Abstract Data Types (ADT)

(ดูรายละเอียดในเรื่อง linked lists, statcks, และ queues สำหรับข้อมูลเบื้องต้น)
คุณลักษณะที่สำคัญของ ADT คือ ไม่มีการจำกัดชนิดของข้อมูลที่จะใส่ใน ADT ทำให้โครงสร้างข้อมูลชนิดนี้มีความยืดหยุ่นสูง เพราะสามารถรองรับข้อมูลได้ทุกประเภท (ในทางทฤษฎี) จึงไม่มีข้อจำกัดในการออกแบบและจัดการกับข้อมูลที่เก็บใน ADT เพียงแต่ต้องดำเนินการตาม operations เฉพาะของ ADT นั้นๆ ในที่นี้จะกล่าวถึง operations และ ADT สำคัญๆ ที่เกี่ยวข้อง คือ

- Push, pop
- Enqueue, dequeue
- monitors

Push และ pop

เป็น operations สำหรับ ADT ประเภท stack ซึ่งเป็นการทำงานในลักษณะ “มาทีหลัง-ออกก่อน” (LIFO หรือ Last-In First-Out) โดยทั่วไปการประยุกต์ใช้งานมักจะทำด้วย array เพราะง่ายและสะดวก

ในการทำงาน เราอาจประยุกต์หลักการออกแบบเชิงวัตถุกับ stack โดยมองว่า สิ่งที่ใส่ใน stack คือวัตถุที่สร้างขึ้น ซึ่งอาจจะเป็นวัตถุพื้นฐานประเภท integer, character (string), double ไปจนถึง class ที่ซับซ้อน เมื่อเรากำหนดชนิดของวัตถุที่ใช้งาน นั่นหมายถึงแต่ละ “ชั้น” ของ stack มีคุณสมบัติเป็นข้อมูลชนิดนั้น เช่น ถ้ากำหนดว่าวัตถุ (ภาษาซี) เป็นชนิด struct ทุกๆ ชั้นของ stack ก็จะเก็บข้อมูลชนิด struct หมด ดังตัวอย่างต่อไปนี้ (ใช้ array ทำหน้าที่เป็น stack โดยสร้าง pointer TOS ชี้ที่ปลายด้านใดด้านหนึ่งของ array ที่กำหนดให้เป็นส่วนบนของ stack)

```
const int   Max_size    = 100;
enum        status
{
      EMPTY = -1,
      FULL  = Max_size – 1
};

struct      st
{
      int   data[Max_size];
      int   TOS;
};
struct      st    STACK;
. . .
STACK.TOS   = (int)EMPTY;          /* initialization of TOS      */
```

หรืออีกวิธีหนึ่งที่ใช้ array of structure เป็นทั้ง stack โดยแต่ละ struct แทนแต่ละชั้นของ stack ดังตัวอย่างข้างล่าง

```
struct  st
{
      int         frequency;
      double      data;
      char        name[40];
};
struct  st  STACK[100], *TOS;
```

ซึ่งเป็นการประกาศวัตถุชนิด “struct st” ชื่อ STACK โดยใช้ array เป็นตัวจำลอง stack และมี pointer ชื่อ TOS ซึ่งเป็นวัตถุชนิดเดียวกันเป็นตัวชี้ top of stack

จากตัวอย่างข้างต้น จะเห็นว่าในมุมมองของ conceptual design เราไม่ต้องกำหนดชนิดของข้อมูล แต่จินตนาการเป็น abstract entity ให้สอดคล้องกับ model ที่เรากำลังจำลอง ปริบทการใช้งานจึงอยู่ในรูปของ ADT และ operations ที่มากับ ADT นั้น เช่น ในระดับ modeling เราอาจต้องการสร้างโปรแกรมเครื่องคิดเลขชนิด Reverse Polish Notation (RPN) เราสามารถจำลองการทำงานของเครื่องคิดเลขที่ต้องการด้วย stack ซึ่งไม่ต้องคำนึงว่าค่าตัวเลข (input) จะต้องเป็น int, float, double, long ฯลฯ เมื่อถึงเวลา implement จริง เราจึงตัดสินใจเลือกประเภทของ data structure ที่เหมาะสม (ซึ่ง primitive type ที่มี อาทิ int, short, double เพียงอย่างเดียวไม่สามารถจำลอง stack ได้) ในกรณีนี้ ทางเลือกที่เหมาะสมคือ struct ซึ่งอาจจะประกอบด้วย primitive type หลายชนิด (ซ้อนกันบนพื้นที่ร่วมแบบ union และฝังอยู่ภายใน struct) เพื่อความยืดหยุ่นสูงของการประยุกต์ใช้งาน ส่วน operations ที่เกี่ยวข้องกับเครื่องคิดเลขดังกล่าวก็คือ push และ pop เราก็จะได้เครื่องคิดเลขระบบ RPN ที่สมบูรณ์

Enqueue และ dequeue

Operations ทั้งสองวิธีนี้ ใช้กับ ADT แบบ queue ซึ่งเป็นการจัดระเบียบการเข้าออกในลักษณะ “มาก่อน-ออกก่อน” (First-in, First-out) ซึ่งมักจะรู้จักกันในนามของ FIFO queue การ implement จึงทำได้อย่างตรงไปตรงมา ส่วนมากจะนิยมใช้ array เป็น data structure หลัก เพราะความง่ายของการเขียนโปรแกรม แต่ก็มีข้อจำกัดของ array ที่ทำให้ต้องเลี่ยงไปใช้ circular queue แทน (บางคนจึงใช้ singly-linked list แทน เพราะสามารถจำลองได้เหมือนกว่า ไม่เปลืองเนื้อที่มาก และไม่ต้องพะวงกับการ insert และ delete ณ ตำแหน่งที่เหมาะสม)

```
#define     Max_no      100
enum        que
{
      ENQUEUE     = 1,
      DEQUEUE     = 0
};
```

```
struct      qq
{
      int         data[Max_no];
      enum  que   queue_state;
};
struct      qq    FIFO_Q;
```

ตัวอย่างข้างต้น แสดงการจำลองโครงสร้างของ queue ด้วย structure โดยตัว queue เองเป็น array ที่มีขนาด Max_no และมี flag บอกสถานะของ queue ที่ขณะใดขณะหนึ่ง (enqueue - เข้า หรือ dequeue - ออก) เพื่อป้องกันการกระทำการที่เหลื่อมซ้อนกันภายใต้ multi-programming environment กล่าวคือ process หนึ่งอาจจะทำการ enqueue วัตถุใส่ใน queue ในขณะที่อีก process ต้องการ dequeue วัตถุออกจาก queue ในช่องเดียวกัน flag ดังกล่าวจึงทำหน้าที่เป็นตัวสลับสถานะการทำงาน เพื่อให้แต่ละ process เข้าสู่ queue (ซึ่งในกรณีนี้เปรียบเสมือน critical section) ตามลำดับที่ถูกต้อง

Monitors

เป็น ADT ที่มีบทบาทสำคัญมากในเรื่อง process synchronization ของระบบปฏิบัติการ operations ที่ใช้ภายใน monitor มักจะเป็น enqueue และ dequeue

โครงสร้างของ monitor มีส่วนประกอบย่อยที่ซับซ้อนมากมาย encapsulate อยู่ภายในกรอบใหญ่ของ monitor ในที่นี้ จะทำให้ง่ายลงโดยเหลือเพียงส่วนสำคัญๆ บางส่วนเพื่อประกอบคำอธิบายภายใต้ปริบทของโครงสร้างข้อมูลเท่านั้น

```
#define     Max         10
#define     Cap         50
struct      part
{
      int   data;                     /* critical section        */
      int   queue[Max];               /* queue size              */
      int   cond_var;                 /* to wait on this queue   */
};

struct      monitor
{
   int      local;                    /* local monitor data      */
   int      trigger_cond;             /* monitor condition var   */
   struct   part    channel[Cap];     /* 50 regular channels     */
   struct   part    urgent;           /* one urgent channel      */
};
struct      monitor     MONITOR;
```

โครงสร้างข้างต้นเป็นกรอบขนาดใหญ่ที่มี local เป็นข้อมูลของตัว monitor เองสำหรับจัดการเกี่ยวกับการทำงาน โดยมี trigger_cond เป็นตัวควบคุม แต่ละ process จะกระทำการได้ก็ต่อเมื่อเงื่อนไขที่ตั้งไว้สำหรับ process นั้นๆ เป็นจริง เช่น process A อาจจะคอยให้ค่าของ trigger_cond เป็นคี่จึงเริ่มทำงาน ดังนั้น เมื่อเข้าไปคอยใน monitor ก็อาจจะถูกจัดให้อยู่ใน channel ที่มี cond_var เป็นค่าคี่ โดย

แต่ละ channel จะมีเงื่อนไขของตัวเอง หาก process ใดเข้าข่ายเงื่อนไขที่กำหนดไว้สำหรับ channel นั้นๆ เมื่อเข้าไปใน monitor ก็จะถูกจัดไว้ใน channel เฉพาะ เช่น channel ที่หนึ่งอาจกำหนดไว้สำหรับ cond_var ที่มีค่ามากกว่า 64 (สมมุติว่าค่าดังกล่าวนี้แทน priority ของ process นั้น) channel ที่สอง สำหรับ cond_var ที่มีค่าเป็นเลขคี่ channel ที่สามสำหรับ cond_var ที่มีค่าเป็นจริงเมื่อ trigger_cond เป็นจริง เป็นต้น สำหรับ process ที่มีความสำคัญมาก โดยเฉพาะ kernel process ก็จะถูกจัดไว้ใน urgent ซึ่งเป็น channel (หรือ queue) พิเศษที่จะได้รับการให้บริการจากระบบปฏิบัติการก่อน channel อื่น

จากตัวอย่างจะเห็นว่า monitor เป็น ADT ที่มีประโยชน์ในงานประยุกต์มาก ถือเป็น programming construct ที่ลดภาระการเขียนโปรแกรมของผู้ใช้ได้มาก เพราะไม่ต้องกังวลถึงกลไกในการ synchronize การทำงานของ process ต่างๆ สามารถทุ่มเทความสนใจไปกับการ “จัดระเบียบ” การทำงาน ไม่ให้เกิดการติดตาย (deadlocks) อันเป็นหัวใจสำคัญของระบบปฏิบัติการ