Intermediate Programming

— Stacks and Queues —

Waseda Univ.

Today's Topics

- Dynamic memory allocation
- The malloc function
- Stacks
- Queues

Dynamic memory allocation

To declare an array of `double` type, we need to specify the size of the array. For example,

```
double x[3];
```

- This deeply depends on the explicit size (here is 3) of the array.
- Modification of the program, recompiling, and another execution are necessary when the size is changed.

Let's write a program that is independent from the size of the array.

- The malloc( ) function enables the dynamic memory allocation, i.e., the malloc( ) function allocates the field of memory that is necessary for declaring the array.

Dynamic memory allocation

malloc(size)

- The malloc() function allocates "size" bytes of memory and returns a pointer to the allocated memory.
- If there is an error, it returns a NULL pointer.
- The free() function frees allocations that were created via the preceding malloc( ) functions.
- The users should include stdlib.h to use malloc( ) function.

Dynamic memory allocation

Example

```
double *x;
x = (double *)malloc(N*sizeof(double));
```

- The malloc() function allocates fields of memory that are N times the size of `double` type. Then the pointer to the allocated memory is assigned to `x`.

- sizeof $\cdots$ returns size in bytes of several types,
  ex. sizeof(double) : 8bytes, sizeof(int) : 4bytes, sizeof(char) : 1byte
- Cast operator $\cdots$ converts a variable to the explicit type data.
  ex. int a=1, b=2; a/b $\Rightarrow$ 0, (double)a/b $\Rightarrow$ 0.5

Dynamic memory allocation

Example

```
double *x;
x = (double *)malloc(N*sizeof(double));
```

- If there is an error in allocating memory, we exit the program.
- After using the allocated memory, we should free the allocations.

Example (Exceptions and release of allocated memory)

```
x = (double *)malloc(N*sizeof(double));
if(x==NULL){
    printf("Can't allocate memory.\n");
    exit(1);
}

/* After using the allocated memory, */
free(x);
```

Dynamic memory allocation

Example

```
double *x;
x = (double *)malloc(N*sizeof(double));
```

⇒ The pointer x behaves like a pointer to the array of N `double` elements. So that x is the same as the name of the array.

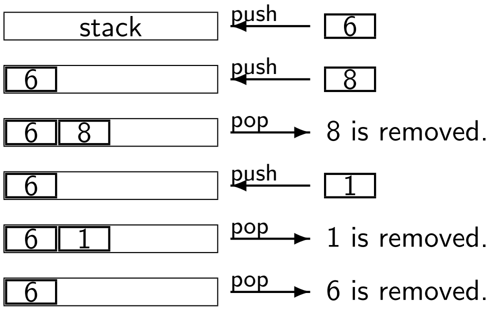
- x+i : points to the i-th element of the array.
- *(x+i) : access the i-th element of the array (same as $x[i]$).

Example of using the malloc() function

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

int main(void){
  int N;
  int *x;

  printf("Input N:");
  scanf("%d",&N);

  x = (int *) malloc(sizeof(int)*N);
  if(x==NULL){
        printf("Can’t allocate memory.\n");
        exit(1);
  }

  for(i=0;i<N;i++) x[i]=i; // The same as the array of N double variables.

  for(i=0;i<N;i++) printf("x[%d]=%d\n",i,x[i]);

  free(x);
  return 0;
}
```

Stack

- Stack is an abstract data type that stores a collection of elements. It is also called LIFO (last in, first out).
- There are two principal operations:
  - push: add an element to the collection.
  - pop: removes the last added element.
- Only the last element comes out of the collection.



Implementation

Stack can be easily implemented through an array by members:

- Array,
- The size of the array (stack),
- The number of items.

These compose the structure named stack.

- By using `typedef`, one can omit "`struct structure_tag`" statement to declare structure variables.

```
struct Stack{
  double *Data; /* Array */
  int Size;     /* Size of array */
  int Count;    /* Num. of items */
};
```

```
typedef struct{
  double *Data; /* Array */
  int Size;     /* Size of array */
  int Count;    /* Num. of items */
} Stack;    /* Define stack type */
```

CreateStack function

To create a stack structure,

- allocate memory for the stack structure,
- allocate memory for the array member of the stack structure,
- assign the size of the stack,
- set the number of items 0.

The allocation of the stack must be done before that of the array member.

```
Stack *CreateStack(int size) {
  Stack *s = (Stack*)malloc(sizeof(Stack));
  s->Data = (double*)malloc(sizeof(double) * size);
  s->Size = size;
  s->Count = 0;
  return s;
}
```

Note: In the above program, the error exception of the malloc function is omitted. This should be done by using the mallocx function that is defined in the next page.

The malloc function with error exception

```
void *mallocx(int size) {
  void *p = malloc(size);
  if (p == NULL) {
    printf("cannot allocate memory\n");
    exit(1);
  }
  return p;
}
```

First we define the mallocx function. Then we use the mallocx function (instead of malloc function) when the memory allocation is needed.

DisposeStack function

To dispose a stack structure,

- free allocations for the array member,
- free allocations for the stack structure.

The stack structure must be free after the array member is released.

```
void DisposeStack(Stack *s) {
  free(s->Data);
  free(s);
}
```

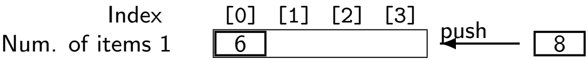
Add an element to stack

PushStack function

Add a given element to the collection of the stack structure.

- The push operation adds an element and increments the number of items (`s->Count`).

```
void PushStack(Stack *s, double x) {
  s->Data[s->Count] = x;
  s->Count++;
}
```



- If the stack is full and does not contain enough space to accept an element to be pushed, the stack is considered to be stack overflow. If the overflow occurs, the function exits the program with printing the error statement.

```
  if (s->Count == s->Size) {
    printf("stack overflow\n");
    exit(1);
  }
```

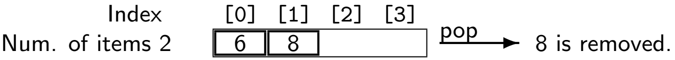
Removes an element from stack

PopStack function

Removes the last added element from the stack structure.

- The pop operation removes the last added element of the stack and decrements the number of items (`s->Count`).

```
double PopStack(Stack *s) {
  s->Count--;
  return s->Data[s->Count];
}
```

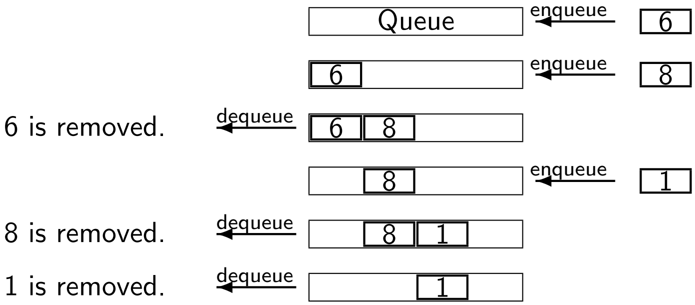


- If the stack is empty, it goes into stack underflow state. This means no items are present in stack to be removed. If the underflow occurs, the function exits the program with printing the error statement.

```
if (s->Count == 0) {
  printf("stack underflow\n");
  exit(1);
}
```

Queue

- Queue is an abstract data type that also stores a collection of elements. It is FIFO (first in, first out) data structure.
- There are two principal operations:
  - enqueue: add an element to the collection.
  - dequeue: removes the first added element.
- The first element added to the queue will be the first one to be removed.



Implementation

Queue can be implemented through an array by members:

- Array,
- The size of the array (queue),
- The number of items,
- The index of first added position.

These compose the structure named queue.

```
typedef struct{
  double *Data;  /* Array */
  int Size;      /* Size of array */
  int Count;     /* Num. of items */
  int Index;     /* Index of first position */
} Queue;
```

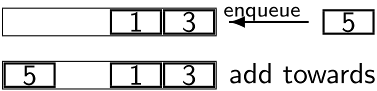
Circular buffer

Difficulty

Fixed length arrays are limited in capacity. For example, there is no capacity of items to add 5 in the below.

Circular buffer

Consider the array member of queue as a closed circle. Then the element 5 is added towards the head of the queue.



add towards the head of queue.

The simple way of turning the array into a closed circle is to use the remainder %. If n is the size of the array, then compute indices modulo n.

CreateQueue function

To create a queue structure,

- allocate memory for the queue structure,
- allocate memory for the array member of the queue structure,
- assign the size of the queue,
- set the number of items 0,
- set the index of first position 0 (or anywhere in the array).

The allocation of the stack must be done before that of the array member.

```
Queue *CreateQueue(int size) {
  Queue *q = (Queue*)mallocx(sizeof(Queue));
  q->Data = (double*)mallocx(sizeof(double) * size);
  q->Size = size;
  q->Count = 0;
  q->Index = 0;
  return q;
}
```

DisposeQueue function

To dispose a queue structure,

- free allocations for the array member of the queue structure,
- free allocations for the stack structure itself.

The queue structure must be free after the array member is released.

```
void DisposeQueue(Queue *q) {
  free(q->Data);
  free(q);
}
```

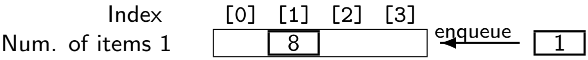
Add an element to queue

Enqueue function

Add a given element to the collection of the queue structure.

- The enqueue operation adds an element and increments the number of items (s->Count).

```
void Enqueue(Queue *q, double x) {
  q->Data[(q->Index + q->Count) % q->Size] = x;
  q->Count++;
}
```



- If the queue is full, the queue is considered to be queue overflow. If the overflow occurs, the function exits the program with printing the error statement.

```
  if (q->Count == q->Size) {
    printf("queue overflow\n");
    exit(1);
  }
```

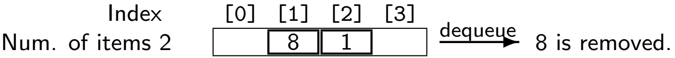
Removes an element from queue

Dequeue function

Removes an element in the first added position from the queue structure.

- The dequeue operation removes an item from the first added position of the queue and decrements the number of items.

```
double Dequeue(Queue *q) {
  int i = q->Index;
  q->Count--;
  q->Index = (q->Index + 1) % q->Size;
  return q->Data[i];
}
```



- If the queue is empty, it is said queue underflow. If the underflow occurs, the function exits the program with printing the error statement.

```
  if (q->Count == 0) {
    printf("queue underflow\n");
    exit(1);
  }
```

Summary

- Dynamic memory allocation
- The malloc function
- Stacks
- Queues