程式設計第十二章

真理大學資工系 洪麗玲
llhung@mail.au.edu.tw

摘要

- 作業檢討
- 結構的進階變化➔鏈結串列
- 鏈結串列練習
- 作業

修改比較N個數字程式，將原數字與數字和當作一筆資料的兩個成員資料

3

12.2 自我參考結構

- 自我參考結構 (self-referential structure) 含有一個指向相同型態之結構的指標成員。例如，以下的定義

```
struct node {
   int data;
   struct node *nextPtr;
};
```

- 定義了struct node型別。struct node型別的結構具有兩個成員——整數成員data以及指標成員nextPtr。

- 將兩個自我參考結構的物件鏈結成一個串列 (list)。圖中的斜線表示空指標 (NULL pointer)，也就是放在第二個自我參考結構的鏈結成員，表示該鏈結不會再指向另一個結構。


```
struct sr
{
 int i;
 struct sr *p;
}
```

```
struct  sr  sr1, sr2;
 sr1.i =15;
sr1.p=&sr2;
sr2.i = 10;
sr2.p = NULL;
```

12.3 動態記憶體配置

- 建立和維護動態資料結構必須使用動態記憶體配置 (dynamic memory allocation)——這個功能讓程式在執行期間可以取得更多的記憶體空間來儲存新的節點，以及釋回不再需要的記憶體空間。
- 函式malloc和free，以及運算子sizeof是專門用於動態記憶體配置。

- 函式malloc的引數為所欲配置記憶體的位元組個數，它會傳回一個型別為void * 的指標 **(指向void的指標，pointer to void)**，該指標會指向配置的記憶體。

```
newPtr = malloc( sizeof( struct node ) );
```

- 計算sizeof(struct node)的值來判斷struct node結構所需要的位元組個數，接著在記憶體中配置一塊大小為sizeof(struct node)個位元組的新區域，然後將指向所配置記憶體的指標存放到變數newPtr。

- free函式會釋回記憶體空間，也就是將記憶體還給系統，使得之後有需要時可以再配置這塊記憶體。要釋放先前呼叫malloc動態配置的記憶體，可使用以下敘述

```
free( newPtr );
```

- C也提供calloc和realloc函式，用來建立及修改動態陣列。

產生的差異是....

```
struct XXX{
}
struct XXX A[50];
Or
scanf("%d", &a);
struct XXX A[a];
```

➔

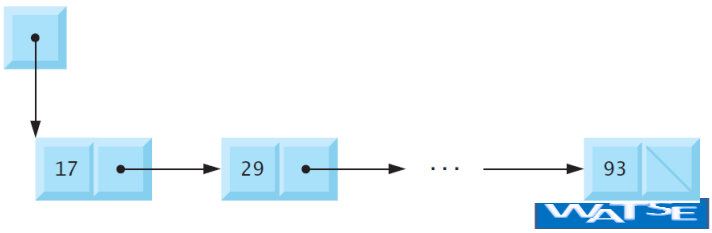
```
scanf("%d", &a);
for (i=0; i<a; i++)
    newPtr = malloc(sizeof(struct XXX));
```

???位址耶!

WATSE

12.4 鏈結串列

- 鏈結串列 (linked list) 是自我參考結構的線性集合，每個物件稱為節點 (node)，它們透過指標鏈結 (link) 串起來，因此稱為「鏈結」串列。我們可用指向串列第一個節點的指標來存取鏈結串列。接下來的節點則會使用儲存在每個節點的鏈結 (link) 指標成員來進行存取。依據慣例，串列最後一個節點的鏈結指標會設定為NULL，藉此表示此串列的結束



WATSE

鏈結串列的優勢

- 動態進行記憶體的配置
  - 不浪費
  - 不缺乏
- 操作更方便
  - 如前後順序的變換

宣告

```
struct listNode {
   int data; // each listNode contains a character
   struct listNode *nextPtr; // pointer to next node
}; // end structure listNode

typedef struct listNode ListNode;
// synonym for struct listNode
typedef ListNode *ListNodePtr;
// synonym for ListNode*
```

```
void insert( ListNodePtr *sPtr, int value )
{   ListNodePtr newPtr; // pointer to new node
   ListNodePtr currentPtr; // pointer to current node in list
   newPtr = malloc( sizeof( ListNode ) ); // create node
   if ( newPtr != NULL ) { // is space available
     newPtr->data = value; // place value in node
     newPtr->nextPtr = NULL; // node does not link to another node
     currentPtr = *sPtr; // setting the current pointer as *sPtr
    while ( currentPtr != NULL && currentPtr->nextPtr != NULL ) {
          currentPtr = currentPtr->nextPtr; // ... next node
     } // end while
     currentPtr->nextPtr = newPtr; // insert new node at the end of list
    } // end if
    else
       printf( "%c not inserted. No memory available.\n", value );
```



```
void printList( ListNodePtr currentPtr )
{
    // if list is empty
    if ( isEmpty( currentPtr ) ) {
      puts( "List is empty.\n" );   } // end if
    else {
      puts( "The list is:" );
      // while not the end of the list
      while ( currentPtr != NULL ) {
        printf( "%d --> ", currentPtr->data );
        currentPtr = currentPtr->nextPtr;   } // end while
      puts( "NULL\n" );
    } // end else
  } // end function printList
```

Delete(Part I)

```
int delete( ListNodePtr *sPtr, int value )
{
  ListNodePtr previousPtr; // pointer to previous node in list
  ListNodePtr currentPtr; // pointer to current node in list
  ListNodePtr tempPtr; // temporary node pointer
  if ( value == ( *sPtr )->data ) {       // delete first node
    tempPtr = *sPtr; // hold onto node being removed
    *sPtr = ( *sPtr )->nextPtr; // de-thread the node
    free( tempPtr ); // free the de-threaded node
    return value;   } // end if
```

…下頁待續

Delete(Part II)

```
else {
   previousPtr = *sPtr;
   currentPtr = ( *sPtr )->nextPtr;
   while ( currentPtr != NULL && currentPtr->data != value ) {
     previousPtr = currentPtr; // walk to ...
     currentPtr = currentPtr->nextPtr; // ... next node
    } // end while
   if ( currentPtr != NULL ) {// delete node at currentPtr
    tempPtr = currentPtr;
     previousPtr->nextPtr = currentPtr->nextPtr;
     free( tempPtr );
     return value;      } // end if
 } // end else
```


本週練習

- 請修改比較N個數字程式，除了將原數字與數字和當作一筆資料的兩個成員資料另外再加上一指標成員，程式中先將輸入的資料存成一鏈結串列，使其中的串列資料是按照資料和由大到小的順序。