目录

Abstract data types

An abstract data type (ADT) is a data type that

- is defined by the operations that may be performed on it
- the data in the ADT is accessible only via operations
- a user of an ADT cannot access or even see the data (it's hidden)

In the specification of an ADT, we must:

- define the data in some way
- define a set of functions to manipulate the data

wikipedia definition of ADTs

In mathematics:

- below is a 'signature' of an ADT for STACK, which uses the format:
    - 'operator': 'type' x 'type' --> 'type'

```
createstack:                --> STACK
push    : STACK x INTEGER --> STACK
pop     : STACK           --> INTEGER | {undefined}
empty   : STACK           --> BOOLEAN
```

- this signature specifies the STACK data type completely

In programming

- the 'signature' above is called an **interface**
- there is also an **implementation** of the **interface**
- and there is the **client** that uses the **interface**

Terminology

- an ADT is accessed (only) through its **interface**
- an **interface** is a contract between the **client** and the **implementation**
- an ADT 'architecture' is:

```
client <=> interface <=> implementation
```

  - an **interface** is a set of **prototypes** that specifies the ADT
    - e.g. a *header file* such as `fredADT.h`
  - a **client** uses the interface to carry out a task
    - the client contains the *main()* function
    - e.g. a client could be called `client.c`
  - the **implementation** is the actual ADT
    - e.g. it could be called `fredADT.c`
  - specifically, in terms of the architecture, we have:

```
client.c <=> fredADT.h <=> fredADT.c
```

Importance of ADTs:

- *abstraction*: hides complexity and detail from a client
- *flexibility*: change the implementation without changing the client
- *decomposition*: decompose the problem into 'component' tasks
- *structure*: structuring for readability and maintainability
- *security*: restricts the ability of hostile clients corrupting data

A Point data type

Consider the following data type for a Cartesian point, defined in *point.h*

切换行号显示

```
// point.h
#include <math.h>
typedef struct point Point;
struct point{
   float x; float y;
};
float distance(Point, Point);  // returns distance between two points
void move(Point, float, float);// moves a point by a certain amount

```

and its implementation

切换行号显示

```
// point.c: an implementation of a point
#include "point.h"
float distance(Point a, Point b) {
   float dx = a.x - b.x;
   float dy = a.y - b.y;
   return sqrt(dx*dx + dy*dy);
}
void move(Point *a, float dx, float dy) {
    a->x += dx;
```

```
        a->y += dy;
    }
```

and its client

切换行号显示

```
 1    // clientPoint.c: a client of the point data type
 2    #include <stdio.h>
 3    #include <stdlib.h>
 4    #include "point.h"
 5    int main(int argc, char *argv[]) {
 6       Point r, s;
 7       float d;
 8
 9       r.x = 5;
10       r.y = 5;
11       s.x = 4;
12       s.y = 6;
13       d = distance(r, s);
14       printf("1st distance: %0.1f\n", d);
15       move(&r, -1, +2);
16       d = distance(r, s);
17       printf("2nd distance: %0.1f\n", d);
18       return EXIT_SUCCESS;
19    }
```

You compile the 'implementation' and 'client' by:

```
prompt$ dcc point.c clientPoint.c
prompt$ ./a.out
1st distance: 1.4
2nd distance: 1.0
```

Is Point an Abstract Data Type? NO, because:

- the declaration of *Point* is visible to the client
  - even worse, *clientPoint.c* sets the values inside the type (in lines 9-12)
  - *Point* is not abstract: it forms part of the code

Point ADT

Here is a *Point* ADT:

切换行号显示

```
// pointADT.h: an interface for the Point ADT
typedef struct point *Point;      // notice that Point is a pointer to a struct
float distance(Point, Point);     // returns distance between two points
void  move(Point, float, float); // moves a point by certain amount
Point create(float, float);       // create a new Point

```

and an implementation of the interface

切换行号显示

```c
// pointADT.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "pointADT.h"

struct point {
      float x;
      float y;
};

Point create(float xpos, float ypos) {
   Point p;
   p  = malloc(sizeof(struct point)); // returns a pointer to the struct
   if (p == NULL) {
      fprintf(stderr, "No memory\n");
      exit(1);
   }
   p->x = xpos;
   p->y = ypos;
   return p;
}

float distance(Point a, Point b) {
   float dx = a->x - b->x;
   float dy = a->y - b->y;
   return sqrt(dx*dx + dy*dy);
}
void move(Point a, float dx, float dy) {
   a->x += dx;
   a->y += dy;
}
```

and to finish up, a client that uses the *Point* ADT

切换行号显示

```c
// clientPointADT.c: a client of the Point ADT
#include <stdio.h>
#include <stdlib.h>
#include "pointADT.h"

int main() {
    Point r, s;
    float d;

    r = create(5,5);
    s = create(4,6);
    d = distance(r,s);
    printf("1st distance: %0.1f\n", d);
    move(r, -1, +2);
    d = distance(r,s);
    printf("2nd distance: %0.1f\n", d);
    return EXIT_SUCCESS;
}
```

- note the design is much cleaner
- we cannot see inside *struct point* pointed to by *Point*

We compile using:

```
dcc pointADT.c clientPointADT.c
```

The output is the same as for the previous version of *Point*.

Stacks and queues

Stacks mimic many real-world behaviours:

- moving boxes
- putting plates away in the cupboard and then setting the table
- placing and removing many rings on a (single) finger
- putting on more than one pair of socks because it is cold and then taking them off
- many cars going down a one-way street that is blocked and having to back out again
- calling functions in C
- ... all are examples of *LAST IN, FIRST OUT (LIFO)*
    - ***notice that in LIFO, you add and remove from the same end***

Queues in fact are even more common-place than stacks:

- the checkout at a supermarket
- people standing at a ticket window
- people queueing to go onto a bus
- cars queueing to go onto a ferry
- objects flowing through a pipe (where they cannot overtake each other)
- messages on an answering machine
- ... all are examples of *FIRST IN, FIRST OUT (FIFO)*
    - ***notice, you add to one end, remove from the other***

Example of a stack (with top-of-stack on the left):

Operation	Resulting Stack	Return Value
push(1)	1	
push(2)	2 1	
push(3)	3 2 1	
pop()	2 1	3
push(4)	4 2 1	

How do we implement and use a stack?

- We can implement a *stack* using:
    - an array
        - fast, easiest to implement (usually fixed length)
    - a linked list
        - slow, but easier on memory (memory is added only when needed)

An ADT for a stack may require a small *library* of functions:

- create a stack
- make the stack empty (i.e. destroy it)
- push data onto the stack
- pop data off the stack,
- is the stack empty
- get the height of the stack
- print or show the contents of a stack
- print or show the top element on the stack

- ...

Normally, need to create an ADT for a stack, and another ADT for a queue

- instead in this course, create an ADT called **Quack** that does both

Quack

Quack interface

The interface is below:

切换行号显示

```
// Quack.h: an interface definition for a queue/stack
#include <stdio.h>
#include <stdlib.h>

typedef struct node *Quack;

Quack createQuack(void);     // create and return Quack
Quack destroyQuack(Quack);   // remove the Quack
void  push(int, Quack);      // put int on the top of the quack
void  qush(int, Quack);      // put int at the bottom of the quack
int   pop(Quack);            // pop and return the top element on the quack
int   isEmptyQuack(Quack);   // return 1 is Quack is empty, else 0
void  makeEmptyQuack(Quack);// remove all the elements on Quack
void  showQuack(Quack);      // print the contents of Quack, from the top down

```

- note I have dropped the *ADT* suffix off the name

Quack implementation

Most popular and efficient is to use an array:

- the first element in the array is the bottom of the stack
- *push* puts an element at the top
- *pop* takes an element from the top

The array 'grows' as the elements are pushed onto the quack

- normally the size of the array is fixed, to *HEIGHT* say
- the top is just an index in the array
    - **underflow** occurs when you *pop* and the top = -1 (= empty quack)
    - **overflow** occurs when you call *push* and the top = HEIGHT (= full quack)

切换行号显示

```
// Quack.c: an array-based implementation of a quack
#include "Quack.h"

#define HEIGHT 1000

struct node {
   int array[HEIGHT];
   int top;
};
```

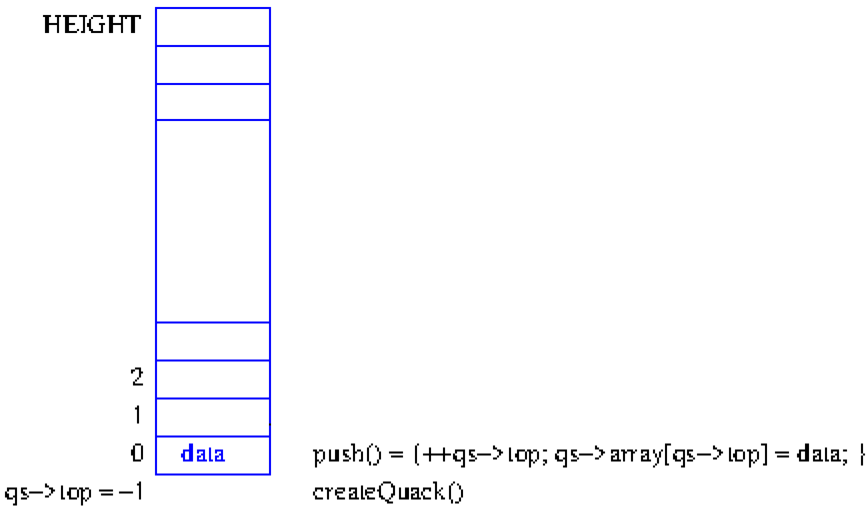
```c

Quack createQuack(void) {
   Quack qs;
   qs = malloc(sizeof(struct node));
   if (qs == NULL) {
      fprintf (stderr, "createQuack: no memory, aborting\n");
      exit(1); // should pass control back to the caller
   }
   qs->top = -1;
   return qs;
}

void push(int data, Quack qs) {
   if (qs == NULL) {
      fprintf(stderr, "push: quack not initialised\n");
   }
   else {
      if (qs->top >= HEIGHT-1) {
         fprintf(stderr, "push: quack overflow\n");
      }
      else {
         ++qs->top;
         qs->array[qs->top] = data;
      }
   }
   return;
}

void qush(int data, Quack que) { // adds data to the bottom of the array

   // code not shown

   return;
}


int pop(Quack qs) { // return top element, or 0 if error
   int retval = 0;
   if (qs == NULL) {
      fprintf(stderr, "pop: quack not initialised\n");
   }
   else {
      if (isEmptyQuack(qs)) {
         fprintf(stderr, "pop: quack underflow\n");
      }
      else {
         retval = qs->array[qs->top]; // top element on stack
         --qs->top;
      }
   }
   return retval;
}

void makeEmptyQuack(Quack qs) {
   if (qs == NULL) {
      fprintf(stderr, "makeEmptyQuack: quack not initialised\n");
   }
   else {
      while (!isEmptyQuack(qs)) {
         pop(qs);
      }
   }
   return;
}

Quack destroyQuack(Quack qs) {
   if (qs == NULL) {
      fprintf(stderr, "destroyQuack: quack not initialised\n");
```

```
    }
    free(qs);
    return qs;
}

int isEmptyQuack(Quack qs) {
    int empty = 0;
    if (qs == NULL) {
        fprintf(stderr, "isEmptyQuack: quack not initialised\n");
    }
    else {
        empty = qs->top < 0;
    }
    return empty;
}

void showQuack(Quack qs) {
    if (qs == NULL) {
        fprintf(stderr, "showQuack: quack not initialised\n");
    }
    else {
        printf("Quack: ");
        if (qs->top < 0) {
            printf("<< >>\n");
        }
        else {
            int i;
            printf("<<");                    // start with a <<
            for (i = qs->top; i > 0; --i) {
                printf("%d, ", qs->array[i]); // print each element
            }
            printf("%d>>\n", qs->array[0]);   // last element includes a >>
        }
    }
    return;
}
```

- note I have dropped the *ADT* suffix off the name to match the interface

Below we see a picture of an array-based quack. Notice,

- the fixed height
- *createQuack()* initialises the top-of-quack index to -1
- *push()* increments the top-of-stack index and places data at that location in the array
- *pop()* returns the element at the top-of-stack and decrements the index



Summarising, we have an ADT consisting of **quack.h** and **quack.c**

Quack clients

Client 1. black-box unit tester

'Black-box' testing is

- a method of testing a 'unit' from the outside
- the tester is just a *client*
- it can call only interface functions
  - this severely limits what can be tested
  - sometimes (often) 'secret' functions are included in ADTs that provide more information
    - including extra functions for BB testing is usually called 'instrumentation'
      - they are often security holes!
- (note 'white-box' testing inserts code (into the ADT) to verify correct behaviour)

One aim of BB testing is to verify every error message

- *Can you generate every error message?*
  - often errors are not handled 'gracefully':
    - with an *exit(1)* for example

BB testing can only ever test some of the behaviours

切换行号显示

```
// blackbox.c: black box unit tester for a quack
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "Quack.h"

int main(int argc, char *argv[]){
    Quack s = NULL;
    Quack t;

    printf("Test #1: Init stack is empty: ");
    s = createQuack();
    assert(isEmptyQuack(s));
    printf("passed\n");

    printf("Test #2: Push, stack not empty: ");
    push(1, s);
    assert(!isEmptyQuack(s));
    printf("passed\n");

    printf("Test #3: Push again, pop twice, then empty: ");
    push(2, s);
    pop(s);
    pop(s);
    assert(isEmptyQuack(s));
    printf("passed\n");

    printf("Test #4: TESTING ERROR: quack underflow\n");
    pop(s);
    return EXIT_SUCCESS;
}
```

```
prompt$ dcc Quack.c blackbox.c
prompt$ ./a.out
Test #1: Init stack is empty: passed
Test #2: Push, stack not empty: passed
```

```
Test #3: Push again, pop twice, then empty: passed
Test #4: TESTING ERROR: quack underflow
pop: quack underflow
```

Note:

- the quack is an *abstract* structure
  - the client cannot see how the quack has been implemented
- error handling is problematical:
  - *what does a client do with errors that are returned?*
  - *what is a 'serious' error, what is a 'usage' error?*
  - *how do you test that every error message is correct?*

Client 2. reversing a string

Pushing the characters in a string onto a stack and then popping them off again until the stack is empty will reverse the order.

- for example: the string "abcde"
  - push('a'); push('b'); ...push('e'); will result in the stack

    ```
    <<e,d,c,b,a>>
    ```

    where *e* is at the top
  - pop(); pop(); ... pop(); will result in 'e' being popped, then 'd', etc

切换行号显示

```c
// revarg.c: reverse the chars in the first command-line argument
#include <stdio.h>
#include <stdlib.h>
#include "Quack.h"

int main(int argc, char *argv[]) {
   Quack s = NULL;

   if (argc >= 2) {
      char *inputc = argv[1];
      s = createQuack();
      while (*inputc != '\0') {
         push(*inputc++, s);
      }
      while (!isEmptyQuack(s)) {
         printf("%c", pop(s));
      }
      putchar('\n');
   }
   return EXIT_SUCCESS;
}
```

Compiling and running:

```
prompt$ dcc Quack.c revarg.c
prompt$ ./a.out stressed
desserts
```

Client 3. a postfix calculator

Reminder:

- **infix notation**: *A * (B + C) / D*
- **prefix notation**: */ * A + B C D* (also called *Polish notation*)
- **postfix notation**: *A B C + * D /* (also called *reverse Polish notation*)

Given an expression in *postfix* notation, return its value

- for example, evaluating the following postfix expression:

  ```
  2 1 3 + 2 5 * * 7 + *
  ```

  means

  ```
  (2 (((1 3 +) (2 5 *) *) 7 +) *)
  ```

  which equals 94

How do we program this? Using a stack, and reading each character.

- when we get an operator character:
  - **pop** the top 2 elements off the quack
  - operate (i.e. '+' or '*') on the two elements
  - **push** the result back onto the quack
- when we get a 'number', **push** it onto the quack ...
  - but the number may be many digits long
    - need to translate this string to a number
      - a *while-loop* reads each digit:
        - converts it to its numerical value
        - adds it to 10 * previous value

切换行号显示

```
// postfix.c
// a calculator for command-line postfix expressions
// reports an error if the expression contains anything but +, *, integer, space, tab
#include <stdio.h>
#include <stdlib.h>
#include "Quack.h"

#define PLUSCHAR '+'
#define MULTCHAR '*'

int main(int argc, char *argv[]) {
  Quack s = NULL;
  int error = 0;
  int operandFound = 0;

  if (argc >= 2) {
     char *inputc = argv[1];
     s = createQuack();
     while (*inputc != '\0') {
        int sum;
        switch (*inputc) {
        case PLUSCHAR: push(pop(s) + pop(s), s);
                   break;
        case MULTCHAR: push(pop(s) * pop(s), s);
                   break;
        case '0':
        case '1':
        case '2':
        case '3':
```

```
         case '4':
         case '5':
         case '6':
         case '7':
         case '8':
         case '9': operandFound = 1;
                   sum = 0;
                   while ((*inputc >= '0') && (*inputc <= '9')) {
                      sum = 10 * sum + (*inputc - '0'); // notice char arithmetic!
                      inputc++;
                   }
                   push(sum, s);
                   inputc--; // the while-loop reads one too many
                   break;
         case ' ':
         case '\t':
                   break;
         default:  fprintf(stderr, "Invalid character %c\n", *inputc);
         }
         inputc++;
      }
      if (operandFound) {
         if (!isEmptyQuack(s)) {   // stack must contain the result
            printf("%d\n", pop(s));
         }
         else {
            fprintf(stderr, "Error: stack empty, no result\n");
            error = 1;
         }
      }
      if (!isEmptyQuack(s)) {      // stack must now be empty
         fprintf(stderr, "Error: extra operand(s)\n");
         error = 1;
      }
      if (error) {
         return EXIT_FAILURE;
      }
   }
   return EXIT_SUCCESS;
}
```

If we want to use the array version of quack, compile using:

```
prompt$ dcc Quack.c postfix.c
prompt$ ./a.out "2 3 +"
5
prompt$ ./a.out "2 4 5 + *"
18
prompt$ ./a.out "1 2 3 4 5 * + * +"
47
prompt$ ./a.out ""
prompt$ ./a.out
prompt$
```

What's the program doing?

- operands are operated on as they are put onto the quack
- e.g., quack contents for "1 2 3 4 5 * + * +":

```
push 1 <<1>>
push 2 <<2,1>>
push 3 <<3,2,1>>
push 4 <<4,3,2,1>>
push 5 <<5,4,3,2,1>>
```

```
oper * <<20,3,2,1>>
oper + <<23,2,1>>
oper * <<46,1>>
oper + <<47>>
```

Stacks versus queues

If the data structure is a stack, then a *push* puts an element onto the **top**:

Operation	Resulting Stack	Return Value
push(1)	1	
push(2)	2 1	
push(3)	3 2 1	
push(4)	4 3 2 1	
pop()	3 2 1	4
pop()	2 1	3
pop()	1	2
pop()		1

where the stack read left-to-right is top-to-bottom.

If the data structure is a queue, then a 'push' puts an element onto the **bottom**:

Operation	Resulting Queue	Return Value
push(1)	1	
push(2)	1 2	
push(3)	1 2 3	
push(4)	1 2 3 4	
pop()	2 3 4	1
pop()	3 4	2
pop()	4	3
pop()		4

You can see that the numbers are popped off in reverse order.

The difference between a *stack push* and a *queue push* is which end is used

- in a stack, a *push* and *pop* both work at the top
- in a queue, a *pop* takes from the head (i.e. top), a *queue push* adds to the tail (i.e. bottom)

We'll call a *queue push* a **qush** from now on:

- a **qush** inserts an element at the bottom of the data structure
- in contrast to a **push**, which inserts an element at the top

The same *pop* is used for a stack and a queue:

Some operations do not know or care whether the ADT is a stack or a queue ...

- *createQuack*
- *isEmptyQuack*
- *showQuack*

If you want to model a stack in an application, *qush* must not be used

If you want to model a queue in an application, *push* must not be used

In real-life, *qush* and *push* can be mixed to model behaviour

- e.g. a supermarket queue, sometimes someone at the tail of the queue gets served first

The function qush() has not been implemented yet in the ADT

- to implement it we need to add a function *qush()* to *quack.c*
- and add its prototype to *quack.c*

This is an exercise for this week.

Client 4. separate stack and queue

Here is a simple program that mixes a stack and a queue

- both are declared as *quacks*
  - but one uses *push* and *pop*
  - the other uses *qush* and *pop*

切换行号显示

```
// separateQuack.c: have both a stack and a queue in the same program
#include <stdio.h>
#include "Quack.h"

int main(void) {
   Quack s = NULL;
   Quack q = NULL;

   s = createQuack();
   q = createQuack();

   push(1, s);
   push(2, s);
   printf("pop from s produces %d\n", pop(s));
   printf("pop from s produces %d\n", pop(s));

   qush(1, q);
   qush(2, q);
   printf("pop from q produces %d\n", pop(q));
   printf("pop from q produces %d\n", pop(q));

   return EXIT_SUCCESS;
}
```

Assuming we have implemented *qush*, compiling and executing:

```
prompt$ dcc Quack.c separateQuack.c
prompt$ ./a.out
pop from s produces 2
pop from s produces 1
```

```
pop from q produces 1
pop from q produces 2
```

Notice that the pushed/qushed integers are popped off in opposite order in the two data structures.

Client 5. mixed stack and queue

You can also mix 'pushes' and 'qushes' on the one quack:

切换行号显示

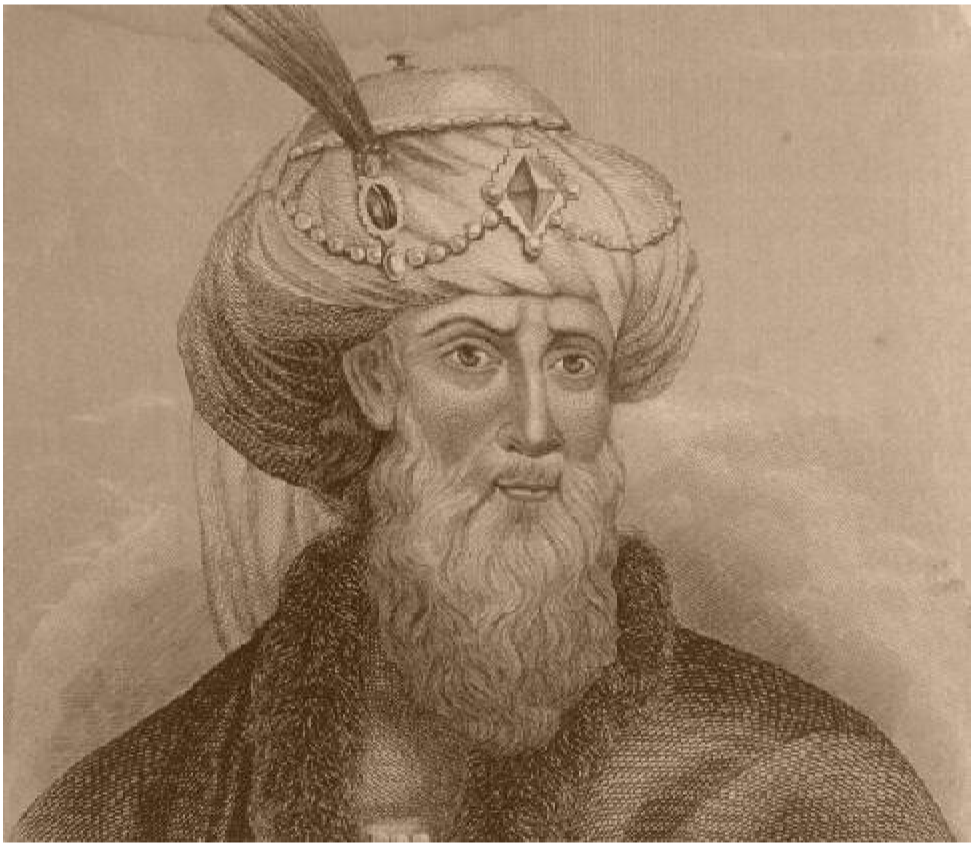
```
// mixedQuack.c: mix qush and push in a quack
#include <stdio.h>
#include "Quack.h"

int main(void) {
   Quack s = NULL;

   s = createQuack();

   printf("push 1 and 2\n");
   push(1, s);
   push(2, s);
   printf("qush 3 and 4\n");
   qush(3, s);
   qush(4, s);
   showQuack(s);
   printf("pop produces %d\n", pop(s));
   printf("pop produces %d\n", pop(s));
   printf("pop produces %d\n", pop(s));
   printf("pop produces %d\n", pop(s));
   return EXIT_SUCCESS;
}
```

- ... interesting, but maybe hard to think of an application

Client 6. 'circular' queue

This is an application that uses a queue only.

This is the 1st century Jewish historian/philosopher/mathematician **Flavius Josephus**



In the book:

- *Matters Mathematical* (Chelsea Publishing, 1978) by *Herstein and Kaplansky* there is a story/legend about *Flavius Josephus*
  - *Flavius was in a group of **10** people that were surrounded by the Romans*
  - *the group decided that they would rather die than surrender*
  - *Flavius suggested forming a ring, and going around the ring clockwise repeatedly*
    - *... killing every **3rd** person*
    - *... until just one person remained*
    - *... who would kill himself*
  - *Flavius placed himself at position **4***

```
          2   3
        1       4 <-- Flavius
       10        5
        9       6
          8   7
```

Starting at 1, you eliminate every 3rd person, people would be removed in the order:

```
3 6 9 2 7 1 8 5 10
```

The last remaining person was the 4th, Flavius.

This strategy can be implemented using a queue.

切换行号显示

```
// Josephus.c: use a queue to simulate a ring of n people, and
// eliminate every mth person until there is just a single person
```

```
remaining
#include <stdio.h>
#include <stdlib.h>
#include "Quack.h"

int main(int argc, char *argv[]) {
   Quack q = NULL;

   int n, m;
   if ((argc != 3) ||
       (sscanf(argv[1], "%d", &n) != 1) ||
       (sscanf(argv[2], "%d", &m) != 1)) {
       fprintf (stderr, "Usage: %s total eliminate\n", argv[0]);
       return EXIT_FAILURE;
   }
   q = createQuack();
   int i;
   for (i=1; i<=n; i++) {   // populate the queue
      qush(i, q);           // top = '1' and bottom = 'n'
   }
   showQuack(q);
   int person=0;
   while (!isEmptyQuack(q)) { // continue until empty
      for (i=0; i<m-1; i++) { // skip m-1 people
         qush(pop(q), q);     // move from front to back
      }
      person = pop(q);        // if this person ...
      if (!isEmptyQuack(q)) { // ... is not last one ...
         printf("byebye %d\n", person); // eliminate him
      }
   }
   printf("%d is the only person left\n", person);
   return EXIT_SUCCESS;
}
```

Compiling and executing with arguments 10 and 3:

```
prompt$ dcc Quack.c Josephus.c
prompt$ ./a.out 10 3
Quack: <<1, 2, 3, 4, 5, 6, 7, 8, 9, 10>>
goodbye 3
goodbye 6
goodbye 9
goodbye 2
goodbye 7
goodbye 1
goodbye 8
goodbye 5
goodbye 10
Quack << >>
4 is the only person left
```

Th << >> notation shows the contents of the queue initially and at the end.

Notice:

- how the queue was initialised
- how people are 'skipped'
    - a person is 'popped' then 'qushed', which moves him from the head to the tail (keeps them alive)
- **the queue itself is not circular, the behaviour is circular**
    - the program loops until the queue is empty
    - after skipping *m* people, the next person is 'popped'
        - if that person

        - is not the last, they stay 'popped', and print *goodbye*
        - is the last, the quack must be empty, and the loop terminates with this person

'Reality' check

- Actually there were 39 soldiers in the group, every 7th was eliminated, and Flavius was 17th
- Flavius decided not to commit suicide
- The Romans were impressed by Flavius
- He ended up joining the Romans, legend has it

The dilemma that *Josephus* had 2000 years ago, children can have every day!

- when children need to choose, or decide who will get something, they often use rhymes
- example 1
    - with a flower that has *n* petals, pick the petals and repeat until there is just one petal left:

    ```
    she loves me,
    she loves me not,
    she loves me,
    she loves me not,
    ...
    ```

    - this is *Josephus ring* solution with *m=2*
- example 2
    - go around a group of *n* children looking for someone to remove from the group:

    ```
    1    2     3    4
    EEny MEEny MIny MOE
    5       6     7      8
    CATCH a TIger BY the TOE
    9     10      11      12
    IF he HOLlers LET him GO
    13   14    15   16
    EEny MEEny MIny MOE
    17   18  19  20
    'O' 'U' 'T' SPELLS
     21
    OUT!
    ```

    - *Josephus ring* is being used: *m=21*

ADTs (2019-07-19 22:30:37由AlbertNymeyer编辑)