Part A Programming Language Concepts

2 Names and Bindings

ในการเขียนโปรแกรมไม่ว่าภาษาอะไรก็ตาม จะมีการตั้งชื่อ (Naming) ให้กับสิ่งต่าง ๆ ภายในโปรแกรมเพื่อให้ง่ายต่อการเขียนและการทำความเข้าใจ เช่น ชื่อตัวแปร ชื่อโปรแกรม ชื่อคลาส ชื่อฟังก์ชัน ชื่อ type เนื้อหาในบทนี้จะอธิบายเกี่ยวกับการตั้งชื่อภายในโปรแกรมอย่างถูกต้องเหมาะสมสำหรับแต่ละภาษาโปรแกรม

2.1 Names and Bindings

ชื่อ (Name) คือ อัตลักษณ์ (identifier) สำหรับใช้อ้างอิงถึงสิ่งต่าง ๆ ในการเขียนโปรแกรม เช่น ตัวแปร (variable) ค่าคงที่ (constant) ชนิดข้อมูล (type) เป็นต้น แทนที่การอ้างอิงในระดับต่ำอย่างการอ้างที่อยู่ของหน่วยความจำเป็นตัวเลขโดยตรงซึ่งจดจำและแปลความหมายได้ยาก ชื่อจึงถูกนำการใช้แทนการอ้างอิงด้วยตัวเลขทำให้การอ้างอิงง่ายต่อการจดจำและทำความเข้าใจโดยที่ผู้เขียนโปรแกรมไม่ต้องทราบถึงตัวเลขจริง ๆ ที่ชื่อนั้นอ้างถึง แต่จะถูกเชื่อมโยงกันโดยตัวแปลภาษาอย่าง Compiler และ Interperter เรียกว่า Binding หรือการยึดเหนี่ยว

Binding เป็นการที่ยึดเหนี่ยวชื่อที่ตั้งขึ้นกับสิ่งที่ชื่อนั้นอ้างถึง เช่น ที่อยู่ในหน่วยความจำ (memory address) โดย Binding จะมีวิธีการต่าง ๆ กันอยู่ 2 แบบจำแนกตามขณะเวลาที่ทำการยึดเหนี่ยว (binding time) เป็นช่วงเวลาที่ชื่อค่าจริงต่าง ๆ ถูกกำหนดให้กับชื่อที่ตั้งไว้ ได้แก่

2.1.1 Static binding

การยึดเหนี่ยวที่ถูกทำตั้งแต่ก่อนช่วงเวลาดำเนินงาน(run-time) เรียกอีกอย่างว่า early binding นั่นคือ ชื่อทุกชื่อจะทราบค่าที่อ้างถึงจริงตั้งแต่ก่อนการดำเนินงาน เช่น ชื่อตัวแปรก็จะทราบค่าที่อยู่ของหน่วยความจำที่อ้างถึง เป็นต้น

วิธีการนี้มีข้อดีในด้านประสิทธิภาพ (efficiency) ในการประมวลผล เนื่องจากทราบแน่ชัดถึงปริมาณทรัพยากรที่จะใช้ตั้งแต่ก่อนเริ่มทำให้สามารถวิเคราะห์และจัดสรรการใช้ทรัพยากรอย่างคุ้มค่าได้ เนื่องจากในชุดคำสั่งที่แปลได้ชื่อจะถูกแทนที่ด้วยค่าจริงหรือในการกระทำการจะอ้างอิงด้วยค่าจริงโดยตรง เช่น ตัวแปรชื่อ x ซึ่งอ้างตำแหน่งข้อมูลในหน่วยความจำ เมื่อการยึดเหนี่ยวเกิดขึ้นแล้ว หน่วยความจำจะถูกจองตามชนิดและขนาดของตัวแปรและนำค่าที่อยู่หน่วยความจำที่จองไว้อย่างค่า 1000 ไปกำหนดเป็นค่าอ้างอิงของตัวแปรชื่อ x ซึ่งเมื่อมีการกระทำกับตัวแปร x คำสั่งก็จะสั่งให้กระทำกับหน่วยความจำ ณ ตำแหน่งที่ 1000 โดยตรง

Static binding จะนิยมใช้ในภาษาโปรแกรมที่ใช้การแปลด้วย Compiler

2.1.2 Dynamic binding

การยึดเหนี่ยวที่ถูกทำระหว่างช่วงเวลาดำเนินงาน(run-time) เรียกอีกอย่างว่า late binding นั่นคือ ชื่อที่ตั้งจะยังไม่ทราบค่าจริงจนกว่าจะถูกเรียกใช้งาน

วิธีการนี้มีข้อดีในด้านความยืดหยุ่น (flexibility) เพราะการตัดสินใจในการกำหนดค่าจริงให้กับชื่อเกิดขึ้นในช่วงเวลาดำเนินการทำให้ชื่อนั้นง่ายต่อการเปลี่ยนแปลงค่าจริงได้ เช่น การเปลี่ยนชนิดข้อมูลที่จัดเก็บในหน่วยความจำของชื่อตัวแปรในภาษาที่ใช้วิธีการนี้ซึ่งเรียกว่า dynamic language อย่างภาษา Python โดยพิจารณาจากโค้ด ดังต่อไปนี้

```
x = "hello"
x = 3
print x * 2
```

(ภาษา python)

ตัวแปร reuse ได้

จากตัวอย่าง มีการกำหนดค่าให้กับตัวแปรชื่อ x ทั้งหมด 2 ครั้ง โดยบรรทัดแรกเป็นข้อมูลชนิดข้อความมีค่าเท่ากับ hello และบรรทัดต่อมา มีกำหนดค่าใหม่เป็นจำนวนเต็มค่าเท่ากับ 3 ซึ่งทำให้ผลในการดำเนินงานโปรแกรมนี้ในบรรทัดสุดท้าย ได้ผลลัพธ์เท่ากับ 6 ซึ่งเป็นการทำคำสั่งพิมธ์ผลคูณของตัวเลข แทนที่จะเป็นคำสั่งพิมพ์ข้อความซ้ำ 2 ครั้ง (กรณีค่า x เท่ากับ hello ผลลัพธ์ในภาษา Python จะเป็น hellohello) เนื่อง binding เกิดในขณะดำเนินงาน ดังนั้น ขณะกระทำการคำสั่ง print จึงทราบได้ว่าค่าและชนิดของตัวแปรชื่อ x ได้เปลี่ยนไปแล้ว ซึ่งเกิดความยืดหยุ่นในการตั้งชื่อและใช้งานตัวแปร

Dynamic binding จะนิยมใช้ในภาษาโปรแกรมที่กระทำการด้วย Interpreter

2.2 Object Lifetime and Storage Management

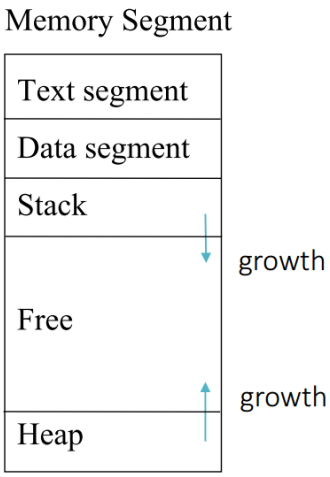
ช่วงชีวิต (Lifetime) คือ ช่วงเวลานับตั้งแต่สิ่งใดสิ่งหนึ่งเกิดจนกระทั่งดับไป เช่นเดียวกันกับ object เมื่อถูกตั้งชื่อแล้วนั้นก็จะมีช่วงเวลาตั้งแต่เกิดจนดับ เรียกว่า object lifetime ตัวอย่างเช่น เราตั้งชื่อตัวแปรให้กับ object พื้นที่ของหน่วยความจำตำแหน่งหนึ่ง ๆ จะถูกจองสำหรับตัวแปรนั้น ช่วงชีวิตของข้อมูลในหน่วยความจำสำหรับ object จะมีช่วงชีวิตตั้งแต่เริ่ม allocate พื้นที่จนกระทั่งถูก deallocate ไป คือ ไม่มีชื่อตัวแปรที่อ้างไปยังหน่วยความจำตำแหน่งนั้นอีกต่อไปแล้ว นั่นคือ object lifetime จะเป็นช่วงเวลาที่ยังคงถูกจัดสรรพื้นที่ไว้ในที่จัดเก็บ (Storage) ไว้อยู่

ช่วงชีวิตของแต่ละ object จะมีระยะเวลาแตกต่างกันไป ยกตัวอย่างเช่น โปรแกรมหนึ่ง ๆ เมื่อถูกกระทำการ ระบบปฏิบัติการจะการจัดสรรพื้นที่หน่วยความจำโดยมีโครงสร้างดังภาพ

ภาพที่ 2.1 แสดงโครงสร้างหน่วยความจำที่ระบบปฏิบัติการจัดสรร (allocate) ให้กับโปรแกรม

คำบรรยายภาพ: แสดงโครงสร้างการจองหน่วยความจำ โดยแบ่งพื้นที่หน่วยความจำออกเป็นส่วนต่าง ๆ เรียงกันไป ได้แก่

1. Text Segment
2. Data segment
3. Stack ซึ่งสามารถขยายกินพื้นที่ไปยังส่วนที่ 4 ได้
4. Free แสดงถึงพื้นที่ที่ยังไม่ได้ใช้ในการจัดเก็บข้อมูล
5. Heap ซึ่งสามารถขยายกินพื้นที่ไปยังส่วนที่ 4 ได้



จากภาพการจัดสรรหน่วยความจำจะมีส่วนของของ text segment หรืออาจเรียกว่า code segement เป็นส่วนสำหรับจัดเก็บลำดับคำสั่งของโปรแกรม ซึ่งเมื่อถูกสั่งกระทำการคำสั่งของโปรแกรมที่เป็นภาษาเครื่องจะถูกจัดเก็บไว้ในส่วนนี้เพื่อนำเข้าไปยังหน่วยประมวลผลทำการประมวลผลต่อไป

ส่วนถัดมาคือ Data segment สำหรับจัดเก็บข้อมูลกลุ่มค่าคงที่หรือขนาดคงที่ เช่น ตัวแปร global, ค่าคงที่ (constant) เป็นต้น ซึ่งจะมีช่วงชีวิตตลอดช่วงเวลาของการกระทำการของโปรแกรม

** พื้นส่วนของ text segment และ data segment จะเป็นส่วนที่เรียกว่า static object คือ พื้นที่หน่วยความจำจองไว้จะมีขนาดที่คงที่ และไม่มีการเปลี่ยนแปลงตลอดการทำงานของโปรแกรม และสำหรับส่วนที่เหลือจะเป็นส่วนที่ขนาดไม่คงที่ ได้แก่ Stack และ Heap โดยมีลักษณะการจัดสรร โดย Stack จะเริ่มจองจากทางหัวของพื้นที่ และHeap เริ่มจองจากด้านท้ายกินพื้นที่ว่างซึ่งคั่นกลางอยู่ระหว่าง 2 ส่วนนี้

ส่วน Stack เป็นส่วนของหน่วยความจำที่เก็บข้อมูลในลักษณะ LIFO สำหรับใช้เก็บข้อมูลและสถานะในการเรียกใช้งานโปรแกรมย่อย (Subroutine) เพื่อการเรียกฟังก์ชันแล้วสามารถกลับมาประมวลผลยังตำแหน่งที่เรียกได้

ส่วน Heap เป็นส่วนสำหรับจัดเก็บข้อมูลในการประมวลผลของโปรแกรม ซึ่งขนาดจะเปลี่ยนไปตามจองพื้นที่หรือคืนพื้นที่ตามความต้องการใช้งานของโปรแกรม

2.3 Stack-Based Allocation

พื้นที่หน่วยความจำแบบ Stack จะใช้ในการเก็บข้อมูลต่าง ๆ ที่จำเป็นในการเรียกโปรแกรมย่อย โดยเมื่อมีการเรียกใช้โปรแกรมย่อยข้อมูลต่าง ๆ ที่จะถูก push ลงใน Stack และจะถูก pop ออกเมื่อการทำงานของโปรแกรมย่อยเสร็จสิ้นแล้ว โดยชุดข้อมูลของแต่ละโปรแกรมย่อยที่ push ลงใน Stack จะเรียกว่า Frame โดยในแต่ละเฟรมจะประกอบด้วยข้อมูลต่าง ๆ ดังนี้

1) *Arguments* = ค่าที่ส่งให้ฟังก์ชัน

ในบางครั้งโปรแกรมย่อยหรือฟังก์ชันจะต้องการข้อมูลนำเข้าเพื่อนำมาใช้เป็นตัวแปรในการประมวลผล โดยข้อมูลเหล่านี้จะถูกส่งให้กับโปรแกรมย่อยผ่าน Stack นั่นคือ ผู้เรียกใช้โปรแกรมย่อยจะทำการ push ค่าอาร์กิวเมนต์ลงใน Stack จากนั้น โปรแกรมย่อยจึง pop ข้อมูลเหล่านี้ไปยังตัวแปรที่ใช้ภายในโปรแกรมสำหรับการประมวลผลต่อไป

2) *Return Address*

ในการประมวลผลโปรแกรมจะเป็นการประมวลผลตามลำดับคำสั่งที่ถูกจัดเก็บไว้ในหน่วยความจำ เมื่อมีการเรียกโปรแกรมย่อย โปรแกรมก็จะกระโดดจากที่อยู่หน่วยความจำปัจจุบันที่กำลังประมวลผลอยู่ไปกระทำการกับคำสั่ง ณ จุดเริ่มต้นของโปรแกรมย่อย และเมื่อการประมวลผลของโปรแกรมย่อยเสร็จสิ้นแล้วโปรแกรมจำเป็นจะต้องกลับไปยังตำแหน่งที่จากมา นั่นคือ Return Address โดยในการจะทราบตำแหน่งที่อยู่หน่วยความจำข้อมูลนี้จะถูก push ลงใน Stack นั่นเอง

3) *Bookkeeping*

เป็นกลุ่มข้อมูลสำหรับบันทึกสถานะของการประมวลผลต่าง ๆ ก่อนการเข้าใช้ทรัพยากร ได้แก่ ข้อมูลที่จัดเก็บไว้ใน Register และค่าอ้างอิงไปยังเฟรมอื่น ๆ (reference to other frame) โดยรายละเอียดจะกล่าวในบทถัด ๆ ไป

ในส่วนของข้อมูลใน Register นั้นจำเป็นต้องมีการเก็บ เพราะเนื่องจากจำนวนของ Register มีน้อยและในการประมวลผลโดยหน่วยประมวลผลจำเป็นต้องดำเนินการกับข้อมูลใน Register เท่านั้น ดังนั้น เพื่อให้ Register สามารถนำมาใช้ประมวลผลในโปรแกรมย่อยได้ ค่า Register ซึ่งเป็นค่าสถานะสุดท้ายก่อนที่โปรแกรมย่อยจะถูกเรียกให้ทำงานจะถูกบันทึกไว้ใน Stack ส่วนของ Bookkeeping นี้ ซึ่งจะทำให้โปรแกรมย่อยสามารถเข้าใช้งาน Register ทุกตัวเพื่อประมวลผลได้อย่างอิสระ และเมื่อโปรแกรมย่อยทำงานเสร็จสิ้นแล้วก็จะคืนค่าสถานะของ Register จากข้อมูลที่ถูกบันทึกไว้ใน Stack

4) *Local variables*

ส่วนนี้จะเก็บข้อมูลของ local variable ต่าง ๆ ที่ถูกประกาศไว้ภายในโปรแกรมย่อย เพื่อให้สามารถจัดการข้อมูลของตัวแปรได้ถูกต้องตามที่ได้ประกาศไว้ในแต่ละโปรแกรมย่อย เพราะอาจจะมีการประกาศชื่อตัวแปรซ้ำกับอยู่ภายในโปรแกรมย่อยที่เรียกซ้อนกันไป

5) *Temporaries*

ในโปรแกรมย่อยอาจมีการเก็บข้อมูลที่ต้องใช้ชั่วคราว เช่น การประมวลผลที่มีความซับซ้อนซึ่งต้องการเก็บข้อมูลในส่วนของ Stack เพิ่มเติม เป็นต้น ก็จะถูกจัดเก็บไว้ในกลุ่มของข้อมูล Temporaries นี้ แต่ในบทนี้จะไม่ได้กล่าวถึงรายละเอียดข้อมูลที่จัดเก็บ

นอกจากนี้ในส่วนของการจัดสรรส่วนข้อมูล Stack ยังมีการใช้ Register 2 ตัว ดังนี้

- **Stack pointer** (sp) เป็น Register เพื่อใช้เก็บค่าที่อยู่หน่วยความจำตำแหน่งบนสุดของ Stack ซึ่งจะเปลี่ยนทุกครั้งที่มีการ push ข้อมูลลงใน Stack นั่นคือสำหรับใช้ในการบอกตำแหน่งการ push หรือ pop ข้อมูลของ Stack
- **Frame pointer** (fp) เป็น Register เพื่อเก็บค่าที่อยู่หน่วยความจำเริ่มต้นของเฟรมที่อยู่บนสุดของ Stack เรียกว่า active frame (หมายถึง โปรแกรมย่อยที่กำลังดำเนินการอยู่) ซึ่งจะใช้เป็นค่าอ้างอิงเพื่อเข้าถึงข้อมูลต่าง ๆ ภายเฟรม

ควบคู่กับค่า offset ซึ่งจะเป็นค่าบวกหรือลบซึ่งห่างจาก fp เป็นจำนวนเท่าใด โดยค่า fp จะเปลี่ยนทุกครั้งที่มีการเรียกประมวลผลโปรแกรมย่อย

2.3.1 Calling Chain and Stack Frames

พิจารณาลำดับการเรียกโปรแกรมย่อยจากตัวอย่างโปรแกรมต่อไปนี้

```
procedure C
    D; E
procedure B
    if … then B else C
procedure A
    B
-- main program
    A
```

จากตัวอย่างโค้ดเมื่อโปรแกรมเริ่มดำเนินงาน จะเริ่มจาก main program โดยเริ่มเรียก procedure A ซึ่งเป็นชื่อของโปรแกรมย่อยแบบหนึ่งซึ่งในภาษาสมัยเก่า ๆ เช่น ภาษา Pascal ใช้ procedure สำหรับโปรแกรมย่อยที่ไม่มีการคืนค่าผลลัพธ์ (return value) โดยโปรแกรมย่อยที่คืนค่าผลลัพธ์จะใช้ function เมื่อ procedure จากนั้นจึงเรียก procedure B ซ้อนไป จากนั้น B จึงเรียกตัวเองหรือเรียก procedure C ขึ้นอยู่กับผลลัพธ์ของเงื่อนไขการประมวลผล หากเมื่อ C ถูกเรียกก็จะดำเนินงานเรียก procedure D และ E จึงจะเป็นลำดับดังนี้ และหากโปรแกรมมีการกระทำการครบทุกพาธจน (เงื่อนไขภายใน procedure B เป็นจริงในนครั้งแรก) ถึงการกระทำการ ณ procedure D ซึ่งจะมีลำดับ คือ A B B C และ D โดยจะมีการจัดสรรพื้นที่ Stack สำหรับแต่ละเฟรมดังภาพ

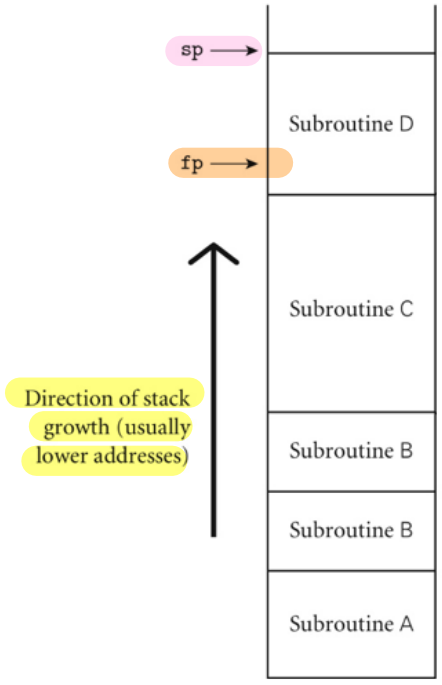
ภาพที่ 2.2 แสดงโครงสร้างหน่วยความจำ Stack ที่จัดสรรให้กับแต่ละเฟรมของโปรแกรมย่อยตามโค้ดตัวอย่าง

คำบรรยายภาพ: แสดงการวางเรียงกล่องลงใน Stack โดยแต่ละกล่องแทนเฟรมของ subroutine ด้านล่างสุดเป็นจุดเริ่มต้น Stack ลำดับของกล่องแต่ละเฟรมมีลำดับจากล่างสุดไปยังบนสุด เป็นดังนี้

1. Subroutine A
2. Subruotine B
3. Subruotine B
4. Subruotine C
5. Subruotine D

แต่ละกล่องจะมีค่าที่อยู่กำกับ ภาพแสดงทิศทางการเปลี่ยนแปลงว่าโดยทั่วไปแล้วกล่องด้านบนจะมีค่าที่อยู่น้อยกว่าด้านล่าง

ในส่วน Stack จะมี pointer 2 ตัว คือ Stack Pointer (sp) และ Frame Pointer (sp) โดยภาพ แสดงตำแหน่งที่แต่ละ pointer ชี้อยู่ ได้ sp ชี้ตำแหน่งบนสุดของ Stack เสมอ และ fp ชี้ไปที่ตำแหน่งเริ่มต้นของเฟรมบนสุดเสมอ



เมื่อมีการเรียกโปรแกรมย่อยชุดข้อมูลสำหรับการประมวลผลของโปรแกรมย่อยซึ่งเรียกว่า frame หรือ activation record ซึ่งเทียบกับกล่อง 1 กล่องในภาพที่ 2.2 จะถูก push ลงใน Stack ตามลำดับการเรียก คือ เมื่อเรียกโปรแกรมย่อย A เฟรม A จะถูก push ลงใน Stack และภายในโปรแกรมย่อย A นี้เรียกโปรแกรมย่อย B อีกก็จะเฟรม B push เพิ่มลงใน Stack ซึ่งจะเกิดขึ้นเรื่อย ๆ ไป อย่างกรณีนี้ ก็จะมี เฟรม B C D เพิ่มมาอีกตามลำดับ และจากสถานะตามภาพตำแหน่งของ sp จะชี้อยู่ที่ address ของข้อมูลสุดท้ายที่ push ลงใน stack และ fp จะชี้อยู่ที่เฟรม subroutine D ซึ่งเป็น active frame และหากทำการประมวลผลต่อไปจนถึง subroutine E ค่าใน stack ที่เปลี่ยนไปจะเป็นเฟรม D จะถูก pop ออกจาก stack และ subroutine E จะถูก push ลงใน stack fp ก็จะเปลี่ยนไปชี้ ณ ตำแหน่งของเฟรม E ซึ่งเป็น active frame ณ ขณะนั้น

ในแต่ละเฟรมจะเก็บข้อมูลต่าง ๆ ที่ใช้ในการจัดการการประมวลผลของโปรแกรมย่อยตามที่ได้กล่าวมาแล้วข้างต้น โดยโครงสร้างในการเก็บข้อมูลเหล่านั้นลง Stack จะเป็นดังภาพ

ภาพที่ 2.3 แสดงโครงสร้างการจัดสรรหน่วยความจำ Stack ในแต่ละเฟรม

คำบรรยายภาพ: แสดงภาพขยายโครงสร้างภาพในกล่อง Subroutine C อ้างอิงจากภาพที่ 2.2 โดยมีลำดับข้อมูลภายใน Stack เรียงลำดับจากล่างสุดของ Stack ดังนี้

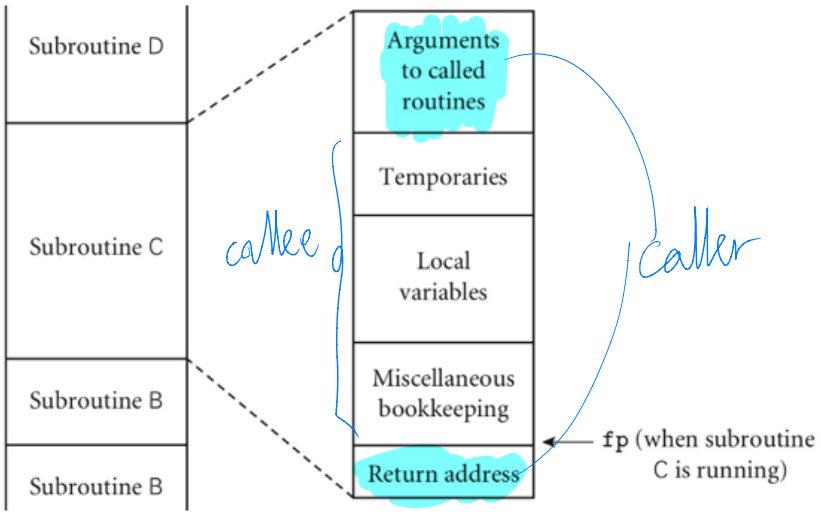
1. Return address

2. Miscellaneous bookkeeping

3. Temporaries

4. Arguments to called routines

โดยมี fp แสดงตำแหน่งภายในเฟรม กรณีทีเฟรมนั้นกำลังดำเนินงานอยู่ (running) โดยตำแหน่ง Address ตำแหน่งที่อยู่ของ Retrun address



โดยในการดำเนินการกับข้อมูลเหล่านี้ลงใน Stack จะถูกกระทำโดยผู้กระทำแบ่งออกเป็น 2 ฝั่ง คือ ผู้เรียก (Caller) และผู้ถูกเรียก (Callee) โดยผู้เรียกจะดำเนินการกับข้อมูล ได้แก่ Return address และ Arguments และผู้ถูกเรียกจะดำเนินการกับข้อมูล Bookkeeping, Local Variables และ Temporaries โดยรายละเอียดลำดับการจักการ Stack จะนำเสนอในหัวข้อถัดไป

2.3.2 Maintenance of Stack

ในการจัดการข้อมูลใน Stack จะมีการนำข้อมูลเข้าออกอยู่ตลอดเวลาที่มีการเรียกใช้งานโปรแกรมย่อย ซึ่งเป็นการจัดการข้อมูลระหว่างโปรแกรมย่อย Caller (ผู้เรียก) และ Callee (ผู้ถูกเรียก) โดยเมื่อเกิดการเรียกโปรแกรมย่อยจะเกิดลำดับขั้นตอนในการกระทำกับข้อมูลใน Stack ซึ่งถูกสร้างขึ้นโดยคอมไพเลอร์ ดังต่อไปนี้

1. Pre-call กระทำโดย Caller จะเป็นขั้นตอนของการ push ค่า argument ต่าง ๆ ที่ถูกประกาศไว้โดยโปรแกรมย่อย สำหรับส่งค่านำเข้าเพื่อประมวลผลโดยโปรแกรมย่อยที่ถูกเรียก ส่วนนี้จะเป็นส่วนของข้อมูล Arguments

2. jsr callee กระทำโดย Caller จะเป็นขั้นตอนในการสั่งให้ย้ายการประมวลผล (Jump to Subroutine) ยังตำแหน่งที่อยู่ของโปรแกรมย่อยที่ถูกเรียก พร้อมกับ push ค่า Return address ลงใน Stack

3. Prologue กระทำโดย Callee จะเป็นขั้นตอนก่อนเริ่มทำการประมวลผลของโปรแกรมย่อย ซึ่งจะเป็นการบันทึกสถานะของโปรแกรมที่เรียกไว้ ประกอบด้วยการ push ค่าปัจจุบันของรีจิสเตอร์ต่าง ๆ ที่ใช้ลงใน stack ซึ่งจะเป็นส่วนของ

bookkeeping โดยในกลุ่มค่าริจิสเตอร์ที่บันทึกจะมีค่ารีจิสเตอร์ fp รวมอยู่ด้วยจึงสามารถที่ถูกแทนด้วยค่าของเฟรมใหม่ได้ ซึ่งหลังจาก push ข้อมูล bookkeeping แล้ว ค่า fp จะถูกเปลี่ยนเป็นค่าที่อยู่หน่วยความจำของเฟรมใหม่ซึ่งเป็นของ Callee เอง จากกนั้นจึง push ข้อมูล local variables ของ callee ลงใน Stack ถ้ามีการประกาศไว้

4. Main Body กระทำโดย Callee ส่วนนี้จะเป็นส่วนของการประมวลผลตามที่ได้โปรแกรมไว้และอาจจะมีการคืนผลลัพธ์ของการประมวลผล โดยในการคืนผลลัพธ์จะมีวิธีการอยู่ 2 แบบ คือ 1) บันทึกค่าลงใน Register ที่กำหนดไว้สำหรับเก็บค่า Return Value โดยเฉพาะซึ่งเมื่อ Caller ต้องการค่าที่คืนก็จะสามารถไปเรียกดูได้ผ่านรีจิสเตอร์นี้ และ 2) ผ่าน Stack นั่นคือ Caller จะจองตำแหน่งใน Stack เอาไว้สำหรับให้ Callee นำค่าของผลลัพธ์มาบันทึกไปไว้ โดยในการเข้าถึงตำแหน่งต่าง ๆ ใน Stack จะใช้การคำนวณตำแหน่งด้วย offset ของข้อมูลนั้น ๆ เช่น หากต้องการเข้าถึงข้อมูล local variable ค่า offet จะมีค่าบวกหรือลบไปจำนวนเท่าใดจาก fp ดังนั้นเมื่อมีการกำหนดตำแหน่ง Return Value ในว่ามีค่า offset เท่าใดแล้ว หาก Callee ต้องจะบันทึกผลลัพธ์ก็จะนำไปใส่ไวยัง Stack ตำแหน่งที่ fp + offset

5. Epilogue กระทำโดย Callee จะเป็นขั้นตอนการคืนทรัพยากรที่ใช้ในการประมวลผลต่าง ๆ ได้แก่ คืนหน่วยความจำที่จัดเก็บเฟรมของตนเองใน Stack นั่นคือ pop ข้อมูลต่าง ๆ จนถึง fp ออกจาก Stack และคืนค่าต่าง ๆ ของรีจิสเตอร์ให้อยู่ในสถานะเดิมก่อนที่ถูกประมวลผลโดยโปรแกรมย่อยนี้ จากนั้นจึงสั่งโปรแกรมให้กระโดดกลับไปประมวลผลต่อยัง Return Address จากค่าใน Stack ที่ push ไว้โดย Caller จากข้อ 2

6. Post-call กระทำโดย Caller นั่นการ pop ข้อมูลต่าง ๆ ที่ส่งไปยังโปรแกรมย่อย ได้แก่ ข้อมูล Arguments เป็นต้น แล้วประมวลผลต่อจากตำแหน่ง Return Address ต่อไป

Exercise 2.1 Write sequence of stack allocation for calling add3()

จากโค้ดต่อไปนี้ จงแสดงสถานะของ Stack ในแต่ละขั้นของการเรียกฟังก์ชัน add3() ตั้งแต่เริ่มต้นจนกระทั่งการทำงานเสร็จสิ้น โดยให้อธิบายถึง ข้อมูลใน Stack ซึ่งแจกแจงรายละเอียดส่วนของ Arguments และ Local Variables และค่ารีจิสเตอร์ sp และ fp โดยกำหนดให้ค่าตำแหน่งเริ่มต้นของ sp เท่ากับ 0 และค่าเพิ่มขึ้นเมื่อมีการ push ข้อมูลลงใน Stack และใช้การคืนค่า Return Value ให้ดำเนินการผ่าน Stack และสมมติขนาดของข้อมูลที่เป็นกลุ่มที่ไม่ได้แจกแจงได้แก่ bookkeeping และ temporaries เท่ากับ 1 หน่วยของ Stack (ขนาดจริงแล้วแต่กำหนดโดย compiler อาจเป็น byte, word หรือ double word)

```
int add2 (int a, int b) {
        return a + b;
}
int add3 (int a, int b, int c) {
        int res;
        res = add2(a,b);
        return res + c;
}
```

```
int sum = 0;
int main() {
        sum += add3(1, 2, 3);
        return 0;
}
```

เฉลย Exercise 2.1

จากข้อความสั่งเพื่อการเรียกประมวลผลฟังก์ชัน add3 ได้เกิดขึ้นในฟังก์ชัน main และในฟังก์ชัน add3 ก็มีการเรียกใช้ฟังก์ชัน add2 ดั้งนั้น เฟรมจะถูก push ลงใน stack ตามลำดับ คือ เฟรมของ main add3 และ add2 โดยสถานะของ Stack แต่ละขั้นตอนจะเป็นดังนี้

1. **Pre-call และ JSR ฟังก์ชัน add3 โดยฟังก์ชัน main**

   ข้อความสั่งเพื่อเรียกฟังก์ชัน add3 เป็นการเรียกฟังก์ชัน add3 เพื่อบวกตัวเลข 3 จำนวน และนำผลลัพธ์ไปบวกเพิ่มไว้กับตัวแปรชื่อ sum ซึ่งเป็น global variable

   กำหนดให้การอธิบาย stack contents จะเป็นลำดับข้อมูล โดยจะเรียงข้อมูลที่ push ก่อนจากด้านซ้ายไปขวา แต่ละข้อมูลคั่นด้วยเครื่องหมายจุลภาค (commas) และแต่ละข้อมูลมีรูปแบบการเขียนอธิบาย คือ

   คำอธิบายข้อมูล = ค่าของข้อมูล

   โดยแต่ละส่วนมีความหมาย ดังนี้

   - คำอธิบายข้อมูล คือ คำอธิบายข้อมูลหรือชื่อตัวแปรของข้อมูล
   - ค่าของข้อมูล คือ ตัวเลขหรือตัวอักษรแสดงถึงค่าที่จัดเก็บจริงใน Stack ละไว้ได้กรณีที่ไม่ทราบข้อมูลจริงที่เก็บ (ข้อมูลชนิด int กำหนดค่าปริยาย (default) เป็น 0)

   ตัวอย่างเช่น เมื่อ push ค่าอาร์กิวเมนต์ a ซึ่งมีค่าเท่ากับ 1 ลงใน stack จะเขียนเป็น a = 1 หรือ ค่า return address ของฟังก์ชัน main จะเขียนโดยละค่าของข้อมูลไว้เพราะไม่ทราบค่าจริงเป็น main’s return address เป็นต้น

   และการระบุตำแหน่งของพอยน์เตอร์จะเขียนโดยใช้ปีกกาครอบชื่อ pointer คือ {sp} และ {fp} โดยจะวางไว้ด้านหน้าหรือหลังข้อมูลใน stack โดยหากวางไว้ด้านหน้าหมายถึงค่าตำแหน่งข้อมูลลบไป 1 หน่วย และด้านหลังมีค่าเท่ากับตำแหน่งของข้อมูล ตัวอย่างเช่น

   ```
   Stack contents: {fp} main’s return value {sp}
   ```

   จากค่าของข้อมูลข้างต้นอ่านค่าของ {fp} ได้เท่ากับ 0 และค่าของ {sp} ได้เท่ากับ 1

   โดยสถานะของ Stack ก่อนการ pre-call (contents ของ stack) เป็นดังนี้

   ```
   sp: 0
   ```

```
fp: 0
Stack contents is empty:
```

ฟังก์ชัน main ทำการ push ข้อมูล ตามลำดับดังนี้

- Return Value
- Arguments ที่ต้องในการเรียกฟังก์ชัน add3 ซึ่งกำหนดไว้คือ a, b, c โดยลำดับการ push จะเริ่มจากขวามาซ้าย ดังนั้นจะทำการ push ค่าได้แก่ c = 3, b = 2 และ a = 1 ตามลำดับ
- Return Address

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นดังนี้

```
sp: 5
fp: 0
Stack contents:
{fp} main's return value, c = 3, b = 2, a = 1, main's return address {sp}
```

2. Prologue ของฟังก์ชัน add3

ก่อนการประมวลผลของโปรแกรมย่อยการ push ข้อมูลต่าง ๆ ดังนี้ ลงใน Stack ดังนี้

- Bookkeeping สถานะเดิมของฟังก์ชัน main
- Local vairables ได้แก่ ตัวแปรชื่อ res
- Temporaries ของฟังก์ชัน add3 โดยสมมติว่ามีการใช้งานโดยฟังก์ชันนี้เท่านั้น

เพิ่ม 3

และทำการปรับปรุง ค่า fp เพื่อไปยัง active frame คือ add3 นั่นคือค่า sp ก่อนที่จะมีการ push ข้อมูลโดยฟังก์ชัน add3

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นดังนี้

```
sp: 8
fp: 5
Stack contents:
main's return value = 0, c = 3, b = 2, a = 1, main's return address {fp},
main's bookkeeping, res = 0, temporaries {sp}
```

frame of add3

3. Pre-call และ JSR ฟังก์ชัน add2 โดยฟังก์ชัน add3

ฟังก์ชัน add3 มีการเรียกฟังก์ชัน add2 โดยมีอาร์กิวเมนต์ 2 ตัว คือ a, b แล้วนำผลลัพธ์มาเก็บไว้ยังตัวแปร res โดยฟังก์ชัน add3 ทำการ push ข้อมูล ตามลำดับดังนี้

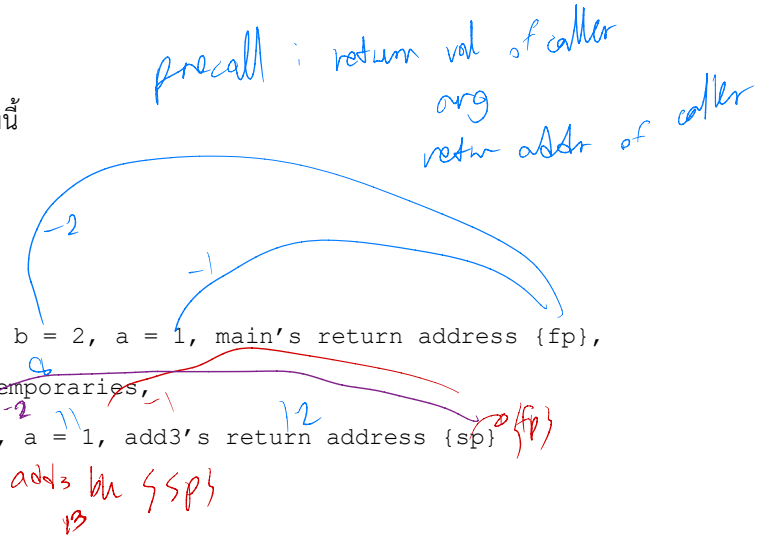
1 - Return Value

2 - Arguments ที่ต้องในการเรียกฟังก์ชัน add2 ซึ่งกำหนดไว้คือ a, b โดยลำดับการ push จะเริ่มจากขวามาซ้าย ดังนั้นจะทำการ push ค่าซึ่งตามโค้ดแล้วเป็นการส่งค่าตัวแปร a และ b ที่เป็นอาร์กิวเมนต์ของฟังก์ชันนี้ ดังนั้นจึงต้องไปนำค่าจาก stack ที่ offset ของ a เท่ากับ -1 และ b = -2 จึงได้ค่าที่จะ push คือ b = 2 และ a = 1 ตามลำดับ

1 - Return Address

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นดังนี้

```
sp: 12
fp: 5
Stack contents:
main's return value= 0, c = 3, b = 2, a = 1, main's return address {fp},
main's bookkeeping, res = 0, temporaries,
add3's return value = 0, b = 2, a = 1, add3's return address {sp}
```



4. Prologue ของฟังก์ชัน add2

ก่อนการประมวลผลของโปรแกรมย่อยการ push ข้อมูลต่าง ๆ ดังนี้ ลงใน Stack ดังนี้

- Bookkeeping สถานะเดิมของฟังก์ชัน add3
- Local vairables ไม่มีการนี้ยังจึงไม่ข้อมูลใน stack
- Temporaries ของฟังก์ชัน add2

```
int add2 (int a, int b) {
        return a + b;
}
```

และทำการปรับปรุง ค่า fp เพื่อไปยัง active frame

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นดังนี้

```
sp: 13
fp: 12
Stack contents:
main's return value = 0, c = 3, b = 2, a = 1, main's return address,
main's bookkeeping, res = 0, temporaries,
add3's return value = 0, b = 2, a = 1, add3's return address {fp},
add3's bookkeeping {sp}
```

5. **Return ของฟังก์ชัน add2**

ในฟังก์ชัน add2 มีการคืนค่าผลลัพธ์จากข้อความสั่ง return a + b; นั่น คือ ผลบวกของค่าอาร์กิวเมนต์ a และ b ซึ่งคือข้อมูลใน Stack ที่ offset ที่ -1 และ -2 ซึ่งหมายถึง 11 และ 10 ตามลำดับ และเมื่อดูค่าใน Stack จะได้แก่ ค่าตัวแปร a คือ 1 และค่าตัวแปร b คือ 2 เมื่อบวกกันแล้วผลลัพธ์ได้เท่ากับ 3

ในการคืนค่า ผลลัพธ์ที่ได้จะถูกนำไปปรับปรุงค่าใน Stack ณ offset ที่ -3 (ค่าลบของจำนวนอาร์กิวเมนต์บวกด้วยหนึ่ง – 1 * (number of arguments + 1)) คือ 12 – 3 เท่ากับ 9

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นดังนี้

```
sp: 13
fp: 12
Stack contents:
main's return value = 0, c = 3, b = 2, a = 1, main's return address,
main's bookkeeping, res = 0, temporaries,
add3's return value = 3, b = 2, a = 1, add3's return address {fp},
add3's bookkeeping {sp}
```

หากพิจารณา stack ในขณะนี้ จะบรรจุเฟรมของฟังก์ชันทั้งหมด 3 เฟรม ได้แก่

- main คือ ตำแหน่งที่ 1 – 4
- add3 คือ ตำแหน่งที่ 5 – 11
- add2 คือ ตำแหน่งที่ 12 – 14

6. **Epilogue ของฟังก์ชัน add2**

ฟังก์ชัน add2 จะทำการคืนค่ารีจิสเตอร์จากค่าที่บันทึกไว้ใน Stack ซึ่งรวมถึง ค่า fp ด้วย และ pop เฟรมของตนเองออกจาก stack

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นดังนี้

```
sp: 12
fp: 5
Stack contents:
main's return value = 0, c = 3, b = 2, a = 1, main's return address {fp},
main's bookkeeping, res = 0, temporaries,
add3's return value = 3, b = 2, a = 1, add3's return address {sp}
```

7. Post-call ของฟังก์ชัน add3

ฟังก์ชัน add3 จะทำการประมวลผลกับ Return value ซึ่งมีค่าเท่ากับ 3 ก่อน นั่นคือนำไปปรับปรุงให้กับตัวแปรชื่อ res ใน stack จากนั้น จึง pop ข้อมูลที่เกี่ยวกับการเรียกฟังก์ชัน add2 ออกจาก stack ได้แก่ Return address Argurments และ Return value

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นดังนี้

```
sp: 8
fp: 5
Stack contents:
main's return value = 0, c = 3, b = 2, a = 1, main's return address {fp},
main's bookkeeping, res = 3, temporaries {sp}
```

8. Return ของฟังก์ชัน add3

ในฟังก์ชัน add3 มีการคืนค่าผลลัพธ์จากข้อความสั่ง return res + c; นั่น คือ ผลบวกของค่าตัวแปร res (local variable) คือค่าใน stack ที่ offset เท่ากับ 872 หน่วย คือ ตำแหน่งที่ 7 ซึ่งขณะนี้มีค่าเท่ากับ 3 และค่าอาร์กิวเมนต์ c ซึ่งคือข้อมูลใน Stack ที่ offset ที่ -3 คือ ตำแหน่งที่ 7 ซึ่งมีค่าเท่ากับ 3 เมื่อคำนวณแล้วจะได้ผลลัพธ์เท่ากับ 6 และนำไปปรับปรุงค่าReturn value ใน Stack ณ offset ที่ -4 คือ ตำ[uแหน่งที่ 1

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นดังนี้

```
sp: 8
fp: 5
Stack contents:
main's return value = 6, c = 3, b = 2, a = 1, main's return address {fp},
main's bookkeeping, res = 3, temporaries {sp}
```

9. Epilogue ของฟังก์ชัน add3

ฟังก์ชัน add3 จะทำการคืนค่ารีจิสเตอร์จากค่าที่บันทึกไว้ใน Stack ซึ่งรวมถึง ค่า fp ด้วย และ pop เฟรมของตนเองออกจาก stack

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นดังนี้

```
sp: 5
fp: 0
Stack contents:
{fp} main's return value = 6, c = 3, b = 2, a = 1, (main's return address) {sp}
```

10. Post-call ของฟังก์ชัน main

ฟังก์ชัน main นำค่า return value ซึ่งมีค่าเท่ากับ 6 ไปบวกเพิ่มให้กับตัวแปรชื่อ sum ซึ่งเป็นตัวแปรแบบ global อย่างที่ทราบมาแล้วข้างต้นว่าส่วนของตัวแปรที่มีอายุตลอดโปรแกรมจะถูกจัดสรรหน่วยความจำไว้ที่ data segment สมมติว่าตัวแปร sum ถูก bind ไว้ที่อยู่หน่วยความจำที่ 200 ค่าข้อมูลในหน่วยความซึ่งถูกจองไว้ จะมีค่าเป็น

```
Data Segment 200: sum = 0
```

และเมื่อนำค่า 6 ก็ไปบวกเพิ่มค่าในหน่วยความจำที่ตำแหน่ง 200 จะได้ผลลัพธ์เป็น

```
Data Segment 200: sum = 6
```

จากนั้น จึง pop ข้อมูลที่เกี่ยวกับการเรียกฟังก์ชัน add3 ออกจาก stack ได้แก่ Return address Argurments และ Return value

ดังนั้น สถานะของ Stack จะเปลี่ยนไปเป็นเหมือนตอนก่อนเรียกฟังก์ชัน add3 ดังนี้

```
sp: 0
fp: 0
Stack contents is empty:
```

เมื่อพิจาณาจากสถานะของแต่ละครั้งในการเรียกโปรแกรมย่อยหรือฟังก์ชันการเปลี่ยนแปลงข้อมูลภายใน stack เกิดขึ้นในช่วงของ run-time ซึ่งปริมาณของข้อมูลที่ push ลงใน stack จะแตกต่างกันออกไปตามแต่จำนวนของตัวแปรที่ใช้ และในแต่ละช่วงเวลาขนาดของ Stack ก็จะแปรผันไปตามระดับความลึกของการเรียกฟังก์ชันยังหลายชั้นขนาดก็จะยาวมากตามไปด้วย

นอกจากนี้ใน Execise ไม่ได้แสดงให้เห็นว่าฟังก์ชัน main ถูกเรียกได้อย่างไร แต่ถ้าหากเข้าใจกลไกการเรียกโปรแกรมย่อย ก็จะเข้าใจได้ว่าฟังก์ชัน main ก็จะถูกมองว่าเป็นโปรแกรมย่อยของ platform ที่โปรแกรมนี้ทำงานอยู่ เช่น ระบบปฏิบัติการ เป็นต้น ซึ่งก็จะเปรียบเหมือนผู้เรียกที่จะมีขั้นตอนการดำเนินการกับข้อมูลใน stack ในแบบเดียวกัน เช่น push อาร์กิวเมนต์ของโปรแกรมเป็นต้น

จบการเฉลย Exercise 2.1

2.4 Heap-Based Allocation

ในการจัดสรรหน่วยความจำในส่วนของ Heap จะถูกจัดการโดยคำสั่งของโปรแกรมเมอร์ นั่นคือ โปรแกรมเมอร์จะต้องเขียนกำหนดไว้ในโปรแกรมเองว่าต้องการใช้หน่วยความจำสำหรับไว้จัดเก็บสิ่งใดบ้าง ซึ่งจะต่างจากหน่วยความจำส่วน Stack ที่จะถูกดำเนินการจัดสรรอย่างอัตโนมัติโดย Compiler

การจะจัดสรรหน่วยความจำส่วน Heap จะต้องมีการเขียนคำสั่งไว้ทุกครั้ง เช่น คำสั่ง new ของภาษา Java หรือคำสั่ง malloc ของภาษาซี เป็นต้น ต่อไปนี้เป็นโค้ดตัวอย่างการจอง heap ของภาษาซี

```
int *p = (int*) malloc(sizeof(int))
```

จากโค้ดเป็นข้อความสั่งเพื่อประกาศ pointer ชื่อ p ซึ่งอ้างอิงไปยังที่อยู่หน่วยความจำหนึ่งซึ่งถูกจัดสรรหรือจองไว้สำหรับจัดเก็บข้อมูลชนิด int ซึ่งมีขนาด 16 บิทหรือ 2 ไบต์สำหรับภาษาซี โดยในกรณีนี้จะมีการใช้หน่วยความจำ 2 ส่วน คือ ส่วน Stack จะมีการ push local variable ชื่อ p ซึ่งเป็นชนิด pointer ซึ่งจำจัดเก็บค่าของข้อมูลเป็นที่อยู่ของข้อมูลจริง ๆ ซึ่ง จะถูกจองไว้ในส่วน Heap ซึ่งจัดเก็บข้อมูลขนาด 2 ไบต์ไว้

นอกจากที่โปรแกรมเมอร์จะต้องเขียนกำหนดการจองใช้เนื้อที่หน่วยความจำเองแล้ว ในบางภาษาก็มีความจำเป็นที่โปรแกรมเมอร์ดต้องเป็นผู้สั่งให้คืนเนื้อที่ที่จองไว้ (deallocate) เองด้วย อย่างเช่น คำสั่ง delete ของภาษา C++ หรือ free ของภาษาซี เป็นต้น เพราะหน่วยความจำมีขนาดที่จำกัด การคืนเนื้อที่ของหน่วยความจำก็เพื่อให้สามารถนำกลับไปใช้จัดเก็บข้อมูลในการประมวลผลอย่างอื่นต่อไปได้

2.4.1 Heap Storage Management

ในการจัดการ Heap Storage ก็จะมีทั้งการจองและคืนสลับสับเปลี่ยนหมุนเวียนกันไปตลอดการทำงานของโปรแกรม ในการจองแต่ละครั้งส่วนของเนื้อที่ว่างที่ต่อเนื่องกันของหน่วยที่มีขนาดมากกว่าหรือเท่ากับขนาดของข้อมูลที่ต้องการจองจะถูกจัดสรรให้ตามความต้องการของโปรแกรม เมื่อมีการจองและคืนก็จะเกิดปัญหาหนึ่งที่ต้องจัดการ หากพิจารณาจากลำดับการจองและคืนต่อไปนี้

โปรแกรมหนึ่งมีเนื้อที่หน่วยความจำ Heap ขนาด 10 ไบต์ โดยมีลำดับการจองและคืน ดังนี้

- จองหน่วยความจำขนาด 2 ไบต์ ที่ตำแหน่งที่ 1 - 2
- จองหน่วยความจำขนาด 2 ไบต์ ที่ตำแหน่งที่ 3 - 4
- จองหน่วยความจำขนาด 3 ไบต์ ที่ตำแหน่งที่ 5 – 7
- จองหน่วยความจำขนาด 2 ไบต์ ที่ตำแหน่งที่ 8 – 9 (มีการใช้งานจริง 1 ไบต์)

สถานะปัจจุบันเนื้อที่หน่วยความจำถูกจองด้วยขนาด 2, 2, 3 และ 2 ไบต์ตามลำดับ ต่อมาเมื่อมีการคืนเนื้อที่หน่วยความจำก้อนที่ 2 ทำให้มีเนื้อที่หน่วยความจำว่างที่ตำแหน่งที่ 3, 4 และ 10 ตามลำดับ

ต่อมาหากมีความต้องการจองเนื้อที่ขนาด 3 ไบต์ สถานะหน่วยความจำในขณะนี้จะไม่สามารถรองรับการจองเนื้อที่ได้ แม้ว่าจะมีเนื้อที่ว่างจำนวน 3 ไบต์ แต่เนื้อที่ไม่ได้ว่างต่อเนื่องกัน ปัญหาที่เกิดขึ้นนี้เรียกว่า fragmentation ซึ่งเป็นการอธิบายได้ถึงเนื้อที่ว่างของหน่วยความจำที่แตกกระจายเป็นส่วยย่อยจนไม่สามารถถูกจองเพื่อใช้งานได้ โดย fragmentation จะสามารถมีได้ 2 แบบ คือ

2.4.1.1 External fragmentation

เป็นเนื้อที่ว่างของหน่วยความจำที่จะสามารถถูกจองเข้าใช้งานได้โดยโปรแกรม แต่เนื้อที่เหล่านั้นไม่สามารถถูกจองได้เพราะขนาดไม่เพียงพอสำหรับการจอง เมื่อเทียบกับตัวอย่างข้างต้น คือเนื้อที่หน่วยความจำที่ตำแหน่ง 3, 4 และ 10 ซึ่งไม่สามารถรองรับการจองขนาด 3 ไบต์

ปัญหานี้จัดการได้โดยวิธีการที่เรียกว่า compaction คือ การจัดเนื้อที่ใหม่โดยการย้ายส่วนเนื้อที่ว่างให้มาติดกัน เช่น การขยับส่วนที่ถูกจอง ใช้ชิดไปทางด้านซ้าย และเมื่อทำกับตัวอย่างข้างต้นหน่วยความจำก็จะเกิดว่างที่ตำแหน่ง 8-10 ซึ่งจะสามารถรองรับการจองหน่วยความจำขนาด 3 ไบต์ได้นั่นเอง

2.4.1.2 Internal fragmentation

จะเป็นส่วนเนื้อที่ว่างของหน่วยความจำที่ถูกจองแล้วแต่ไม่ได้ถูกใช้งานเพื่อเก็บข้อมูล ซึ่งเนื้อที่ส่วนนี้โปรแกรมเองจะมองว่าเป็นเนื้อที่ถูกจองเพื่อใช้งานแล้วซึ่งจะนำมาใช้งานอื่นไม่ได้จนกว่าจะถูกคืน ซึ่งจากตัวอย่างข้างต้นส่วนของ internal fragment จะอยู่ที่การจองที่ตำแหน่ง 8-9 ซึ่งมีการใช้หน่วยความจำน้อยกว่าที่ได้จองเอาไว้ ซึ่งอาจเกิดจากหน่วยความจำที่ต้องการใช้มีขนาดเล็กกว่าค่า minimum threshold ที่ถูกตั้งเอาไว้ในที่นี้ คือ 2 ไบต์ นั่นเอง ผลกระทบของ fragmentation แบบนี้คือ เสียเนื้อที่ว่างไปโดยไม่จำเป็นนั้นเอง

สำหรับในบางระบบปฏิบัติการจะไม่ทำการจัดสรรหน่วยความจำตรงตามขนาดที่โปรแกรมร้องขอ แต่จะกำหนดเป็นขนาดบล็อกมาตรฐาน (standard block size) เช่น กำหนดขนาดบล็อก เป็น 4, 8 หรือ 16 กิโลไบต์ เป็นต้น เพื่อเป็นตัววัดว่าจะจองเท่าไหร่เป็นหน่วยบล็อก ตัวอย่างเช่น เมื่อกำหนดบล็อกให้มีขนาด 4 KB เมื่อต้องการใช้เนื้อที่หน่วยความจำขนาด 3 KB ก็ต้องจองทั้งบล็อก คือ 4 KB หรือหากต้องการการใช้ 6 KB ก็ต้องจอง 2 บล็อกที่ขนาด 8 KB วิธีการนี้ทำให้ง่ายต่อการจัดการ fragmentation เพราะเมื่อหน่วยความจำถูกคืนทำให้มั่นใจได้ว่าจะยังถูกนำไปใช้งานได้เพราะทำกันเป็นบล็อก ๆ ขณะเดียวกันก็ต้องแลกกับโอกาสที่จะเกิด internal fragment เพราะความต้องการขอจองจะไม่ได้มีขนาดเท่ากับบล็อกพอดีเสมอไป นั่นเอง

2.4.2 Problems with Manual Deallocation

ในบางภาษาโปรแกรมกำหนดให้โปรแกรมเมอร์เป็นผู้จองและคืนเนื้อที่หน่วยความจำส่วน Heap เอง และเมื่อกิจกรรมใด ๆ ที่กระทำโดยผู้ใช้มักจะเกิดข้อผิดพลาด (error) ได้ โดยในการคืนหน่วยความจำก็จะมีปัญหาเช่นกัน ดังนี้

2.4.2.1 Dangling reference

เป็นการอ้างอิงส่วนของหน่วยความจำโดยตัวแปร ซึ่งหน่วยความจำนั้นถูกคืนไปแล้ว พิจารณาจากโค้ดภาษา C++ ดังต่อไปนี้

```
int *x = new int;       int *y ;
*x = 2;         y = x;
delete x;
*y = 3;
```

จากโค้ดเมื่อโปรแกรมทำงานถึงบรรทัดสุดท้ายก็จะเกิดเป็น segmentation fault อธิบายได้ ดังนี้

มีการประกาศตัวแปรชนิด pointer 2 ตัว ชื่อ x และ y โดยกำหนดค่าเริ่มต้นของตัวแปร x ให้เก็บที่อยู่หน่วยความจำ heap ซึ่งจองไว้สำหรับเก็บค่าชนิด int สมมติว่าถูกจองไว้ที่ตำแหน่ง 200

จากนั้นมีการ assign ค่า 2 ให้เก็บยังหน่วยความจำที่ตัวแปร x อ้างถึงอยู่ จากนั้นจึงกำหนดให้ตัวแปร y เก็บค่าที่อยู่หน่วยความจำเดียวกับ x นั่นคือ ตอนนี้ทั้งตัวแปร x และตัวแปร y เก็บค่า 200 เอาไว้

บรรทัดต่อมาคือ delete x; เป็นคำสั่งคืนหน่วยความที่ x อ้างถึงอยู่นั่น คือ ตำแหน่งที่ 200 ทำให้หน่วยความจำถูกคืนไป และล้างค่าที่อยู่หน่วยความจำของ x ไป

บรรทัดถัดมามีความพยายามที่จะเขียนค่า 3 ไปยังตำแหน่งหน่วยความจำที่ y อ้างถึงคือ ตำแหน่งที่ 200 อาจทำให้เกิดปัญหาตามมาได้เพราะเนื้อที่หน่วยความจำนี้ถูกคืนไปแล้วนั่นหมายถึงว่ามีโอกาสที่จะถูกจองไปใช้กับตัวแปรอื่นเพื่อประมวลผลอย่างอื่น ถ้าโชคดีโปรแกรมก็จะสามารถทำงานได้ตามปกติเพราะการอ่านเขียนหน่วยความจำส่วนนี้ยังไม่ถูกดำเนินการผ่านตัวแปรอื่น แต่ในทางกลับกันก็อาจเขียนค่าทับลงบนข้อมูลที่ถูกจองไว้แล้วและถูกเก็บข้อมูลบางอย่างไว้แล้วก็อาจทำให้ข้อมูลเกิดความเสียหรือการประมวลผลผิดพลาดได้ ซึ่งทำให้ debug ยาก เนื่องความผิดพลาดไปปรากฏที่อื่นซึ่งไม่ได้เกี่ยวข้องกันโค้ดที่เป็นต้นเหตุ หรืออาจเกิด segmetation fault (เป็นการอ้างถึงหน่วยความจำที่ไม่ valid แล้ว) ซึ่งเป็นปัญหาที่ยากจะจัดการสำหรับโปรแกรมเมอร์

2.4.2.2 Memory leak

เป็นการรั่วของหน่วยความจำหมายถึงมีส่วนของหน่วยความจำที่สูญเสียการควบคุมไปไม่สามารถจะถูกจัดการได้อีก นั่นคือ ไม่สามารถนำมาจองและคืนได้อีก จะเกิดกับส่วนของหน่วยความจำที่ถูกจองแล้วไม่มีตัวแปรใดอ้างถึงโดยที่ยังไม่ได้ทำการคืน พิจารณาจากโค้ดต่อไปนี้

```
int *x = new int;  int *y  = new int;
*x = 2;       *y = 3;
x = y;
```

จากโค้ดเมื่อโปรแกรมทำงานจะสามารถอธิบายได้ ดังนี้

มีการประกาศตัวแปรชนิด pointer 2 ตัว ชื่อ x และ y ให้ทั้งคู่เก็บที่อยู่หน่วยความจำ heap ซึ่งจองไว้สำหรับเก็บค่าชนิด int สมมติว่าถูกจองไว้ที่ตำแหน่ง 200 และ 201 ตามลำดับ

จากนั้นมีการ assign ค่า 2 ให้เก็บยังหน่วยความจำที่ตัวแปร x อ้างถึงอยู่ และ assign ค่า 3 ให้เก็บยังหน่วยความจำที่ตัวแปร y อ้างถึงอยู่

ในบรรทัดสุดท้ายมีการเปลี่ยนค่าที่อยู่ที่ตัวแปร x อ้างถึงให้ไปอ้างที่เดียวกับตัวแปร y นั่นคือ ตัวแปรทั้ง 2 อ้างไปยังตำแหน่งที่อยู่ที่ 201 ทำให้ตอนนี้ หน่วยความจำตำแหน่งที่ 200 ซึ่งเก็บค่า 2 เอาไว้ไม่มีตัวแปรอ้างถึงซึ่งเกิดการรั่วขึ้น นั่นคือ หน่วยความจำตำแหน่ง 200 จะไม่สามารถนำกลับมาใช้ได้อีกแล้วเพราะยังไม่ได้คืน และคืนไม่ได้เพราะการคืนต้องสั่งผ่านตัวแปรที่อ้างถึงที่ไม่มีเหลืออยู่แล้ว และเมื่อเกิดหน่วยความจำชนิดนี้จำนวนมาก เช่น โค้ดชุดนี้ถูกทำวนลูปซ้ำ ๆ หน่วยความจำที่จะสามารถใช้งานได้จะเหลือน้อยลงเรื่อย ๆ จนไม่เหลือให้ใช้งานได้อีกจะเกิด error เป็น out of memory

จากปัญหาต่าง ๆ ที่กล่าวมาแล้วข้างต้น เกิดจากการที่ภาษาโปรแกรมปล่อยให้การจัดการหน่วยความจำเป็นหน้าที่ของโปรแกรมเมอร์ ซึ่งมีโอกาสที่จะเกิด error ขึ้นได้ นอกจากนี้ยังทำให้โปรแกรมที่เขียนมีความซับซ้อนอีกด้วย

Exercise 2.2 Heap-Based Objects and Binding

เติมช่องว่าง (.....) เพื่อให้ประโยคถูกต้องจากโจทย์ดังต่อไปนี้

Since lifetime means the time between creation and destruction,

Binding lifetime is .....

Object lifetime is .....

If object lifetime is longer than binding lifetime, we have .....

If binding lifetime is longer than object lifetime, we have .....

เฉลย Exercise 2.2

Binding lifetime is time between creation and destruction of name-to-object binding.

Object lifetime is time between creation and destruction of object.

If object lifetime is longer than binding lifetime, we have memory leak.

If binding lifetime is longer than object lifetime, we have dangling reference.

อธิบายเพิ่มเติม object คือ ส่วนความจำส่วน Heap ที่ถูกจองแล้ว binding คือ ส่วนของ variable ใน Stack หรือ data segement.

จบการเฉลย Exercise 2.2

2.4.3 Garbage Collection

เนื่องจากปัญหาที่เกิดจากการปล่อยให้โปรแกรมจัดการในการคืนเนื้อที่หน่วยความจำที่ถูกจองไว้เอง ในภาษาโปรแกรมใหม่ ๆ จึงได้มีอัลกอริทึมในการจัดการคืนหน่วยความจำไว้ในเบื้องหลังเพื่อตรวจสอบและคืนหน่วยความจำที่ไม่ถูกอ้างถึงโดยตัวแปรใด ๆ แล้ว เรียกว่า garbage collection ซึ่ง feature นี้ได้ช่วยลดภาระในการเขียนโปรแกรมทำให้การเขียนโปรแกรมสะดวกขึ้นที่จะไม่ต้องกังวลถึงการคืนหน่วยความจำว่าทำได้ถูกต้องครบถ้วนหรือไม่ เพียงแต่ปล่อยให้การคืนถูกจัดการโดยอัตโนมัติ

วิธีการนี้มีข้อดี คือ ลดข้อผิดพลาดจากการจัดการเนื้อที่หน่วยความจำในการเขียนโปรแกรม

และมีข้อเสีย คือ ความซับซ้อนใน implementation ของภาษาโปรแกรม ซึ่งต้องเพิ่มอัลกอริทึมส่วนของ garbage collection เพิ่ม และในรุ่นแรกอาจจะมีประเด็นเรื่องประสิทธิภาพการทำงานเพราะอัลกอริทึมนี้จะทำงานอยู่เบื้องหลัง แต่ในปัจจุบันความซับซ้อนในการจัดการหน่วยความจำระหว่างมีกับไม่มี garbage collection ไม่ได้มีความแตกต่างกันมากนักแล้ว และตัวอัลกอริทึมเองก็ได้รับการปรับปรุงได้ดีขึ้นอย่างต่อเนื่อง ดังนั้น ประเด็นด้านประสิทธิภาพจึงมีผลเสียน้อยลงไปด้วย

2.5 Scopes

คำศัพท์ที่เกี่ยวข้องกับ Scope ได้แก่

- **Scope of binding**

คือ บริเวณในโปรแกรมที่ binding นั้นใช้งานได้ เช่น เมื่อตั้งชื่อตัวแปรตัวหนึ่ง ซึ่งถูกยึดเหนี่ยวไว้กับเนื้อที่หน่วยความจำหนึ่ง ๆ ส่วน binding ที่ใช้งานได้หรือ active ในส่วนใดของโปรแกรมบ้างที่ชื่อที่ตั้งขึ้นนี้จะยังอ้างไปยังหน่วยความจำนั้น ๆ อยู่

- Scope

คือ ขอบเขตหรืออาณาบริเวณของโปรแกรม (program region) ที่ขนาดใหญ่ที่สุดที่ binding ไม่มีการเปลี่ยนแปลง เช่น ภายในขอบเขตของ block { } หรือการประกาศคลาสหนึ่ง ชื่อต่าง ๆ ที่ถูก declare อยู่ภายใต้คลาสนั้นถือว่าอยู่ภายใต้ scope ของคลาสเดียวกัน

- Referencing environment

เป็นเซตของ binding ที่สามารถเข้าถึงและใช้งานได้ ณ ตำแหน่งใดตำแหน่งหนึ่งของโปรแกรม

Exercise 2.3 Scope and Referencing Environment

จาก code ตัวอย่างโปรแกรมภาษาซีต่อไปนี้

```
float op1(int x, float y) {
    int z;
    …
}
float op2(int z) { … }
```

พิจารณาจาก scope ของ op1 และ op2 ให้เติมช่องว่าง (.....) เพื่อให้ประโยคต่อไปนี้สมบูรณ์

Scope of x and y is .....

Scope of z is .....

Referencing environment of op1 consists of .....

Referencing environment of op2 consists of .....

เฉลย Exercise 2.3

Scope of x and y is op1.

Scope of z is op1 and op2.

Referencing environment of op1 consists of x, y, z, op1 and op2.

Referencing environment of op2 consists of z, op1 and op2.

คำอธิบายเฉลย

op1 มี 2 argument คือ x และ y มี local variable ตัวหนึ่ง คือ z

op2 มี argument คือ z

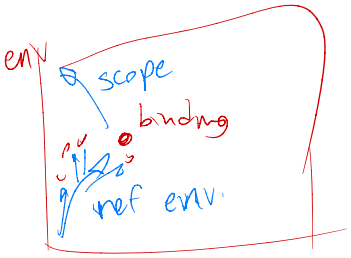
เมื่อพิจารณา op1 และ op2 เป็น scope ภายใน op1 ก็มีชุดของ binding ที่ active สามารถเรียกใช้ได้ภายใต้บล็อกของ op1 และเมื่อย้ายไปที่บล็อก op2 ชุดของ binding ที่เรียกได้ก็จะเปลี่ยนไป

เมื่อมีการเรียกฟังก์ชัน op1 ข้อมูลใน stack เป็นดังนี้

```
Stack contents: y, x, return address, bookkeeping, z
```

เมื่อมีการเรียกฟังก์ชัน op2 ข้อมูลใน stack เป็นดังนี้

```
Stack contents: z, return address, bookkeeping
```



อันที่จริงแล้ว scope กับ referencing environment เป็นเรื่องเดียวแต่ต่างกันที่มุมมอง นั่น scope มองที่ binding ไปหาขอบเขตของโปรแกรมว่าสามารถถูกใช้งานได้ ณ ตำแหน่งใด ขณะที่ referencing environment จะเริ่มจากตำแหน่งในโปรแกรมไปหา binding ว่าจะสามารถใช้งานตัวไหนได้บ้าง

จบการเฉลย Exercise 2.3

2.5.1 Static Scoping (or Lexical Scoping)

ภาษาโปรแกรมในปัจจุบันส่วนมากจะเป็นแบบ static scoping หรือ lexical scoping หมายถึง การพิจารณา scope ตามข้อความที่เขียนในโปรแกรม ว่ามีกำหนด scope ของแต่ละ binding อย่างไร โดยไม่สนใจถึงลำดับการทำงานที่จะเกิดขึ้นในระหว่างโปรแกรมทำงานจริง ๆ ซึ่งจะไม่มีผลกระทบต่อ scope นั่นคือสามารถพิจารณาได้จากตัวโค้ดโดยตรงว่า ณ จุดหนึ่ง ๆ ของโปรแกรม binding ไหนสามารถใช้งานได้ จึงสามารถจัดการ scope ได้ตั้งแต่ช่วงการคอมไพล์

2.5.1.1 Classic Example: Nested Scope in Nested Subroutines

ในหลาย ๆ ภาษาโปรแกรมจะมีความสามารถอย่างหนึ่ง คือ nested subroutines หรือสามารถประกาศโปรแกรมย่อยซ้อนอยู่ภายในโปรแกรมย่อยได้ พิจารณาจากโค้ดซึ่งมีหมายเลขบรรทัดกำกับต่อไปนี้

```
1:  procedure P1(A1 :T1);
2:  var X : real;
3:      ...
4:      procedure P2(A2 : T2);
5:          ...
6:          procedure P3(A3 : T3);
7:          ...
```

```
 8:        begin
 9:           ...    (* body of P3 *)
10:        end;
11:        ...
12:     begin
13:        ...       (* body of P2 *)
14:     end;
15:     ...
16:     procedure P4(A4 : T4);
17:        ...
18:        function F1(A5 : T5) : T6;
19:        var X : integer;
20:        ...
21:        begin
22:           ...    (* body of F1 *)
23:        end;
24:        ...
25:     begin
26:        ...    (* body of P4 *)
27:     end;
28:     ...
29:  begin
30:     ...    (* body of P1 *)
31:  end;
```

ในภาษาโปรแกรมที่มี nested subroutines นั้นจะมีการหา binding ของชื่อต่างด้วยวิธี closest nested scope rule คือ การมองหาการประกาศตัวแปรใน scope ระดับที่ใกล้ที่สุด ณ จุดที่เราต้องการอ้างถึงตัวแปรนั้น โดยมีกฎดังนี้

- ชื่อจะถูกรู้จักหรือเห็นได้ภายใน scope ที่ชื่อนั้นถูกประกาศไว้ และทุก scope ที่อยู่ภายใน
- แต่ชื่อที่ประกาศจะถูกซ่อนหรือบดบังจาก ชื่อเดียวกันที่ถูกประกาศซ้ำอยู่ใน scope ที่อยู่ภายในในระดับที่ลึกมากกว่า
- ในการหาว่า object ใด ๆ ถูก bind ไว้กับชื่อใดนั้นทำได้โดย
    - เริ่มค้นหาจากการประกาศชื่อนั้นใน scope ระดับในหรือลึกสุดก่อน

- ถ้าพบการประกาศชื่อนั้นใน scope คือ พบการ bind ของ object นั้นแล้ว
- หากไม่พบก็ไปค้นใน scope ระดับนอกกว่าออกมาเรื่อย ๆ ที่ละระดับจนกว่าจะพบการประกาศชื่อนั้น

หากมีการประกาศตัวแปรใด ๆ ชื่อตัวแปรนั้นจะ active ภายใน scope ที่ตัวแปรถูกประกาศเอาไว้ เช่น จากตัวอย่างโค้ด ใน procedure P1 มีการประกาศตัวแปร X ชนิด real ไว้ นั่นคือ ชื่อตัวแปร X จะ active ภายใน scope ของ procedure P1 ซึ่งจะรวมไปถึง scope procedure ที่ถูกประกาศไว้ภายใน scope ของ P1 คือ P2 และ P4 และยังขยายต่อไปยัง procedure และ function ที่ซ้อนลึกลงไปอีก ซึ่ง P3 ภายใต้ P2 และ F1 ภายใต้ P4

ตัวอย่างในการหาว่า ชื่อนี้คือ binding ไหนนั้น ยกตัวอย่าง เช่น หากในส่วนของโปรแกรมย่อย P3 มีการเขียนคำสั่งในบรรทัดที่ 9 เป็น X = A3

หากต้องการทราบว่า A3 เป็น binding ใด ก็เริ่มดูจาก block ของ scope ที่ใกล้ที่สุด ก่อนนั่นคือ block ของ P3 ซึ่งเมื่อพิจารณาแล้ว จะพบว่ามีการประกาศตัวแปร A3 ไว้ภายใน scope นี้ คือเป็นอาร์กิวเมนต์ของโปรแกรมย่อยชนิด T3 ของ P3 นั่นเอง

เช่นเดียวกันหากต้องทราบว่า binding ของ X ณ บรรทัดที่ 9 เกิดขึ้นที่ไหน ก็เริ่มไล่จาก scope ที่ใกล้ที่สุดคือ P3 ซึ่งจะไม่พบการประกาศ ดังนั้นจึงขยายไปยัง scope ที่ใหญ่กว่าหรือไกลออกไป คือ P2 เมื่อไม่เจอก็ขยายไปยัง P1 ตามลำดับ ซึ่งจะพบการประกาศตัวแปร X ชนิด real นั่นคือ ตัวแปร X ณ บรรทัดที่ 9 เป็นการอ้างถึงตัวแปร X ซึ่งประกาศไว้ในบรรทัดที่ 2 นั่นเอง

จากโค้ดเราสามารถบอก scope ของ binding ต่าง ๆ ซึ่งถูกประกาศตามลำดับของ Nested subroutine ได้ ดังต่อไปนี้

A1 ประกาศไว้ที่บรรทัดที่ 1 ใน scope P1 มีสถานะ active ตั้งบรรทัดที่ 1 ถึง 31

X ประกาศไว้ที่บรรทัดที่ 2 ใน scope P1 มีสถานะ active ตั้งบรรทัดที่ 1 ถึง 17 และบรรทัดที่ 24 ถึง 31

P2 ประกาศไว้ที่บรรทัดที่ 4 ใน scope P1 มีสถานะ active ตั้งบรรทัดที่ 1 ถึง 31

A2 ประกาศไว้ที่บรรทัดที่ 4 ใน scope P2 มีสถานะ active ตั้งบรรทัดที่ 4 ถึง 14

P3 ประกาศไว้ที่บรรทัดที่ 6 ใน scope P2 มีสถานะ active ตั้งบรรทัดที่ 4 ถึง 14

A3 ประกาศไว้ที่บรรทัดที่ 6 ใน scope P3 มีสถานะ active ตั้งบรรทัดที่ 6 ถึง 10

P4 ประกาศไว้ที่บรรทัดที่ 16 ใน scope P1 มีสถานะ active ตั้งบรรทัดที่ 1 ถึง 31

A4 ประกาศไว้ที่บรรทัดที่ 16 ใน scope P4 มีสถานะ active ตั้งบรรทัดที่ 16 ถึง 27

F1 ประกาศไว้ที่บรรทัดที่ 18 ใน scope P4 มีสถานะ active ตั้งบรรทัดที่ 16 ถึง 27

A5 ประกาศไว้ที่บรรทัดที่ 18 ใน scope F1 มีสถานะ active ตั้งบรรทัดที่ 18 ถึง 23

```
1:  procedure P1(A1 :T1);
2:  var X : real;
3:      ...
4:      procedure P2(A2 : T2);
5:          ...
6:          procedure P3(A3 : T3);
7:          ...
 8:          begin
 9:              ...    (* body of P3 *)
10:          end;
11:          ...
12:      begin
13:          ...        (* body of P2 *)
14:      end;
15:      ...
16:      procedure P4(A4 : T4);
17:          ...
18:          function F1(A5 : T5) : T6;
19:          var X : integer;
20:          ...
21:          begin
22:              ...    (* body of F1 *)
23:          end;
24:          ...
25:      begin
26:          ...    (* body of P4 *)
27:      end;
28:      ...
29:  begin
30:      ...    (* body of P1 *)
31:  end;
```

X ประกาศไว้ที่บรรทัดที่ 19 ใน scope F1 มีสถานะ active ตั้งบรรทัดที่ 18 ถึง 23

จากรายการข้างต้นจะพบว่ามีการประกาศ binding ชื่อ X ถึง 2 ครั้งนั่นใน P1 และ F1 ในกรณีนี้ จะมีกฎกำหนดไว้ว่า binding ใด ๆ จะถูกซ่อนไว้จาก scope ที่ซ้อนลงไปหามีการประกาศชื่อเดียวกันซ้ำ นั่นคือ ภายใน scope ของ F1 (บรรทัดที่ 16 ถึง 27) จะไม่รู้จักตัวแปร X ชนิด real แต่รู้จัก X ที่เป็นชนิด integer เพราะถูกประกาศซ้ำกัน

และจากรายการ binding ข้างต้นหากต้องการจะทราบเซตของ binding environment ณ ตำแหน่งบรรทัดใด เช่น ของ P3 คือบรรทัดที่ 6 ก็สามารถดูได้จากรายการที่มีสถาน active ครอบบรรทัดนี้ไว้ ซึ่งได้แก่ A1, X (real), P2, A2, P3, A3 และ P4 เป็นต้น

2.5.1.2 Access to Non-local Objects in Nested Subroutines

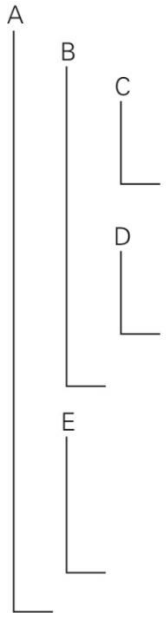
ในการทำงานในการเรียกโปรแกรมย่อยจะมีการ push ข้อมูลการ binding ต่าง ๆ ลงใน stack ด้วย เช่น local variables และ Arguments เป็นต้น คำถามที่เกิดขึ้นในการค้น stack นี้ด้วย nested scope rule เพื่อหาชื่อเหล่านี้ใน stack ได้อย่างไร เช่น ในโปรแกรมย่อย P3 จะเข้าถึงค่าของตัวแปร X ซึ่งถูก push อยู่ในเฟรมของ P1 ได้อย่างไรโดยที่ข้ามเฟรมที่ไม่เกี่ยวข้องออกไป เช่น F1 เรียก A3 เมื่อต้องการทราบการอ้างอิงของตัวแปร X ใน Scope ของ A3 จะข้ามเฟรมของ F1 ได้อย่างไร

วิธีการนั้นก็ คือ ในทุกเฟรมที่ถูก push ลงใน Stack จะมีใส่ข้อมูลเรียกว่า static link ลงไปด้วย โดย static link นี้จะเป็นค่าที่ชี้ไปยัง parent frame ซึ่งเป็นเฟรมของโปรแกรมย่อยที่ครอบโปรแกรมย่อยนั้น ๆ ตามโครงสร้างการซ้อนของโปรแกรมย่อยที่ได้เขียนไว้ในโปรแกรม และโดยจะเป็นเฟรมล่าสุดซึ่งถูก push ลงใน Stack โดยค่า static link ที่เก็บจะเป็นค่าของ frame pointer ซึ่งจะ push ลงไปในส่วนข้อมูลของ Bookkeeping ใน Stack ซึ่งจะเป็นค่าที่จะใช้สำหรับใช้ในการกระโดดไปค้นหา binding ที่อยู่นอก scope ของตนเองไปหา scope ที่ใหญ่กว่าได้ถูกต้อง

พิจารณาโปรแกรมที่มีลักษณะ nested subroutine โดยมีการซ้อนของโปรแกรมย่อย ได้แก่ A, B, C, D และ E คือ A { B { C, D } E } อธิบายได้ว่า ในโปรแกรมย่อย A มีโปรแกรมย่อย B และ E อยู่ภายใน และในโปรแกรมย่อย B มีโปรแกรมย่อย C และ D อยู่ภายใน โดยหากเขียนเป็นโค้ดจะมีโครงสร้าง ดังต่อไปนี้

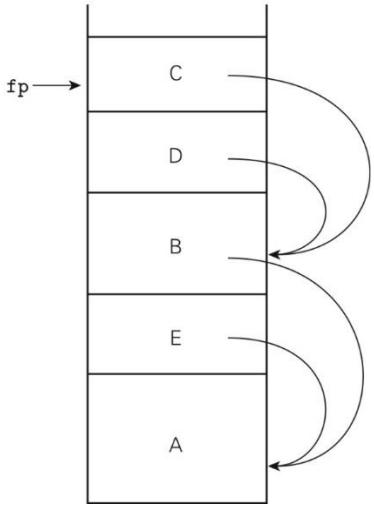
```
procedure A();
   procedure B();
      procedure C();
      begin
         ... (* body of C *)
      end;
      procedure D();
      begin
         ... (* body of D *)
      end;
   begin
```

```
        ... (* body of B *)
    end;
    procedure E();
    begin
        ... (* body of E *)
    end;
begin
  ... (* body of A *)
end;
```



สมมติว่าในตอน run-time มีการลำดับการเรียกโปรแกรมย่อยซ้อนกันโดยมีลำดับ คือ A, E, B, D และ C ข้อมูลเฟรมใน Stack ที่เก็บค่า parent frame มีรูปแบบเป็น “ชื่อเฟรม(ชื่อเฟรมแม่)” เช่น B(A) คือ เฟรม B มี parent frame เป็น A จะเป็นดังนี้

```
Stack contents: A, E(A), B(A), D(B), C(B)
```



จากข้อมูลใน Stack เฟรมจะสามารถคงความสัมพันธ์ระหว่าง Scope เอาไว้ได้ และเมื่อต้องค้นหา binding การท่องตาม scope จะสามารถทำได้ถูกต้องผ่าน static link เช่น เมื่อต้องการหาตัวแปรชื่อ X ที่ประกาศไว้อยู่ในเฟรม A จาก scope ของ C การค้นหาก็จะเริ่มจากเฟรม C เมื่อไม่พบก็จะย้ายตาม Static link นั่นคือ ย้ายไปยังเฟรม B จนกระทั่งไปพบในเฟรม A ซึ่งจะข้ามเฟรม D และ E ไปซึ่งเป็น scope ที่เฟรม C ไม่สามารถจะเข้าถึงได้ โดยจะเป็นไปตามโครงสร้างที่เขียนไว้ในโปรแกรม

Exercise 2.4 Static Link in Nested Subroutines

ให้เขียนแสดงสถานะของข้อมูล Stack ในแต่ละครั้งที่โปรแกรมย่อยถูกเรียกพร้อมทั้งอธิบายการค้นหาตัวแปรผ่าน static link โดยในส่วนของข้อมูล bookkeeping ให้เพิ่มการแสดงรายละเอียดข้อมูล frame pointer (fp) ด้วยว่าเก็บข้อมูลอะไรไว้ในรูปแบบของเซตซึ่งแต่ละข้อมูลเรียงลำดับตามการ push จากซ้ายไปขวา เช่น bookkeeping{parent's fp = 1, caller's fp = 10} เป็นต้น เมื่อมีการเรียกโปรแกรมย่อย P1(1) จากโค้ดดังนี้

```
procedure P1(A1 :T1);
var X : real;
   procedure P2(A2 : T2);
      procedure P3(A3 : T3);
      begin
         X := A3;
      end;
   begin
      P3(3);
   end;
   procedure P4(A4 : T4);
   begin
      P2(2);
   end;
 begin
   P4(4);
end;
```

เฉลย Exercise 2.4

1. Call P1(1) จะได้สถานะของ Stack จากขั้นตอน Pre-call จนกระทั่ง Epilogue เป็น ดังนี้

Stack contents:

A1 = 1, Return Address, Bookkeeping {parent's fp, caller's fp}, X = 0

จากข้อมูลใน Stack แสดงถึงการที่โปรแกรมย่อย P1 ถูกเรียกด้วยอาร์กิวเมนต์ A1 มีค่าเท่ากับ 1 ข้อมูลในส่วนของ bookkeeping จะมีการเก็บ parent's fp เพิ่มเข้ามา ซึ่งเป็น static link ไปยัง parent frame ในที่นี้จะไม่ทราบค่าจริงเพราะอยู่นอกเหนือ scope ที่สนใจ

2. P1 Call P4(1) จะได้สถานะของ Stack จากขั้นตอน Pre-call จนกระทั่ง Epilogue เป็น ดังนี้

Stack contents:

A1 = 1, Return Address, Bookkeeping {parent's fp, caller's fp}, X = 0,

A4 = 4, P1's Return Address, Bookkeeping {parent's fp = 2, caller's fp = 2}

จากข้อมูลใน Stack แสดงถึงการที่โปรแกรมย่อย P4 ถูกเรียกด้วยอาร์กิวเมนต์ A4 มีค่าเท่ากับ 4 ข้อมูลในส่วนของ bookkeeping จะมีการเก็บ parent's fp เพิ่มเข้ามา ซึ่งเป็น static link ไปยัง parent frame ซึ่งตามโครงสร้างจากโค้ดข้างต้น คือ A1 นั่นคือ 2 และ P4 ถูกเรียกโดย P1 นั่นคือ caller's fp จึงมีค่าเท่ากับ 2 เช่นกัน

3. P4 Call P2(2) จะได้สถานะของ Stack จากขั้นตอน Pre-call จนกระทั่ง Epilogue เป็น ดังนี้

Stack contents:

A1 = 1, Return Address, Bookkeeping {parent's fp, caller's fp}, X = 0,

A4 = 4, P1's Return Address, Bookkeeping {parent's fp = 2, caller's fp = 2},

A2 = 2, P4's Return Address, Bookkeeping {parent's fp = 2, caller's fp = 7}

จากข้อมูลใน Stack แสดงถึงการที่โปรแกรมย่อย P2 ถูกเรียกด้วยอาร์กิวเมนต์ A2 มีค่าเท่ากับ 2 ข้อมูลในส่วนของ bookkeeping จะมีการเก็บ parent's fp เพิ่มเข้ามา ซึ่งเป็น static link ไปยัง parent frame ซึ่งตามโครงสร้างจากโค้ดข้างต้น คือ A1 นั่นคือ 2 และ P2 ถูกเรียกโดย P4 นั่นคือ caller's fp จึงมีค่าเท่ากับ 7 (bookkeeping มีขนาดข้อมูลเท่ากับ 2)

4. P2 call P3 (3) จะได้สถานะของ Stack จากขั้นตอน Pre-call จนกระทั่ง Epilogue เป็น ดังนี้

Stack contents:

A1 = 1, Return Address, Bookkeeping {parent's fp, caller's fp}, X = 0,

A4 = 4, P1's Return Address, Bookkeeping {parent's fp = 2, caller's fp = 2},

A2 = 2, P4's Return Address, Bookkeeping {parent's fp = 2, caller's fp = 7},

A3 = 3, P2's Return Address, Bookkeeping {parent's fp = 11, caller's fp = 11}

จากข้อมูลใน Stack แสดงถึงการที่โปรแกรมย่อย P3 ถูกเรียกด้วยอาร์กิวเมนต์ A3 มีค่าเท่ากับ 3 ข้อมูลในส่วนของ bookkeeping จะมีการเก็บ parent's fp เพิ่มเข้ามา ซึ่งเป็น static link ไปยัง parent frame ซึ่งตามโครงสร้างจากโค้ดข้างต้น P3 ถูกประกาศอยู่ภายใต้ P2 ดังนั้นจึงเป็นค่า fp ของ P2 ที่ถูกเรียกล่าสุด คือ 10 นั่นคือ 1 และ P3 ถูกเรียกโดย P2 นั่นคือ caller's fp จึงมีค่าเท่ากับ 10 เช่นเดียวกัน

5. P3 body ในส่วนของ P3 ได้มีการ assign ค่าให้ตัวแปร X ดังนั้นทำสถานะ ของ Stack เป็นดังนี้

Stack contents:

A1 = 1, Return Address, Bookkeeping {parent's fp, caller's fp}, X = 3,

A4 = 4, P1's Return Address, Bookkeeping {parent's fp = 2, caller's fp = 2},

A2 = 2, P4's Return Address, Bookkeeping {parent's fp = 2, caller's fp = 7},

A3 = 3, P2's Return Address, Bookkeeping {parent's fp = 11, caller's fp = 11}

ในการ assign ค่าตัวแปร X ด้วยค่าอาร์กิวเมนต์ A3 ซึ่งมีค่าเป็น 3 จะเกิดการค้นหาตัวแปรใน Stack ซึ่งถูกประกาศไว้ในเฟรม P1 ซึ่งการค้นหาหากไม่พบในเฟรมของตัวเองจะกระโดด (dereference) ไปหาเฟรมแม่ขอตัวเองผ่านค่า parent's fp ซึ่งในกรณีจะกระโดดผ่านเฟรม P3 ไปยัง P2 ด้วยค่า parent's fp = 10 จากนั่นจาก P2 จะกระโดดไปยัง P1 ซึ่งคือ parent's fp = 1 แล้วค้นเจอตำแหน่งที่เก็บข้อมูลของตัวแปร X คือ offset = 3 เท่ากับ 5 ดังนั้นค่าที่เก็บตัวแปร X จึงถูกปรับปรุงค่าเป็น 3

จบการเฉลย Exercise 2.4

2.5.1.3 Nested Blocks

ในบางภาษาก็มีโครงสร้างในลักษณะของ nesting เช่นเดียวกัน ดังตัวอย่างโค้ดภาษา Java ต่อไปนี้

```
1: //Java
2: class a {
3:   void b(…) {
4:     int c; …
5:     while (…) {
6:       int d; …
```

```
 7:         {
 8:            int e;
 9:            c = d+e;
10:         }
11:      }
12:    }
13: }
```

จากโค้ดจะสังเกตได้ว่ามีการประการศ scope ในรูปแบบ block อยู่ คือ ตั้งแต่บรรทัดที่ 7 – 10 โดยถูกครอบด้วยเครื่องหมายปีก นั่นคือภายในเครื่องหมายปีกกาจะเป็นการเริ่ม Scope ใหม่ ซึ่งทำให้ binding ใดก็ตามที่ถูกประกาศอยู่ภายใน block จะมีช่วงชีวิตอยู่แค่ภายในเครื่องหมายปีกกา

โดยหากจะอธิบาย scope ที่เกิดขึ้นจากโค้ดข้างต้น สามารถดูได้จากการครอบของเครื่องหมายปีกกา ซึ่งจะประกอบด้วย scope ของคลาส a ซึ่งภายในมี scope ของเมธอด b อยู่ภายใน โดยมี scope ของ while block อยู่ภายใน ที่ได้มีการประกาศ block (บรรทัดที่ 7-10) ไว้เป็นชั้นสุดท้าย โดยขอบเขตของ scope จะเป็นแบบเดียวกัน คือ การอ้างอิงไปยังข้อมูลนอก scope จะอ้างไปยัง block ที่ใหญ่กว่าและครอบอยู่เท่านั้น สังเกตได้จากบรรทัดที่ 9 c = d + e ได้มีการอ้างถึงตัวแปรนอก block คือ c และ d เป็นต้น

ในภาษา C ก็มีการซ้อน block เช่นกันดังตัวอย่าง

```
1: //C
2: int a;      /*global var*/
3: main () {
4:    a = 1;
5:    {
6:      int a;
7:      a = 2;
8:      …
9:      {
10:       int a;
11:       a = 3;
12:       …
13:     }
14:   }
15: }
```

จากโค้ดจะเห็นว่ามีการซ้อน block อยู่ 3 ชั้น และมีการประกาศตัวแปรชื่อ a ซ้ำกันอยู่ทั้งหมด 3 scope คือ บรรทัดที่ 2, 6 และ 10 โดยในแต่ละ block มีการ assign ค่าให้กับตัวแปร a ซึ่งหมายถึงแต่ละครั้งที่ assign ค่าให้กับตัวแปร a จะมีผลกับข้อมูลในหน่วยความจำคนละตำแหน่งกันตามกฎการซ่อน binding ของ nested subroutine นั่นคือ จะเห็นเฉพาะตัวแปรที่ประกาศใน scope ที่ใกล้ที่สุด นั่นคือ สุดท้ายแล้วตัวแปร a ที่ประกาศไว้ใน global scope จะมีค่าเท่ากับ 1 ซึ่งเป็นคำสั่ง assign ตัวแรกสุดใน code เพราะคำสั่ง assign อื่นกระทำกับตัวแปร a คนละ scope กัน

ยังการอ้างอิง scope ไปยังตัวแปรที่ถูกประกาศซ้ำได้ เช่น ตัวอย่างโค้ดต่อไปนี้

```
1: //Java
2: class a {
3:  int c;
4:  void b(…) {
5:   int c;
6:   c =1;
7:   this.c = 2;
8:   …
9:  }
10:}
```

จากโค้ดจะสังเกตเห็นว่า มีการประกาศตัวแปร c ซ้ำกัน คือ บรรทัดที่ 3 และ 5 ซึ่งตัวแรกอยู่ใน class scope ส่วนตัวหลังอยู่ใน method scope หากในเมธอด b มีการอ้างถึงตัวแปร c เฉยๆ จะถูกแปลตาม scope นั่นคือ อ้างถึงตัวแปรที่ประกาศไว้ภายในเมธอด นั่นคือในบรรทัดที่ 6 จะเป็นการ assign คือ 1 ให้กับตัวแปรที่ประกาศไว้ใน บรรทัดที่ 5 แต่สำหรับการเขียนโปรแกรมเชิงวัตถุสามารถกำหนด scope เข้าถึงตัวแปรได้ ดังตัวอย่างที่แสดงในบรรทัดที่ 7 แสดงการระบุ class scope ซึ่งจะเป็นการ assign ค่า 2 ให้กับตัวแปร c ที่ประกาศในบรรทัดที่ 3 นั่นเอง

2.5.2 Dynamic Scoping

ภาษาโปรแกรมแบบ Dynamic Scope จะพิจารณา binding จากลำดับขั้นตอนการทำงานของโปรแกรม แทนที่จะพิจารณาจากโครงสร้างของโค้ดโดยตรงอย่างของ Static Scope นั้นคือ การค้นหาชื่อตัวแปรว่ายึดเหนี่ยวกับอ็อบเจกต์ใด จะขึ้นอยู่ลำดับของการเรียกซ้อนกันของโปรแกรมย่อย นั่นคือ การค้นหาจะมุ่งไปยังชื่อที่ถูก bind ไว้ล่าสุดและยังคงอยู่ ไม่ถูกทำลายไปโดยการที่การประมวลผลพ้น scope ของ binding นั้นไป นั่นคือ ทุกอย่างจะเกิดขึ้นในขณะ run-time ทั้งการตรวจสอบชิดของตัวแปรในนิพจน์ต่าง ๆ รวมไปถึงอาร์กิวเมนต์ของโปรแกรมย่อยที่ถูกเรียก

ภาษาโปรแกรมที่ใช้ dynamic scope มักจะเป็นภาษาที่ใช้ interpreter เป็นตัวแปลภาษา มากกว่าที่จะพบในภาษาที่ใช้ compiler ตัวอย่างเช่น ภาษา Lisp และ Perl เป็นต้น

พิจารณาจากโค้ดดังต่อไปนี้

```
1: a: integer  -- global declaration
```

```
2: procedure first
3:    a := 1
4: procedure second
5:    a : integer  -- local declaration
6:    first()
7: a := 2
8: if read_integer() > 0
9:    second()
10: else
11:   first()
12: write_integer(a)
```

จากโค้ดประกอบด้วยการประกาศตัวแปร a เป็นตัวแปรแบบ global และมี procedure 2 ตัว คือ first และ second ซึ่งมีการประกาศตัวแปร a เป็นแบบ local อยู่ภายใน และโปรแกรมเริ่มการทำงานตั้งแต่บรรทัดที่ 7 เป็นต้นไป

เมื่อพิจารณาถึงตัวแปรชื่อ a จะปรากฏอยู่ในหน่วยความจำอยู่ 2 ส่วน คือ data segment สำหรับการประกาศแบบ global และใน Stack สำหรับการประกาศแบบ local และหากพิจารณาการ assign ค่าตัวแปรในแบบ static scope ในบรรทัดที่ 3 ผลของการทำงานของโปรแกรมจะเป็นการบันทึกค่า 1 ลงไปยังหน่วยความจำที่จองไว้ในส่วนของ data segment นั่นคือ ไปยังตัวแปร global ที่ถูกประกาศไว้ในบรรทัดที่ 1 ดังนี้ หากโปรแกรมนี้ทำงานจนถึงบรรทัดสุดท้ายซึ่งเป็นการพิมพ์ค่าตัวแปร a ออกมา จะเป็นการพิมพ์ค่าตัวแปรที่มีค่าเป็น 1 ออกมา เพราะในการทำงานโปรแกรมย่อยถูกเรียกทำงานทุกครั้ง นั้นคือ เมื่อบรรทัดที่ 3 ทำงาน ค่าตัวแปร a ที่เป็น global จะถูก assign ค่าเป็น 1 ดังนี้ผลการทำงานโปรแกรมจะพิมพ์เลข 1 ออกมาเสมอไม่ว่าโปรแกรมจะเกิดการเรียกโปรแกรมย่อย first หรือ second

แต่หากเป็นการทำงานแบบ dynamic scope ผลลัพธ์การทำงานจากแตกต่างกันออกไป หากโปรแกรมย่อย second ถูกเรียกให้ทำงาน โดยสมมติ ว่าในบรรทัดที่ 8 ฟังก์ชัน read_integer() ให้ผลลัพธ์ที่มีค่ามากกว่า 0 ณ บรรทัดนี้ ค่าตัวแปร a ถูก assign ด้วยค่า 2 ก่อนแล้ว ซึ่งด้วยเงื่อนไขดังกล่าวทำให้โปรแกรมเกิดการเรียกโปรแกรมย่อย second และซ้อนด้วย first ตามลำดับ จึงทำให้เกิดข้อมูลใน Stack เมื่อโปรแกรมทำงานถึงบรรทัดที่ 3 ดังนี้

Stack Content:

Return address, bookkeeping, a = 0, Return Address, bookkeeping

Data Segment:

a = 2

จากข้อมูลใน Stack จดประกอบด้วย 2 เฟรม คือ second ตั้งแต่ตำแหน่งที่ 1 – 3 และ first ตั้งแต่ตำแหน่งที่ 4 – 5 ดังนี้ ขณะนี้ที่บรรทัดที่ 3 จะมี ตัวแปร a อยู่ใน Stack ด้วย ในการค้นหาของ dynamic scope จะทำเช่นเดียวกับ static scope คือ เริ่ม

ค้นใน Stack ก่อนถัดไปจึงเป็น Data Segment เว้นเสียแต่ว่าจะไม่มี Static link แต่จะท่องไปตาม caller's fp แทน ซึ่งเท่ากับต้องค้นจากทุกเฟรมใน Stack ดังนี้ในกรณีนี้ ตัวแปร a ที่ถูก assign ค่า 1 ตามบรรทัดที่ 3 นั้น จึงเป็นตัวแปร a แบบ local ที่ประกาศไว้ในบรรทัดที่ 5 ภายใน procedure ชื่อ second นั่นเอง จึงทำให้สถานะหน่วยความจำเป็นดังนี้

Stack Content:

Return address, bookkeeping, a = 1, Return Address, bookkeeping

Data Segment:

a = 2

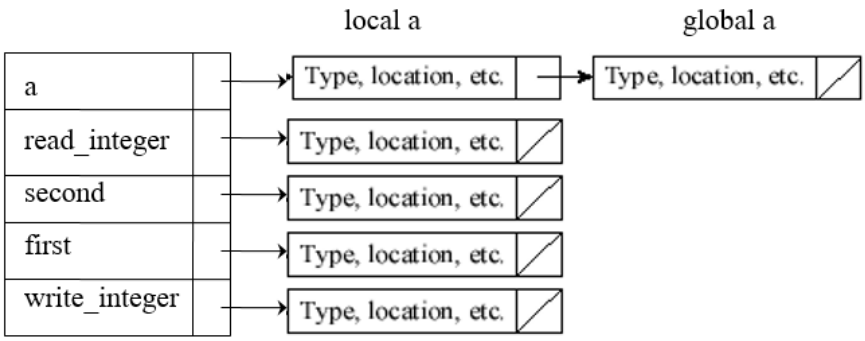
และเมื่อโปรแกรมทำงานจนถึงบรรทัดสุดท้ายที่สั่งพิมพ์ค่าตัวแปรชื่อ a สิ่งที่เกิดขึ้น คือ ข้อมูลใน Stack ข้างต้นถูกล้างหมด เนื่องจากขั้นการจบการทำงานของโปรแกรมย่อย second ทำให้ตัวแปร a แบบ local ถูกทำลายไป ตอนนี้การอ้างชื่อตัวแปร a จึงเป็นการอ้างถึงตัวแปรแบบ global ทำให้ค่าที่ถูกพิมพ์ออกมาจึงมีค่าเท่ากับ 2 แทนที่จะเป็นค่า 1 เพราะการทำงานใน procedure ชื่อ first นั้นได้กระทำกับตัวแปร local ซึ่งหมดช่วงชีวิตไปแล้วนั่นเอง

2.5.2.1 Looking up for Bindings in Dynamic Scoping

เนื่องจากไม่มีการใช้ Static link การค้นหาจึงต้องทำทั้ง Stack ดังนั้นแทนที่จะต้องค้นหา binding ใน Stack ก็ได้มีการใช้หน่วยความจำสำหรับเก็บข้อมูล binding แยกออกมาต่างหาก โดยจะเป็นตารางแสดงรายชื่อ binding ที่ไม่ซ้ำกันอาจจะใช้เป็น Hash table เพื่อแสดงว่าขณะนั้น binding ได้ถูกยึดเหนี่ยวไว้กับอ็อบเจกต์หรือหน่วยจำใดอยู่ โดยแต่ละรายการในตารางจะเสมือนเป็นหัวของ linked list ชี้ไปยังอ็อบเจกต์ที่เป็นตัวล่าสุด ดังนั้น เมื่อมี binding ใหม่เกิดขึ้นกับชื่อใด ๆ ในรายการนี้ก็จะถูกนำมาแทรกถัดจากหัวของรายการเหล่านี้ และเมื่อพ้น scope ตัวที่เข้ามาใน linked list ล่าสุดก็จะถูกเอาออกก่อน (Last In First Out: LIFO)

ต่อไปนี้เป็นตัวอย่างตาราง binding ของโปรแกรมตัวอย่างข้างต้น เมื่อ procecure ชื่อ first ถูกเรียกจาก procedure ชื่อ second

```
Binding Table (binding = data pointer):
a = 6, read_integer = 2, second = 3, first = 4, write_integer 5
Binding Object (Address: Binding object = previous binding object pointer):
1: global a = NULL, 2: read_integer = NULL, 3: second = NULL, 4: first = NULL,
5: write_integer = NULL, 6: local a = 1
   *Note: each binding object may contain information about type, location, etc.
```



จากโครงสร้างข้อมูลตัวอย่างเมื่อมีการอ้างถึงชื่อ binding table ก็จะถูกค้นเพื่อหาว่า binding นั้นอ้างอิงไปยัง binding object ใด เช่น ฟังก์ชันต่าง ๆ ต้องไปเรียกที่หน่วยความจำใด ของฟังก์ชันได้แก่ read_integer, second, first, write_integer หรือ หากเป็นตัวแปรก็จะบอกว่าจัดเก็บไว้ที่ตำแหน่งใด เป็นต้น โดยข้อมูล type ของ binding object จะบอกชนิดของ binding เช่น a มี type เป็น variable หรือ read_integer มี type เป็น procedure เป็นต้น

และตัวอย่างโครงสร้างหากพิจารณา binding ชื่อ a ก็จะเป็นตัวแปรที่เกิด binding ตัวแปร 2 ตัว คือ แบบ global และ local ตามลำดับ โดย binding ที่ active อยู่คือ binding ตัวที่ binding table ชี้อยู่ กรณีนี้คือ 6 ก็คือตัวแปร a แบบ local นั่นเอง และหากต่อมา procedure ชื่อ second จบการทำงานลง binding object ชื่อ a คือตำแหน่งที่ 6 จะถูกลบออกและเปลี่ยนไปชี้ยังตำแหน่งที่ local a ชี้อยู่คือ 1 นั่นเอง ทำตัวแปร global เกิด active ขึ้นแทน

Exercise 2.5 Bindings in Dynamic Scoping

จากโค้ดตัวอย่าง dynamic scope ข้างต้น หากในส่วน Main นั้น procedure ที่ถูกเรียกคือ first (กระทำการยังบรรทัดที่ 11) ให้ตอบคำถามต่อไปนี้

1. What does write_integer refer to, global a or local a?
2. What does write_integer write?

เฉลย Exercise 2.5

คำตอบข้อ 1 คือ global a

อธิบายคำตอบ พิจารณาจาก flow ของการทำงานของโปรแกรมไม่มีการเรียกฟังก์ชัน second จึงไม่มีโอกาสเกิดของ local a และแม้จะมีการเรียกฟังก์ชัน second ขณะที่กระทำการบรรทัดของ write_integer ฟังก์ชัน second ก็ทำงานเสร็จสิ้นแล้วดังนั้นช่วงชีวิต local a ก็จะหมดลงเช่นกัน จึงเหลือแต่ global a ที่ยังคงมีชีวิตอยู่

คำตอบข้อ 2 คือ 1

```
2: procedure first
3:    a := 1
4: procedure second
5:    a : integer  -- local declaration
6:    first()
7: a := 2
8: if read_integer() > 0
9:    second()
10:else
11:   first()
12:write_integer(a)
```

อธิบายคำตอบ พิจารณาการ assign ค่าให้กับตัวแปรชื่อ a ซึ่ง ได้แก่บรรทัดที่ 3 และ 7 โดยพิจารณาจากลำดับการกระทำการ คือ บรรทัดที่ 7 ก่อน ขณะนั้นยังไม่มีโปรแกรมย่อยใด ๆ ถูกเรียก ดังนั้น Stack จึงยังว่าอยู่ โดยสถานะหน่วยความจำจะเป็นดังนี้

```
Stack content is empty:
Data Segment:
a = 2
```

และเมื่อการกระทำการมาถึงบรรทัดที่ 3 ซึ่งมีการ assign ค่า 1 ไปยังตัวแปร a สถานะหน่วยความจำจึงเปลี่ยนเป็นดังนี้

```
Stack content:
Return address, bookkeeping
Data Segment:
a = 1
```

ขณะกระทำการฟังก์ชัน first ไม่มี binding a ปรากฏใน Stack ดังนั้น การ assign ค่า จึงถูกกระทำกับตัวแปร global a นั่นเอง และเมื่อ write_integer พิมพ์ค่าตัวแปร global a จึงได้ผลลัพธ์เป็น 1 นั่นเอง

จบการเฉลย Exercise 2.5

ในเนื้อหาส่วน dynamic scope นี้จะเห็นได้ว่า flow หรือลำดับการทำงานของโปรแกรมมีผลกระทบต่อผลลัพธ์การทำงานของโปรแกรม ทำให้ได้ผลลัพธ์ที่แตกต่างกันได้ ขึ้นอยู่กับ binding ที่เกิดขึ้นในแต่ละการทำงานของโปรแกรม