```
#include <stdio.h>
#include <stdlib.h> // required for malloc()
//  Queue ADT Type Defintions คิวประเภาคำนิยาม
typedef struct node
{
void*       dataPtr;
struct node* next;
} QUEUE_NODE;
typedef struct
{
QUEUE_NODE* front;
QUEUE_NODE* rear;
int       count;
} QUEUE;

//  Prototype Declarations ประกาศตัวแปร
QUEUE* createQueue  (void);
QUEUE* destroyQueue (QUEUE* queue);

bool  dequeue   (QUEUE* queue, void** itemPtr); // * = pointer พอยเตอร์
bool  enqueue   (QUEUE* queue, void*  itemPtr); // ** = pointer of pointer พอยเตอร์ชี้พอยเตอร์
bool  queueFront (QUEUE* queue, void** itemPtr);
bool  queueRear  (QUEUE* queue, void** itemPtr);
int   queueCount (QUEUE* queue);
bool  emptyQueue (QUEUE* queue);
bool  fullQueue  (QUEUE* queue);
//  End of Queue ADT Definitions จบนิยามของคิว

void printQueue   (QUEUE* stack);

int main (void)
{
```

```
//  Local Definitions
QUEUE* queue1;
QUEUE* queue2;
int*   numPtr;
int** itemPtr;

//  Statements
// Create two queues
queue1 = createQueue();
queue2 = createQueue();
for (int i = 1; i <= 5; i++)
{
numPtr  = (int*)malloc(sizeof(i)); // set pointer to memory
*numPtr = i;
enqueue(queue1, numPtr);
if (!enqueue(queue2, numPtr))
{
printf ("\n\a**Queue overflow\n\n");
exit (100);
} // if !enqueue
} // for
printf ("Queue 1:\n");
printQueue (queue1); // 1 2 3 4 5
printf ("Queue 2:\n");
printQueue (queue2); // 1 2 3 4 5
return 0;
}

/*  ================= createQueue ================= สร้างคิว
Allocates memory for a queue head node from dynamic จัดสรรหน่วยความจำสำหรับโหนดหัวคิวจากไดนามิก
memory and returns its address to the caller. หน่วยความจำและส่งกลับที่อยู่ไปยังผู้โทร
Pre  nothing ไม่มีอะไรก่อน
```

```
Post   head has been allocated and initialized หัวการปัดส่วนและจุดเริ่มต้น
Return head if successful; null if overflow ย้อนกลับเมื่อสำเร็จ ไม่เป็นผลเมื่อ overflow
*/
QUEUE* createQueue (void)
{
// Local Definitions ตำแหน่งนิยาม
QUEUE* queue;
// Statements  คำสั่ง
queue = (QUEUE*) malloc (sizeof (QUEUE));
if (queue)
{
queue->front  = NULL;
queue->rear   = NULL;
queue->count  = 0;
} // if
return queue;
}   // createQueue สร้างคิว

/*  ================= enqueue ================  การนำข้อมูลเข้า
This algorithm inserts data into a queue. ตามข้อมูลแทรกของคิวในอันกอริทึม
Pre  queue has been created จากสร้างพรีคิว
Post   data have been inserted ไม่รู้
Return true if successful, false if overflow ย้อนกลับเมื่อสำเร็จ ไม่เป็นผลเมื่อ overflow
*/
bool enqueue (QUEUE* queue, void* itemPtr)
{
// Local Definitions
// QUEUE_NODE* newPtr;
// Statements
// if (!(newPtr = (QUEUE_NODE*)malloc(sizeof(QUEUE_NODE)))) return false;
QUEUE_NODE* newPtr = (QUEUE_NODE*)malloc(sizeof(QUEUE_NODE));
newPtr->dataPtr = itemPtr;
```

```
newPtr->next = NULL;
if (queue->count == 0)
// Inserting into null queue
queue->front  = newPtr;
else
queue->rear->next = newPtr;
(queue->count)++;
queue->rear = newPtr;
return true;
}   // enqueue

/*  ================= dequeue ================
This algorithm deletes a node from the queue. อัลกอริธึมนี้จะลบโหนดออกจากคิว
Pre  queue has been created   สร้างคิวก่อนแล้ว
Post   Data pointer to queue front returned and ตัวชี้ข้อมูลโพสต์ข้อมูลไปยังคิวที่ส่งกลับและ
front element deleted and recycled. ส่วนหน้าถูกลบและนำกลับมาใช้ใหม่
Return true if successful; false if underflow  กลับเป็นจริงถ้าสำเร็จ; เท็จถ้า underflow
*/
bool dequeue (QUEUE* queue, void** itemPtr)
{
//  Local Definitions
QUEUE_NODE* deleteLoc;
//  Statements
if (!queue->count)
return false;
*itemPtr  = queue->front->dataPtr;
deleteLoc = queue->front;
if (queue->count == 1)
// Deleting only item in queue ลบเฉพาะรายการในคิว
queue->rear  = queue->front = NULL;
else
queue->front = queue->front->next;
```

```
(queue->count)--;
free (deleteLoc);
return true;
}   // dequeue

/*  ================== queueFront ================== การนำสมาชิกออกจากคิว
This algorithm retrieves data at front of the queue อัลกอริธึมนี้จะเรียกข้อมูลที่ด้านหน้าของคิว
queue without changing the queue contents. คิวโดยไม่ต้องเปลี่ยนเนื้อหาคิว
Pre  queue is pointer to an initialized queue คิวก่อนเป็นตัวชี้ไปยังคิวเริ่มต้น
Post   itemPtr passed back to caller ไม่รู้
Return true if successful; false if underflow ย้อนกลับเมื่อสำเร็จ ไม่เป็นผลเมื่อ overflow
*/
bool queueFront (QUEUE* queue, void** itemPtr)
{
//  Statements
if (!queue->count)
return false;
else
{
*itemPtr = queue->front->dataPtr;
return true;
} // else
}   // queueFront การนำข้อมูลที่อยู่ตอนต้นของคิวมาแสดง

/*  ================== queueRear ================== การนำข้อมูลที่อยู่ตอนท้ายของคิวมาแสดง
Retrieves data at the rear of the queue  ดึงข้อมูลที่ด้านหลังของคิว
without changing the queue contents. โดยไม่ต้องเปลี่ยนเนื้อหาคิว
Pre  queue is pointer to initialized queue คิวก่อนเป็นตัวชี้ไปที่คิวเริ่มต้น
Post   Data passed back to caller ไม่รู้
Return true if successful; false if underflow ย้อนกลับเมื่อสำเร็จ ไม่เป็นผลเมื่อ overflow
*/
bool queueRear (QUEUE* queue, void** itemPtr)
```

```
{
//  Statements
if (!queue->count)
return true;
else
{
*itemPtr = queue->rear->dataPtr;
return false;
} // else
}   // queueRear

/*  ================== emptyQueue ================= คิวว่าง
This algorithm checks to see if queue is empty อัลกอริทึมนี้จะตรวจสอบว่าคิวว่างหรือไม่
Pre queue is a pointer to a queue head node คิวก่อนเป็นตัวชี้ไปยังโหนดหัวคิว
Return true if empty; false if queue has data
*/
bool emptyQueue (QUEUE* queue)
{
//  Statements
return (queue->count == 0);
}   // emptyQueue

/*  ================== fullQueue ================= คิวเต็ม
This algorithm checks to see if queue is full. It อัลกอริทึมนี้จะตรวจสอบเพื่อดูว่าคิวเต็มหรือไม่ มัน
is full if memory cannot be allocated for next node. เต็มหากหน่วยความจำไม่สามารถจัดสรรสำหรับโหนดถัดไปได้
Pre  queue is a pointer to a queue head node คิวก่อนเป็นตัวชี้ไปยังโหนดหัวคิว
Return true if full; false if room for a node กลับเป็นจริงถ้าเต็ม; เท็จไปห้องโหนด
*/
bool fullQueue (QUEUE* queue)
{
//  Check empty เช็คพื้นที่ว่าง
if(emptyQueue(queue)) return false; // Not check in heap
```

```
//  Local Definitions *
QUEUE_NODE* temp;
//  Statements
temp = (QUEUE_NODE*)malloc(sizeof(*(queue->rear)));
if (temp)
{
free (temp);
return false; // Heap not full ลำดับแบบฮีฟไม่เต็ม
} // if
return true; // Heap full
}   // fullQueue คิวเต็ม

/*  ================== queueCount ================  นับจำนวนคิว
Returns the number of elements in the queue. ส่งกลับจำนวนขององค์ประกอบในคิว
Pre  queue is pointer to the queue head node คิวก่อนเป็นตัวชี้ไปยังโหนดหัวคิว
Return queue count ย้อนกับไปนับ
*/
int queueCount(QUEUE* queue)
{
//  Statements
return queue->count;
}   // queueCount

/*  ================== destroyQueue ================ลบคิว
Deletes all data from a queue and recycles its ลบข้อมูลทั้งหมดออกจากคิวและรีไซเคิล
memory, then deletes & recycles queue head pointer. หน่วยความจำแล้วลบและรีไซเคิลตัวชี้หัวคิว
Pre  Queue is a valid queue คิวก่อนเป็นคิวที่ถูกต้อง
Post   All data have been deleted and recycled ข้อมูลทั้งหมดถูกลบและรีไซเคิลแล้ว
Return null pointer ย้อนกลับ พอยเตอร์
*/
QUEUE* destroyQueue (QUEUE* queue)
{
```

```
//  Local Definitions
QUEUE_NODE* deletePtr;
//  Statements
if (queue)
{
while (queue->front != NULL)
{
free (queue->front->dataPtr);
deletePtr   = queue->front;
queue->front = queue->front->next;
free (deletePtr);
} // while
free (queue);
} // if
return NULL;
}   // destroyQueue

/*  ==================== printQueue ================ ปริ้นคิว
A non-standard function that prints a queue. It is ฟังก์ชันที่ไม่ได้มาตรฐานที่พิมพ์คิว มันคือ
non-standard because it accesses the queue structures. ไม่ได้มาตรฐานเพราะเข้าถึงโครงสร้างคิว
Pre  queue is a valid queue คิวก่อนเป็นคิวที่ถูกต้อง
Post queue data printed, front to rear พิมพ์ข้อมูลคิวข้อมูลที่พิมพ์จากด้านหน้าไปด้านหลัง
*/
void printQueue(QUEUE* queue)
{
//  Local Definitions
QUEUE_NODE* node = queue->front;
//  Statements
printf ("Front=>");
while (node)
{
printf ("%3d", *(int*)node->dataPtr);
```

```
node = node->next;
} // while
printf(" <=Rear\n");
return;
}  // printQueue ปริ้นคิว
```

ตัวอย่างการประกาศ variable แบบ pointer ในภาษา Cview code : http://www.thaiall.com/datastructure/pointer.cpp ?

```
1   #include <stdio.h>
2   #include <conio.h>
3   typedef struct {
4   int mydata;
5   } DATA;
6   DATA* getdata (void);
7   //
8   void usedata (DATA* x);
9   //
10  main(void) {
11  clrscr(); // use conio.h
12  DATA* q1;
13  q1->mydata = 1;
14  int a; // variable
15  a =1;
16  int *b; // pointer
17  b = &a;
18  void* c; // nothing in c
19  c = &b;
20  void** d;
21  *d = (void *)&c;
22  usedata(q1); // 1
23  printf("value of a is %d \n", a); // 1
24  printf("a store in %p \n", (void *)&a); // 0013FF54
```

```
25  printf("value of b is %p \n", b); // 0013FF54
26  printf("b store in %p \n", (void *)&b); // 0013FF50
27  printf("value of q1->mydata is %d \n", q1->mydata); // 1
28  printf("value of c is %p \n", c); // 0013FF50
29  printf("c store in %p \n", (void *)&c); // 0013FF4C
30  printf("value of d is %d \n", d); // 4246996 (pointer of pointer)
31  printf("d store in %p \n", (void *)&d); // 0013FF48 (pointer of pointer)
32  }
33  //
34  void usedata (DATA* x) {
35  printf("%d \n", x->mydata);
36  }
```