CS101 Computer Programming and Utilization

Milind Sohoni

June 17, 2006

1. So far

2. Queues-Introduction

The story so far ...

- functions
- file handling
- structs
- Srirang's problem
- Classes

This week...

Queues

A practical problem

- Gulmohar has a limited number of seating (say 10).
- If a seat is empty, then a guest may occupy it.
- However, if there is no seat empty, the guest should form a queue outside.

How is this queue implemented?

- The queue is two operations:
  - pop pulls out the first person in the queue.
  - push name registers the person to be in the queue.
- It is assumed that the order of exiting the queue is the same as joining.

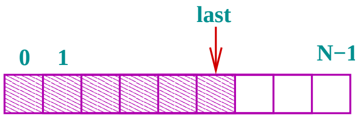
A practical problem

- Gulmohar has a limited number of seating (say 10).
- If a seat is empty, then a guest may occupy it.
- However, if there is no seat empty, the guest should form a queue outside.

How is this queue implemented?

- The queue is two operations:
    - pop pulls out the first person in the queue.
    - push name registers the person to be in the queue.
- It is assumed that the order of exiting the queue is the same as joining.

The queue may be implemented as an array:



- We estimate that there will be no more than N people in the queue.
- The queue is then an array of names, say list.
- The first is list[0] and the last is list[last].
- push and pop are easily implemented.

Qarray.cpp

```
const int N=5;
struct entry
{
  char name[7];
};
class Q
{
  private:
    entry list[5];
    int last;
  public:
    void init(void);
    // initializes the queue
    int push(entry);
    // pushes an entry on Q
    entry pop(void);
    // returns the first entry
};
```

- Here N is fixed to be 5.
- Q is a class:
  - list stores the list of entrys.
  - last stores the location of the last entry in the list.
- The class functions are typical. Here is init:

Qarray.cpp

```
const int N=5;
struct entry
{
  char name[7];
};
class Q
{
  private:
    entry list[5];
    int last;
  public:
    void init(void);
    // initializes the queue
    int push(entry);
    // pushes an entry on Q
    entry pop(void);
    // returns the first entry
};
```

- Here N is fixed to be 5.
- Q is a class:
  - list stores the list of entrys.
  - last stores the location of the last entry in the list.
- The class functions are typical. Here is init:

```
void Q::init(void)
{
  last=-1;
}
```

class functions

```
int Q::push(entry ee)
{
 if (last==N-1)
 {
   return(1);
 }
 else
 {
   list[last+1]=ee;
   last=last+1; return(0);
 };
}
entry Q::pop(void)
{
  entry ee;
  ee=list[0];
  for (int i=0;i<last;i=i+1)
    list[i]=list[i+1];
  last=last-1; return(ee);
}
```

Whats happening:

- push: if the last entry is N-1, then Q is full; return 1 (error).
- push: Otherwise append the entry after last and update it.
- pop: first, return the first entry in the list, i.e., list[0].
- pop: Next, move all elements one step left.

The main program

What is the main program? It is to test the following input:

```
1 ace
1 king
-1
-1
1 queen
1 jack
1 ten
1 nine
-1
-1
0
```

- 1 ace means push ace.
- -1 means a pop
- 0 means shut this program.
- The program should give a trace:

```
[sohoni@nsl-13 talk14]$ ./a.out
push  ace
push  king
pop   ace
pop   king
push  queen
push  jack
push  ten
push  nine
pop   queen
pop   jack
done
```

Structure of the main program

- Initialize the Q.
- while option != 0 do
  - If option==1, read in name and push.
  - If option==-1, pop the Q.
  - If option==0 do nothing.
- endwhile;

```
int main()
{
  entry ee; Q QQ;
  QQ.init(); int option=1;
  WHILE code HERE
  cout << "done\n";
}
```

Structure of the main program

- Initialize the Q.
- while option != 0 do
  - If option==1, read in name and push.
  - If option==-1, pop the Q.
  - If option==0 do nothing.
- endwhile;

```
int main()
{
  entry ee; Q QQ;
  QQ.init(); int option=1;
  WHILE code HERE
  cout << "done\n";
}
```

```
while (option!=0)
{
  cin >> option;
  if (option==1)
  {
    cin >> ee.name;
    cout << "push  " << ee.name
    h=QQ.push(ee);
    if (h==1)
    {
      cout << "error \n";
      option=0;
    };
  };
  if (option==-1)
  {
    ee=QQ.pop();
    cout << "pop   "<< ee.name
  };
};
```

The output again

```
1 ace
1 king
-1
-1
1 queen
1 jack
1 ten
1 nine
-1
-1
0
```

- 1 ace means push ace.
- -1 means a pop
- 0 means shut this program.
- The program should give a trace:

```
[sohoni@nsl-13 talk14]$ ./a.out
push  ace
push  king
pop   ace
pop   king
push  queen
push  jack
push  ten
push  nine
pop   queen
pop   jack
done
```

Problems?

- Well, we havent really implemented pop properly: pop on an empty queue should be an error.
- When the number in the Q exceeds N, then there is an error.
- A pop on a Q takes $O(n)$-time. We need to move the entries.

Problems?

- Well, we havent really implemented pop properly: pop on an empty queue should be an error.
- When the number in the Q exceeds N, then there is an error.
- A pop on a Q takes $O(n)$-time. We need to move the entries.

Solutions:

- Implement pop correctly.
- Make N large.

Problems?

- Well, we havent really implemented pop properly: pop on an empty queue should be an error.
- When the number in the Q exceeds N, then there is an error.
- A pop on a Q takes $O(n)$-time. We need to move the entries.

Solutions:

- Implement pop correctly.
- Make N large.
- Wasteful.

Problems?

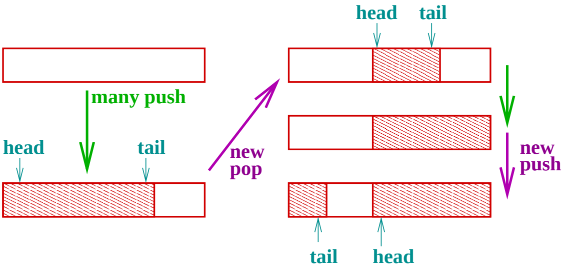
- Well, we havent really implemented pop properly: pop on an empty queue should be an error.
- When the number in the Q exceeds N, then there is an error.
- A pop on a Q takes $O(n)$-time. We need to move the entries.

Solutions:

- Implement pop correctly.
- Make N large.
- Wasteful.

There is actually an array implementation which does not move elements.
This is called the circular queue implementation.
Two new variables:

- head: the first element.
- tail: the last element.



Implement `cicrularQarray.cpp`.

Static and Dynamic Memory allocation

- So far, all our variables and their sizes were declared up-front.
- This means that we can estimate the memory requirement of your program even before the program has started running.

Static and Dynamic Memory allocation

- So far, all our variables and their sizes were declared up-front.
- This means that we can estimate the memory requirement of your program even before the program has started running.
- This seems to be the essential bottle-neck for implementing a queue where there is no bound on the length.
- C++ allows this: Dynamic Data Structures

Static and Dynamic Memory allocation

- So far, all our variables and their sizes were declared up-front.
- This means that we can estimate the memory requirement of your program even before the program has started running.
- This seems to be the essential bottle-neck for implementing a queue where there is no bound on the length.
- C++ allows this: Dynamic Data Structures

Implement the following requirement:

- A long list and *increasing* list is to be maintained. The length of this list is not predictable.
- The program should readin in inputs of the type:

```
1 ashank
2 vibha
0
```

- 1 ashank: add ashank to the list.
- 2 vibha: check if vibha is in the list.
- 0: end the session.

Static and Dynamic Memory allocation

A popular technique of implementing dynamic data structures is through the use of Pointers. Recall:

```
struct entry
{
  char name[7];
};
```

Here is a pointer:

```
entry *w;
```

This says that w is a *pointer* to a data-item of type *entry*.

Our first objective will be to create long lists using pointers. A pointer is declared using the *-notation.

classname *PointerVariableName

This declares PointerVariableName as the address of a location which stores an entity of the type `classname`.

A *looong* list

Let us create a very long list of *entrys*.

```
struct Qentry
{
  entry field;
  Qentry *next;
};
```

This creates a structure which has a *field* to store the data, and *next* which *points* to a similar Qentry.

A *looong* list

Let us create a very long list of *entrys*.

```
struct Qentry
{
  entry field;
  Qentry *next;
};
```

This creates a structure which has a *field* to store the data, and *next* which *points* to a similar Qentry.

```
Qentry *w,*head;
head->field=firstentry;
head->next=NULL;
while (cond)
{
  w=new Qentry;
  w->field=newentry();
  w->next=head;
  head=w;
};
```

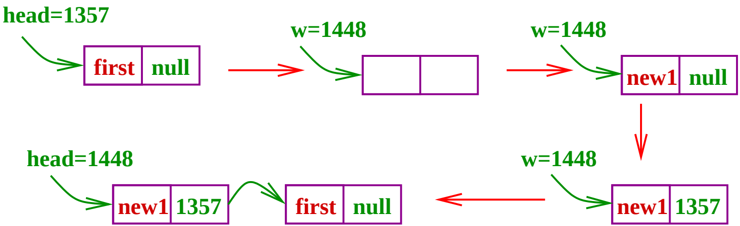
A *looong* list

Let us create a very long list of *entrys*.

```
struct Qentry
{
  entry field;
  Qentry *next;
};
```

This creates a structure which has a *field* to store the data, and *next* which *points* to a similar Qentry.

```
Qentry *w,*head;
head->field=firstentry;
head->next=NULL;
while (cond)
{
  w=new Qentry;
  w->field=newentry();
  w->next=head;
  head=w;
};
```

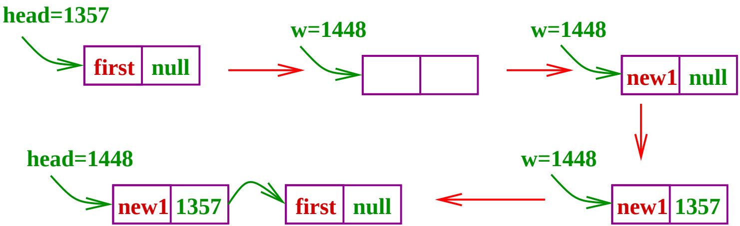


A *looong* list

What happens is:

- The statement `w=new entry` creates a *template*, i.e., storage of the type `Qentry` with junk entries.
- These fields are accessed by `w->....`
- Once correctly set, we have created a *network* of data items.

```
Qentry *w,*head;
head->field=firstentry;
head->next=NULL;
while (cond)
{
  w=new Qentry;
  w->field=newentry();
  w->next=head;
  head=w;
};
```

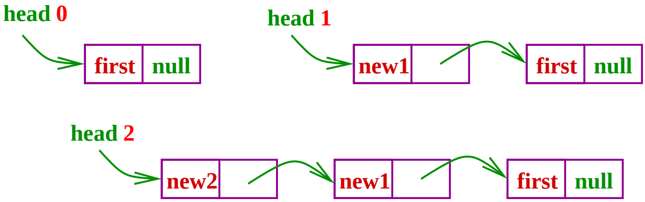


A *looong* list

What happens is:

- The statement `w=new entry` creates a *template*, i.e., storage of the type `Qentry` with junk entries.
- These fields are accessed by `w->....`
- Once correctly set, we have created a *network* of data items.

```
Qentry *w,*head;
head->field=firstentry;
head->next=NULL;
while (cond)
{
  w=new Qentry;
  w->field=newentry();
  w->next=head;
  head=w;
};
```



How do I search?

```
Qentry *head, *runner;
entry field0, currfield;
runner=head;
currfield=runner->field;
int found=0;
while ((runner!=NULL)&&
{                (found==0))
   currfield=runner->field;
   if (currfield==field0)
        found=1;
   runner=runner->next;
};
return (found);
```

- The program needs a head which is a pointer to the head of the list.
- Next, it needs field0 which is the field to be searched.
- It maintains a runner which goes from the head of the list to the tail until field0 is found.
- This is done by the statement:

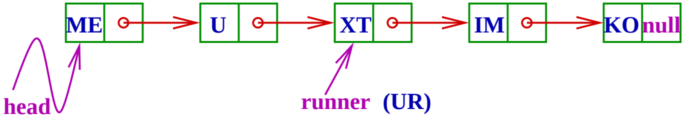
  runner=runner− >next;

How do I search?

```
Qentry *head, *runner;
entry field0, currfield;
runner=head;
currfield=runner->field;
int found=0;
while ((runner!=NULL)&&
{                 (found==0))
   currfield=runner->field;
   if (currfield==field0)
        found=1;
   runner=runner->next;
};
return (found);
```

- The program needs a head which is a pointer to the head of the list.
- Next, it needs field0 which is the field to be searched.
- It maintains a runner which goes from the head of the list to the tail until field0 is found.
- This is done by the statement:

runner=runner− >next;



Queues again

- 1 ace means push ace.
- -1 means a pop
- 0 means shut this program.

```
1 ace
1 king
-1
-1
1 queen
1 jack
1 ten
1 nine
-1
-1
0
```

```
[sohoni@nsl-13 talk14]$ ./a.out
push  ace
push  king
pop   ace
pop   king
push  queen
push  jack
push  ten
push  nine
pop   queen
pop   jack
done
```

We want...

No LIMITS on how long the queue can get!

The classes

```
struct Qentry
{
  entry field;
  Qentry *next;
};
class Q
{
  private:
    Qentry *head, *tail;
  public:
    void init(void);
    // initializes the queue
    int push(entry);
    // pushes entry onto queue
    entry pop(void);
    // returns the first entry
};
```

- Our old implementation had an array of entry.
- Now, instead, we have a Qentry with a pointer.
- head points to the head of the Q, while tail points to the last entry.
  - entry leaves from the head, but
  - comes in at the tail.
- The class interface remains the same. This means that the old main program will still work!

The functions

```
void Q::init(void)
{
  head=NULL; tail=NULL;
}
int Q::push(entry ee)
{
  Qentry *w;
  w=new Qentry;
  w->field=ee;
  w->next=NULL;
  if (head==NULL)
  {
    head=w; tail=w;
  }
  else
  {
    tail->next=w;
    tail=w;
  };
  return(0);
```

The functions

```
void Q::init(void)
{
  head=NULL; tail=NULL;
}
int Q::push(entry ee)
{
  Qentry *w;
  w=new Qentry;
  w->field=ee;
  w->next=NULL;
  if (head==NULL)
  {
    head=w; tail=w;
  }
  else
  {
    tail->next=w;
    tail=w;
  };
  return(0);
```

- init is nothing. Set head, tail to NULL.
- push has two cases:
  - When the Q is empty and a new element is to be added.
  - When the Q is non-empty.
- Both cases are easy.

The functions

```
void Q::init(void)
{
  head=NULL; tail=NULL;
}
int Q::push(entry ee)
{
  Qentry *w;
  w=new Qentry;
  w->field=ee;
  w->next=NULL;
  if (head==NULL)
  {
    head=w; tail=w;
  }
  else
  {
    tail->next=w;
    tail=w;
  };
  return(0);
```

- init is nothing. Set head, tail to NULL.
- push has two cases:
  - When the Q is empty and a new element is to be added.
  - When the Q is non-empty.
- Both cases are easy.
- If head is NULL $\longrightarrow$ make w the head, tail.
- If head exists $\longrightarrow$ append to the tail, and modify it.

```
entry Q::pop(void)
{
  entry ee; Qentry *dum;
  if (head==NULL)
    cout << "error\n";
  if (head==tail)
  {
    ee=head->field;
    delete(head);
    head=NULL;tail=NULL;
  }
  else
  {
    ee=head->field;
    dum=head;
    head=head->next;
    delete(dum);
  };
  return(ee);
}
```

- pop is simple as well except for the delete function.
- delete(pointerVar); returns the memory location back from the program to the system.
- If head is NULL, error.
- If head==tail then there is only one element, so the Q becomes empty.
- Else, everything is normal:
  - Remove the head entry, and update the head.

```
entry Q::pop(void)
{
  entry ee; Qentry *dum;
  if (head==NULL)
    cout << "error\n";
  if (head==tail)
  {
    ee=head->field;
    delete(head);
    head=NULL;tail=NULL;
  }
  else
  {
    ee=head->field;
    dum=head;
    head=head->next;
    delete(dum);
  };
  return(ee);
}
```

- pop is simple as well except for the delete function.
- delete(pointerVar); returns the memory location back from the program to the system.
- If head is NULL, error.
- If head==tail then there is only one element, so the Q becomes empty.
- Else, everything is normal:
  - Remove the head entry, and update the head.
- **Note how delete is used.**

Summary

- Pointers enable us to request and release memory for our use.
- They enable us to create intricate data-structures with great conceptual ease.
- The main functions are new, delete.
- For a program using pointers, it CANNOT be predicted how much memory it will use.
- If we dont delete what we dont need, then that is called a MEMORY LEAK.

Assignment

Two lists of students exist in two files `db1.txt` and `db2.txt`. Using pointers, prepare a list of students which exist on both lists. In other words, compute the intersection.