมูลของตัวแปรนั้นก็จะมีการพิมพ์ข้อมูลนั้นออกมา ในกรณีที่มีตัวแปรใหม่ปรากฏก็จะพิมพ์ออกมาด้วย

ประเภทที่ 3 เป็น traces subroutines calls trace ประเภทนี้จะมีประโยชน์มากสำหรับโปรแกรมที่มีอยู่หลาย ๆ subroutines โดยที่แต่ละครั้งที่มีการเรียกซับรูทีนเข้ามาทำงานก็จะมีการพิมพ์ชื่อของซับรูทีนนั้นออกมา และภายหลังที่กลับจากซับรูทีนมายังโปรแกรมหลักก็จะมีการพิมพ์ return message ออกมาด้วย

กรรมวิธี traces นี้จะช่วยให้ข้อมูลและตำแหน่งของ bug ในโปรแกรม แต่วิธีการนี้ก็มีข้อเสียตรงที่จะมีข้อความมากมายจากโปรแกรม จึงทำให้เสียเวลาตรวจสอบและเสียเวลาของเครื่องเพิ่มขึ้นจากปกติถึง 10 ถึง 40 เท่าตัวของเวลาที่ใช้ในการรันโปรแกรม

ดังนั้นเพื่อให้การ trace มีประสิทธิภาพ เราจึงควรจะมีการวางแผน design flow trace ก่อนที่จะ trace โปรแกรมโดยที่ให้ flow trace อยู่ในสภาพ on หรือ off ได้ตามความต้องการ เช่นอาจจะให้ turned on เฉพาะในส่วนของโปรแกรมที่เราคิดว่าจะมี bug ปรากฏอยู่ ส่วนโปรแกรมใน section อื่นก็ให้อยู่ในสภาพ turned off

Variables Traces นั้นเราอาจจะวางแผนให้มีเฉพาะตัวแปรที่เราสงสัยเท่านั้นจึงจะทำการ trace ถ้าเป็นตัวแปรอื่น ๆ ที่ไม่น่าจะมี bug เราก็ไม่ trace จะได้ประหยัดเวลา

Subroutine Traces ก็อาจจะปฏิบัติการได้ในลักษณะเดียวกับ Variables Traces ได้

Subscript Check ให้ตรวจดูว่าตัวแปรที่เก็บค่าบ่งลำดับของสมาชิกไว้นั้นมีเกินจำนวนสูงสุดที่กำหนดไว้หรือไม่ ถ้าหากเกินหรือถ้าหากตัวบ่งลำดับมิติที่เป็นไปได้ เช่น เป็นเลขลบ หรือทศนิยม ก็ให้พิมพ์ข้อความออกมาเพื่อบ่งถึงสภาพเหตุการณ์นั้น ๆ

Display เป็น debugging command ที่จะให้ผู้ใช้เลือกการแสดงผลพิมพ์ทางจอภาพ

กรรมวิธีของการ debugging ที่กล่าวมาตั้งแต่ต้นนั้นเป็นแนววิธีปฏิบัติที่เราสามารถใช้กับระบบ batch processing ในกรณีของการประมวลผลในระบบ online นั้นเราก็สามารถจะใช้

กรรมวิธีของการ debugging ในระบบ batch มาใช้ได้ นอกเหนือจากนี้ยังมีกรรมวิธีโดยเฉพาะสำหรับงานในระบบ online ที่เพิ่มเติมขึ้นมาอีกดังนี้คือ

1. *Breakpoints* คือการสั่งให้เครื่องหยุดปฏิบัติการกับโปรแกรมนั้นชั่วคราว เพื่อให้โปรแกรมเมอร์ตรวจสอบผลบางอย่าง

2. *Error bearkpoint* ความประสงค์ของผู้ใช้ในที่นี้ก็คือ เมื่อการดำเนินงานของโปรแกรมประสบกับเหตุการณ์ที่ไม่ถูกต้องก็ไม่ควรจะประมวลผลต่อไป แต่ให้โปรแกรมนั้นกลับมาอยู่ภายใต้การควบคุมของผู้ใช้ ตัวอย่างเช่น ในโปรแกรมพบว่าจะต้องถอดรูทของเลขติดลบ หรือลำดับของอะเรย์สูงกว่าค่าที่กำหนดไว้

3. *Examining and modification* เมื่อใดก็ตามที่โปรแกรมหยุดการทำงานอันเนื่องจาก breakpoint หรือเกิดจาก error ก็ตามเราอาจจะต้องการความเป็นอิสระและต้องการความสามารถในการเปลี่ยนแปลงค่าของข้อมูลในตัวแปรได้ เพื่อวัตถุประสงค์บางอย่างลักษณะความต้องการเช่นนี้จะทำได้เฉพาะในระบบ online เท่านั้น

4. *Restarts* ความสามารถอีกอย่างที่สามารถกระทำได้ในระบบ online ก็คือ การ restart โปรแกรมได้ เช่น เราอาจจะทำการ restart หลังจาก breakpoint

5. *Program modification* ในระบบ online เรามีความสามารถที่จะ insert, delete หรือ modify ในบางคำสั่งที่ต้องการได้

ในระบบ **Time sharing** บางอันมีความสามารถพิเศษ ตัวอย่างเช่น BASIC PLUS ของเครื่อง PDP 11 มีความสามารถพิเศษในการใช้ debugging aid ซึ่งทำงาน debugging ได้รวดเร็วมาก

ตารางต่อไปนี้จะแสดงเวลาโดยประมาณที่ใช้ในงานแต่ละขั้นตอน

Task	**Units**
Planning	1
Writing	1
Debugging	4
Testing	1

จะเห็นได้ว่าเวลาที่ใช้ในการ debugging มากกว่าเวลารวมทั้งหมดของงานอื่น ๆ

Preventing Bugs งาน debugging นั้นนับว่าเป็นงานที่เสียค่าใช้จ่ายมากที่สุด และบางครั้งดูน่าเบื่อหน่ายสำหรับโปรแกรมเมอร์อีกด้วย ดังนั้นการเขียนโปรแกรมของเราต้องพยายามหลีกเลี่ยงการจะก่อให้เกิด bugs มากที่สุด กฎเกณฑ์เบื้องต้นต่อไปนี้คงจะช่วยลด bugs ที่เกิดขึ้นได้บ้าง

1. Avoid questionable coding

พยายามหลีกเลี่ยงที่จะใช้วิธีการหรือสูตรที่ยุ่งยากซับซ้อนโดยเฉพาะอย่างยิ่งกรรมวิธีที่เรายังไม่คุ้นเคยใช้มาก่อนเลย พยายามใช้รูปแบบวิธีการที่ดูแล้วเข้าใจง่าย

2. Avoid dependence on defaults

ขอให้ระลึกไว้เสมอว่า ภาษาคอมพิวเตอร์แต่ละภาษานั้นจะมีส่วนของ language default ซึ่งเป็นที่รับรู้ของ compiler อยู่แล้ว การใช้ default เหล่านี้อาจจะเป็นประโยชน์แก่โปรแกรมเมอร์ในขณะนั้น แต่อาจจะกลายเป็นอันตรายต่อไปภายภาคหน้าเมื่อมีการเปลี่ยนระบบหรือมีการเปลี่ยนแปลง default ไปจากเดิม

3. Never allow data dependency

โปรแกรมเมอร์จะต้องพึงระลึกไว้อยู่เสมอว่า อย่าพยายามเขียนโปรแกรมโดยพึ่งพิงอยู่กับข้อมูลในรูปแบบใดรูปแบบหนึ่ง พึงระลึกเผื่อไว้สำหรับกรณีของ input data ทุกรูปแบบที่เป็นไปได้เสมอ

4. Always complete your logic decisions

ในกรณีที่ input data มีเพียง 2 code คือ 1 หรือ 2 โปรแกรมเมอร์ไม่ควรจะตรวจสอบเฉพาะ code 1 โดยตั้งเงื่อนไขว่า ถ้าไม่ใช่ 1 จะต้องเป็นรหัส 2 ซึ่งความเป็นจริงอาจจะมีรหัสอื่นซึ่งเราเจาะผิดปรากฏเข้ามาก็ได้ ดังนั้นจึงควรหลีกเลี่ยงคำสั่งประเภทนี้ คือ IF (code = 1) is flase Then assume (code = 2) แต่ให้เขียนในลักษณะนี้แทนคือ If (code = 1) is false and (code = 2) is false Then print message error นั่นหมายความว่าถ้าเรามี code ถึง N ทางที่เป็นไปได้ เราจะต้องตรวจสอบถึง (N + 1) เงื่อนไข

5. Operator independence

ขอให้พึงระลึกไว้เสมอว่า operator อาจจะทำอะไรผิดพลาดได้ ทั้งนี้เพราะ operator ทำงานอยู่กับคอมพิวเตอร์ทั้งวัน อาจจะมีอะไรที่ผิดพลาดได้ ดังนั้นถ้าเป็นไปได้ไม่ควรจะสั่งให้ operator ทำอะไรพิเศษมากกว่างานปกติ งานอะไรที่อยู่ในขอบเขตที่เขียนลงในโปรแกรมได้เราควรจะทำเองมากกว่า ภายหลังการรันโปรแกรมแล้วพบว่ามี bug เราอาจจะตรวจสอบดูการทำงานของ operator ได้ว่ามีอะไรผิดพลาดบ้างไหม เช่น mount tape ผิด หรือให้ JCL ผิดพลาด

PROGRAMMING MAXIMS

Use a debugging compiler

Deck check first

Echo check input data

Introduce debugging aids early

check input for reasonableness

Find out which debugging aids are available

Get it right the first time

แบบฝึกหัด

1. จงอธิบายคำต่อไปนี้

    1. Abend
    2. Dererminacy
    3. Storage map
    4. Cross-reference list
    5. Echo checking
    6. Cancellation
    7. Bug arresting
    8. Assertions
    9. GIGO
    10. Breakpoint

2. จงอธิบายถึงข้อดีข้อเสียของ debugging techniques แต่ละวิธี
3. Trace มีไว้เพื่อประโยชน์อย่างไร และ Trace แบ่งออกเป็นกี่ประเภท
4. การ debugging subroutine ทำได้อย่างไรบ้าง
5. จงบอกถึงความแตกต่างระหว่าง syntax error และ execution error