Topic 9: Pointer to pointer

call by value

```
int main (void)
{
    int a=10;
    int b=20;
    swap (a,b);
    printf("a=%d, b=%d", a,b);
    return 0;
}

void swap (int aa, int bb)
{
    int temp;
    temp = aa;
    aa = bb;
    bb = temp;
}
```

aa and bb are the copies of a and b

→swap not work

call by address

```
int main (void)
{
    int a=10;
    int b=20;
    swap (&a, &b);
    printf("a=%d, b=%d", a,b);
    return 0;
}

void swap (int *aa, int *bb)
{
    int temp;
    temp = *aa;
    *aa = *bb;
    *bb = temp;
}
```

pass the container of a and b to swap()

aa and bb point to the memory of a and b

*aa and *bb exchange the contents of a and b

→swap works

Can allocate?

```
#include <stdio.h>
#include <stdlib.h>

void Allocate (int *p)
{
    p = (int *) malloc (sizeof(int));
}

int main (void)
{
    int *p;

    p = NULL;
    Allocate(p);

    if (p == NULL)
        printf ("allocate NOT OK\n");
    else
        printf ("allocate OK\n");
    return 0;
}
```

Pointer to pointer **p

- Common use to redirect a pointer to another space

```
in main.c

int main (void)
{
    int a = 10;
    int *p;
    p = &a;
    printf(" *p=%d ", *p);

    changeP(&p);
    printf(" *p=%d ", *p);
}
```

variable of **a**

variable of **b**

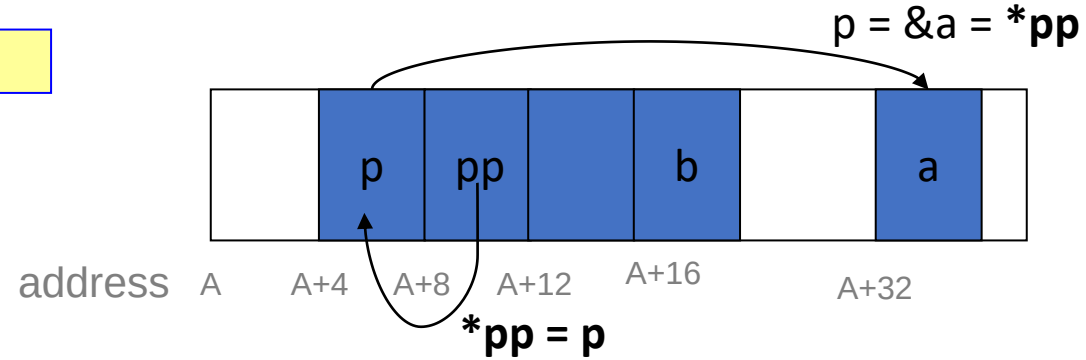
```
in change.c

int b = 100;

void changeP (int **pp)
{
    *pp = &b;
    **pp = 1000;
}
```

Direct to a reference → use one *

The type in the reference is a pointer → use another *



Pointer & structure & linked list & pointer to pointer

```
    pStudent = AddStudent (pStudent, 20, 40);
    pStudent = AddStudent (pStudent, 52, 100);

    pStudent = head;
    while (pStudent != NULL)
    {
        printf("ID: %d with score: %d \n",
            pStudent->ID, pStudent->score);
        pStudent = pStudent -> next;
    }
    return 0;
}
```

```
tReg * AddStudent (tReg *p, int ID, int score)
{
    p->next = (tReg *) malloc (sizeof(tReg));
    p->next->ID = ID;
    p->next->score = score;
    p->next->next = NULL;

    return (p->next);
}
```

OLD fashion

```
    AddStudent (&pStudent, 20, 40);
    AddStudent (&pStudent, 52, 100);

    pStudent = head;
    while (pStudent != NULL)
    {
        printf("ID: %d with score: %d \n",
            pStudent->ID, pStudent->score);
        pStudent = pStudent -> next;
    }
    return 0;
}
```

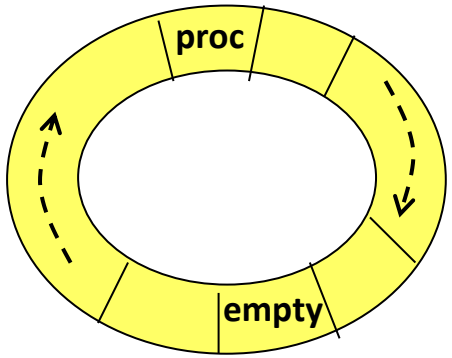
```
void * AddStudent (tReg **p, int ID, int score)
{
    (*p)->next = (tReg *) malloc (sizeof(tReg));
    (*p)->next->ID = ID;
    (*p)->next->score = score;
    (*p)->next->next = NULL;

    *p = (*p)->next;
}
```

NEW fashion

A use case of pointer to pointer

- In an embedded system
  - Most system resources are predefined
  - To dynamic allocate memory is time-consuming
- Solution
  - Take a pre-allocated space as a ring buffer
  - Use pointer to pointer to return a pre-allocated space for a process



in receiveSdu.c

```
int receiveSdu ()
{
    char *p;
    getSpaceForSdu(&p);
    copySduFromHW(p);
}
```

Callback or ISR

in bufManager.h

```
#define SIZE 100
typedef char Buf[1024];

typedef   struct   tBufSt
{
    int empty;
    int processing;
    Buf buf[SIZE];
} tSduBuf;
```

Ring buffer implementation

in bufManager.c

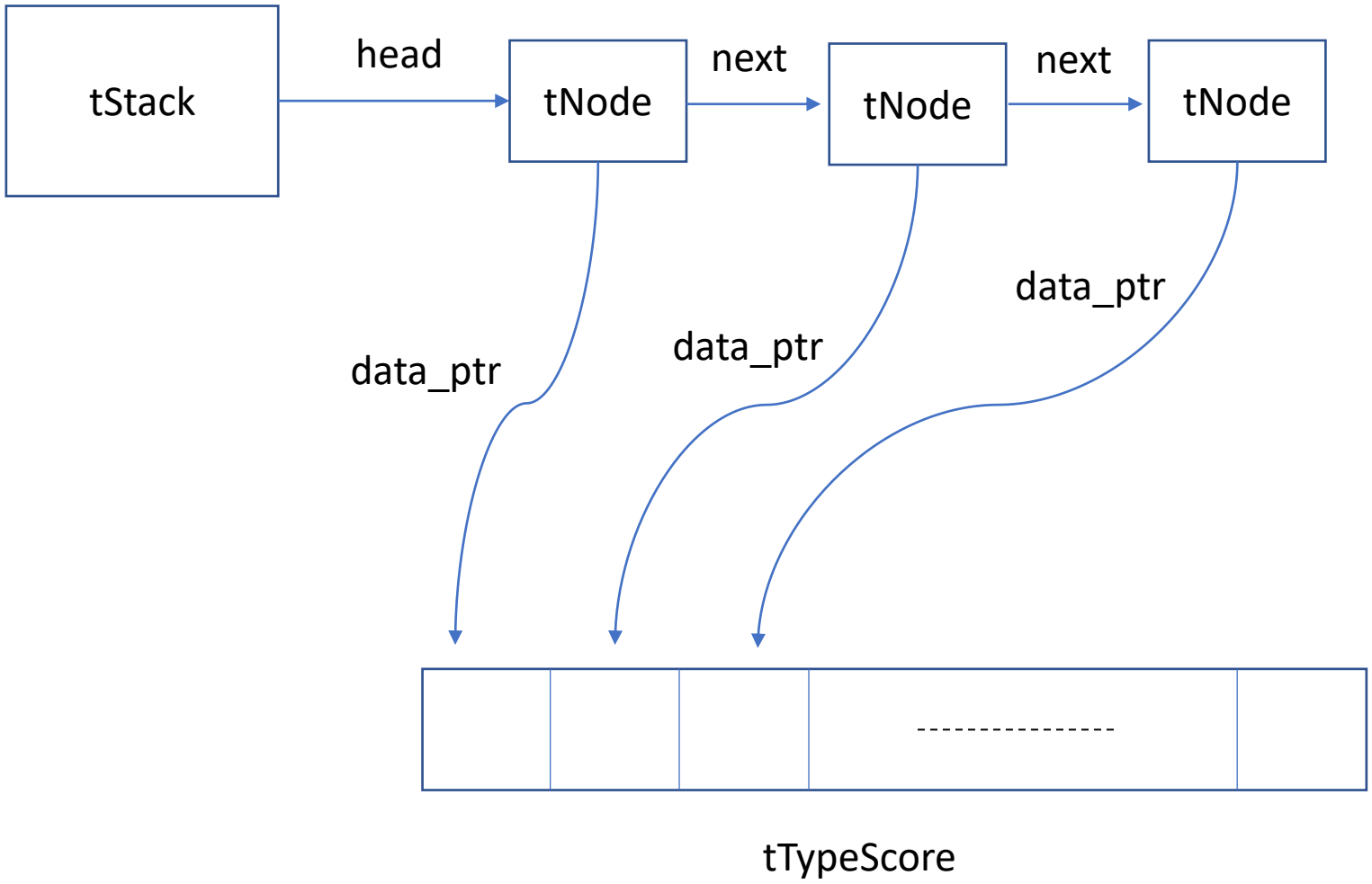
```
tSduBuf pMem;

void getSpaceForSdu (int **pp)
{
    *pp = &pMem.buf[empty];
     empty = (empty + 1) % SIZE
}
```

Ring buffer usage

W11-on-site assignment

- Use the provided stack.c to implement your own stack API
- Use pointer to pointer to allocate memory
- You can have only two malloc in your entire program
  - One is for allocating stack header
  - The other is for allocating stack body
- See the attachments


tStack
head
tNode
next
tNode
next
tNode
data_ptr
data_ptr
data_ptr
tTypeScore

```
(1) push or (2) pop a item to/from stack: 2
  [Error] handlePopOperation(): nothing in stack

   printStackContent(): stack items ->

(1) push or (2) pop a item to/from stack: 1
  handlePushOperation(): enter score value: 1
     getScoreSpace(): giving space numbered 0

   printStackContent(): stack items -> 1(0)

(1) push or (2) pop a item to/from stack: 1
  handlePushOperation(): enter score value: 2
     getScoreSpace(): giving space numbered 1

   printStackContent(): stack items -> 2(1) 1(0)

(1) push or (2) pop a item to/from stack: 1
  handlePushOperation(): enter score value: 3
     getScoreSpace(): giving space numbered 2

   printStackContent(): stack items -> 3(2) 2(1) 1(0)

(1) push or (2) pop a item to/from stack: 1
  handlePushOperation(): enter score value: 4
     getScoreSpace(): giving space numbered 3

   printStackContent(): stack items -> 4(3) 3(2) 2(1) 1(0)
```

```
(1) push or (2) pop a item to/from stack: 1
  handlePushOperation(): enter score value: 5
     getScoreSpace(): giving space numbered 4

   printStackContent(): stack items -> 5(4) 4(3) 3(2) 2(1) 1(0)

(1) push or (2) pop a item to/from stack: 1
  [Error] handlePushOperation(): space full

   printStackContent(): stack items -> 5(4) 4(3) 3(2) 2(1) 1(0)

(1) push or (2) pop a item to/from stack: 6

   printStackContent(): stack items -> 5(4) 4(3) 3(2) 2(1) 1(0)

(1) push or (2) pop a item to/from stack: 2
  handlePopOperation(): poped value: 5
     returnScoreSpace(): return space numbered 4

   printStackContent(): stack items -> 4(3) 3(2) 2(1) 1(0)

(1) push or (2) pop a item to/from stack: 2
  handlePopOperation(): poped value: 4
     returnScoreSpace(): return space numbered 3

   printStackContent(): stack items -> 3(2) 2(1) 1(0)
```

```
(1) push or (2) pop a item to/from stack: 2
  handlePopOperation(): poped value: 3
    returnScoreSpace(): return space numbered 2

   printStackContent(): stack items -> 2(1) 1(0)

(1) push or (2) pop a item to/from stack: 1
  handlePushOperation(): enter score value: 2
    getScoreSpace(): giving space numbered 2

   printStackContent(): stack items -> 2(2) 2(1) 1(0)

(1) push or (2) pop a item to/from stack: 2
  handlePopOperation(): poped value: 2
    returnScoreSpace(): return space numbered 2

   printStackContent(): stack items -> 2(1) 1(0)

(1) push or (2) pop a item to/from stack: 2
  handlePopOperation(): poped value: 2
    returnScoreSpace(): return space numbered 1

   printStackContent(): stack items -> 1(0)

(1) push or (2) pop a item to/from stack: 2
  handlePopOperation(): poped value: 1
    returnScoreSpace(): return space numbered 0

   printStackContent(): stack items ->
```

```
(1) push or (2) pop a item to/from stack: 2
  [Error] handlePopOperation(): nothing in stack

   printStackContent(): stack items ->

(1) push or (2) pop a item to/from stack: ^C
```