Massey University
COLLEGE OF SCIENCES

159201

Week 2

Summer 2014

Te Kunenga
ki Pūrehuroa

L05

Te Kunenga
ki Pūrehuroa

Dynamic X static

Static storage:

- does not change size during the running of the program
- must guess the size which may be too big or too small
- random access
- fast
- simple

Dynamic storage:

- changes size during the running of the program
- always exactly the right size
- sequential access
- slower
- a bit more complicated

Stacks based on Arrays

A stack is an **Abstract Data Type (ADT)**

It is a pile of things...

Stack operations:

Push – place a new item on top of the stack

Pop – remove the top item from the stack

Top – look at the top item in the stack

IsEmpty – returns true if the stack is empty

Why do we use them?

Stacks have a useful characteristic: **LIFO**

(***L***ast item ***I***n is ***F***irst item ***O***ut)

Stacks based on Arrays

We can use stacks whenever we need LIFO, e.g., calls to recursive functions.

A stack is an ADT, independent of the implementation.

Any programming language with any data structure can be used.

```
class Stack {
        private:
          float data[100];
          int index;
        public:
          Stack();
          ~Stack();
          void Push(float newthing);
          void Pop();
          float Top();
          bool isEmpty();
};
```

Massey University
COLLEGE OF SCIENCES

Stack Constructor/Destructor

```
Stack::Stack() { //Note: no data type in front
// constructor
  index = -1;
}


Stack::~Stack() {
// destructor
}
```

Te Kunenga
ki Pūrehuroa

Stack Functions

```
void Stack::Push(float newthing) {//new thing on top of the stack
  index++;
  data[index] = newthing;//Warning: watch for overflow
}

void Stack::Pop() {// remove the top item from the stack
  if (index > -1) { index--; } //Takes care of underflow
}

float Stack::Top() {//return value of the top item in the stack
  return data[index]; //Warning: what if stack is empty?
}
bool Stack::isEmpty() {// return true if the stack is empty
  if (index < 0) { return true; }
  return false;
}
```

Stack Push/Pop etc

```
int main() {
  Stack A, B;  //This is how to declare a stack
  A.Push(62.3);
  A.Push(2.4);
  B.Push(47.1);
  A.Pop();
  if (A.isEmpty()) {
     printf("Stack A is empty");
  } else {
     float x = A.Top();
     printf("Top item in A is %1.1f", x);
  }
}
```

Te Kunenga ki Pūrehuroa

Brief introduction to Object Oriented Programming: Classes

Classes: templates for objects

It usually contains *members*:

- Properties (variables)
- Methods (“functions”)

Brief introduction to Object Oriented Programming: Properties

The properties of an instance of an object has to be stored somewhere.

In OO:

- These properties have different scopes
- Accessing/modifying properties often uses a method that belong to the class
- Data can be private or public

Brief introduction to Object Oriented Programming: Methods

Set of procedures, or functions (in C++ at least...)

In OO:

- Methods are part of the class
- To call the “function” one has to call the method that belong to the class
- A method can also be private or public

Brief introduction to Object Oriented Programming: Scope

Private X Public

Private: only accessible through the class own methods

Public: accessible by anyone anywhere in the code (not exactly, this is an oversimplification of the scope issue).

Massey University
COLLEGE OF SCIENCES

Brief introduction to Object Oriented Programming:
Constructors and Destructors

In C++ we use Constructors and Destructors

These are a bit more than simple allocation and de-allocation of memory

They can also be used for initialisation of properties

*Although in this paper we use classes, the full OO approach with C++ is covered by **159234***

Here we use classes in a simplistic way.

Te Kunenga
ki Pūrehuroa

L06

Stacks based on Linked-Lists

Remember, a Stack is an **Abstract Data Type (ADT)**

Independent of the implementation

We saw how to implement a Stack based on arrays (static)

Stacks can grow in size, so a better option is to use dynamic allocation

No problems with accessing stacks, you only need to access them by the top (no random access)

Stacks based on Linked-Lists

A stack is an **Abstract Data Type (ADT)**

It is a pile of things...

Stack operations:

- **Push** – place a new item on top of the stack
- **Pop** – remove the top item from the stack
- **Top** – look at the top item in the stack
- **IsEmpty** – returns true if the stack is empty

Why do we use them?

Stacks have a useful characteristic: **LIFO**

(***L***ast item ***I***n is ***F***irst item ***O***ut)

Te Kunenga ki Pūrehuroa

Stacks based on Linked-Lists

Apart from Node struct, almost no changes.

```
struct Node {
  float data;
  Node *next;
};
class Stack {
        private: //the data differs from the array
          Node *listpointer;
        public:
          Stack();
          ~Stack();
          void Push(float newthing);
          void Pop();
          float Top();
          bool isEmpty();
};
```

Stack Constructor/Destructor

Constructor is different ...

```
Stack::Stack() { //Note: no data type in front
// constructor
  listpointer = NULL;
}


Stack::~Stack() {
// destructor
}
```

Stacks based on Linked-Lists

```
void Stack::Push(float newthing) {
  //newthing on top of the stack
  Node *temp;
  temp = new Node; //same as add node to front of linked-list
  temp->data = newthing;
  temp->next = listpointer; //NOTE: no overflow problem
  listpointer = temp;
}
void Stack::Pop() {// remove the top item from the stack
  Node *p;
  p = listpointer;
  if (listpointer != NULL) { //check to avoid underflow
     listpointer = listpointer->next;
     delete p; //always delete a TEMPORARY variable
  }
}
```

Stacks based on Linked-Lists

```
float Stack::Top() {//return value of the top item in the stack
  return listpointer->data;//WARNING: what if listpointer is NULL?
}

bool Stack::isEmpty() {// return true if the stack is empty
  if (listpointer ==NULL ) {
        return true;
  }
  return false;
}
```

Stacks based on Linked-Lists

Main() is the same as before...

```
int main() {
   Stack A, B;  //This is how to declare a stack
   A.Push(62.3);
   A.Push(2.4);
   B.Push(47.1);
   A.Pop();
   if (A.isEmpty()) {
      printf("Stack A is empty");
   } else {
      float x = A.Top();
      printf("Top item in A is %1.1f", x);
   }
}
```

Stack implementation

Compare the codes based on arrays and LL

Question: what is the best way to implement a stack in C/C++?

Clues: *we don't need random access, and we don't know the size before running the code*

Stack implementation Problems

If we have

```
Stack A, B;
```

Does `A = B;` works??

Use a method to do that instead (say `A.Copy(B)`)

How about `A == B;` ??

It is better to have another method (`A.Compare(B)`)

Why?

Stack implementation Problems

Answer:

It has all to do with pointers!

When you copy or compare stacks, you are just copying or comparing *pointers*, not different *stacks*.

You need to copy or compare *values* instead.

e.g., you could have identical values for A and B, but they are pointing to different places.

So, `if ( A == B)` could be FALSE even with identical stacks

If you code `A = B;` you are leaving a dangling stack pointer, for which you no longer store the address. This stack still occupies memory though!

Massey University
COLLEGE OF SCIENCES

Challenges

1) Write a method that implements Copy() for a Stack.

You need to think about: does the stack already exists?

This is the so-called deep copy, i.e., every element is recreated (new allocations) to the copy of the stack.

2) Write a method that implements Compare() for a Stack.

You need to think about what you should return (true or false) depending on: if a single element if missing or is different from the stack, is it a different stack? How about if all elements exist and are identical, but the ORDER is different?

NOTE: do not worry about overloading the operator ==

Te Kunenga ki Pūrehuroa

L07

Te Kunenga ki Pūrehuroa

A bit more on Classes etc...

Let's take a closer look at classes

Class: a collection of data and methods (a fancy name for functions).

Classes have specific scopes:

`Private:` data and methods are only available internally, i.e., can be manipulated/called by other methods of the same class

`Public:` accessible by any other functions, classes, main() etc

A bit more on Classes etc...

ADTs: the operations are well known, although the implementation details do not matter.

`Private:` for ADTs, place all data here

`Public:` for ADTs, place all methods here

Constructors and Destructors

We have seen that:

Every class K has a constructor called K

Every class K has a destructor called ~K

Example 1

This example uses static memory allocation.

The destructor is NEVER used.

```
Stack S;//constructor is used here
        //– static memory allocation
int main() {
  S.Push(6.2);
}
```

Te Kunenga
ki Pūrehuroa

Example 2

This example uses dynamic memory allocation: both constructor and destructor used

```
Stack *P;
int main() {
  P = new Stack;//constructor is used here
                //– dynamic memory allocation
  P->Push(6.2);
  delete P;  //destructor is used here
}
```

Example 3

Inside a function: both constructor and destructor used.

```
void func1() {
   Stack A;    //constructor is used here
   A.Push(4.8);
}  //destructor is used when function ends
```

Constructors and destructors

TIPS:

These are **not** explicitly called by the program.

Automatically used by the system.

Always write code with a destructor that *do nothing* to avoid problems!

L08

Queues

A QUEUE is an Abstract Data Type (ADT).

A queue is a line of things. Elements or items join the queue at the rear and leave at the front.

Queues are ***FIFO*** **(*F*irst *I*n is *F*irst *O*ut)**

(compared to Stacks that are LIFO)

Basic Operations on Queues

Join – place a new item on the rear of a queue

Leave – remove from the front of the queue

Front – look at the first (front) item in the queue

IsEmpty – returns true if the queue is empty

Queues based on Arrays

A queue is represented as

(front) A C D R F G (rear)

Or by an array:

first = 0

last = 5

0	1	2	3	4	5	6
A	C	D	R	F	G	

`first` keeps track of the index of the first item

`last` keeps track of the index of the last item

If an item join the queue, `last = last +1;`

If an item leave the queue, `first = first + 1;`

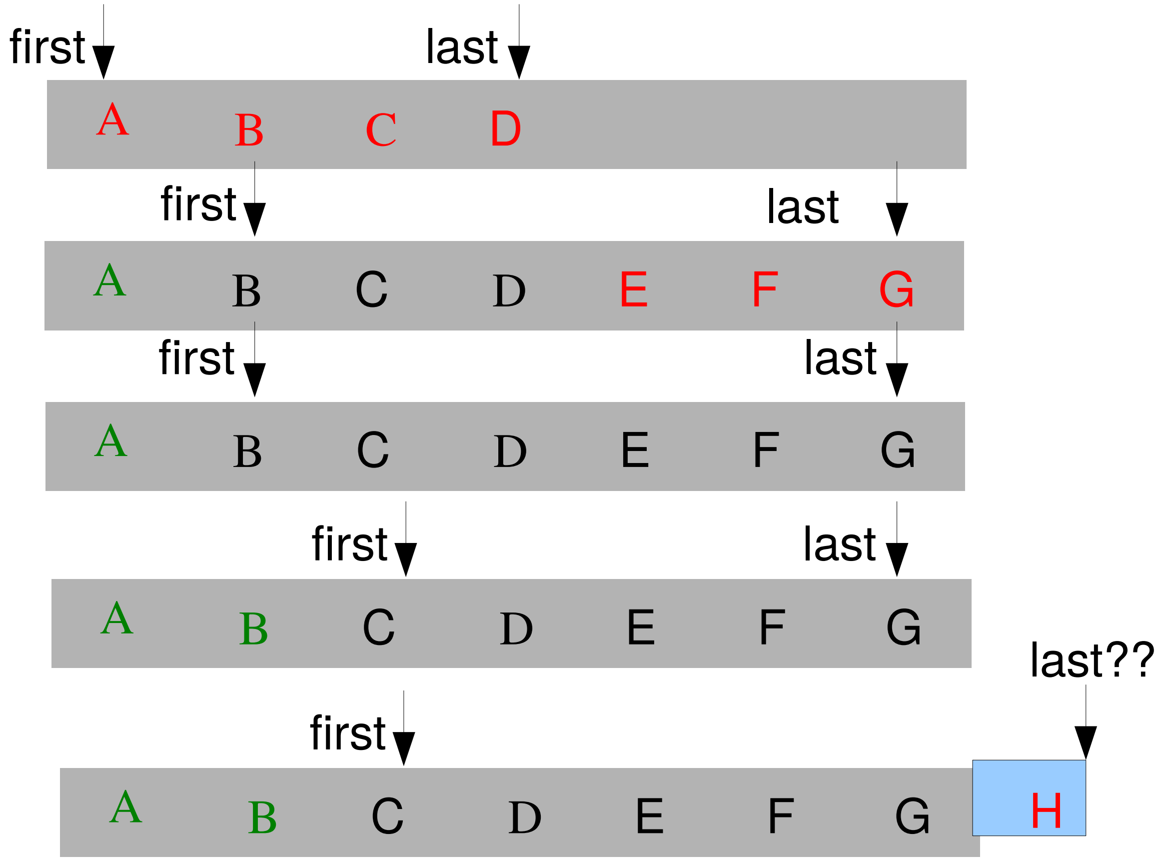
Queues based on Arrays

Problem: even if not using the entire array, items may be unable to join. Known as ***creeping problem***

Solution: use a ***circular array***. When last or first reaches the end of the array, they can move to the beginning of the array and continue to join new items.

Queues: creeping problem

A, B, C, D join, A leaves, E, F, G joins, B leaves, H joins



Queues: array implementation

```
const int Qmax = 100; //note the use of constants


class Queue {
private:
   float data[Qmax];
   int first, last, count;
public:
   Queue();
   ~Queue();
   void Join(float newthing);
   void Leave();
   float Front();
   bool isEmpty();
};
```

Queues: constructor / destructor

```
Queue::Queue() {
// constructor
  first = 0;  last = -1;  count = 0;
}

Queue::~Queue() {
// destructor


}
```

Queues: Join and Leave

```
void Queue::Join(float newthing) {
// place the new thing at the rear of the queue
  last++;
  if (last >= Qmax) { last = 0; }
  data[last] = newthing;
  count++; //watch out for overflow!!
}

void Queue::Leave() {
// front item is removed from the queue
  first++;
  if (first >= Qmax) { first = 0; }
  count--;   //watch out for underflow!!
}
```

Te Kunenga ki Pūrehuroa

Queues: Front and isEmpty

```
float Queue::Front() {
// return the value of the first item
  return data[first];
} //what if the queue is empty?

bool Queue::isEmpty() {
// returns true if the queue is empty
  if (count == 0) { return true; }
  return false;
}
```

Te Kunenga
ki Pūrehuroa

Massey University
COLLEGE OF SCIENCES

Queues: main() example

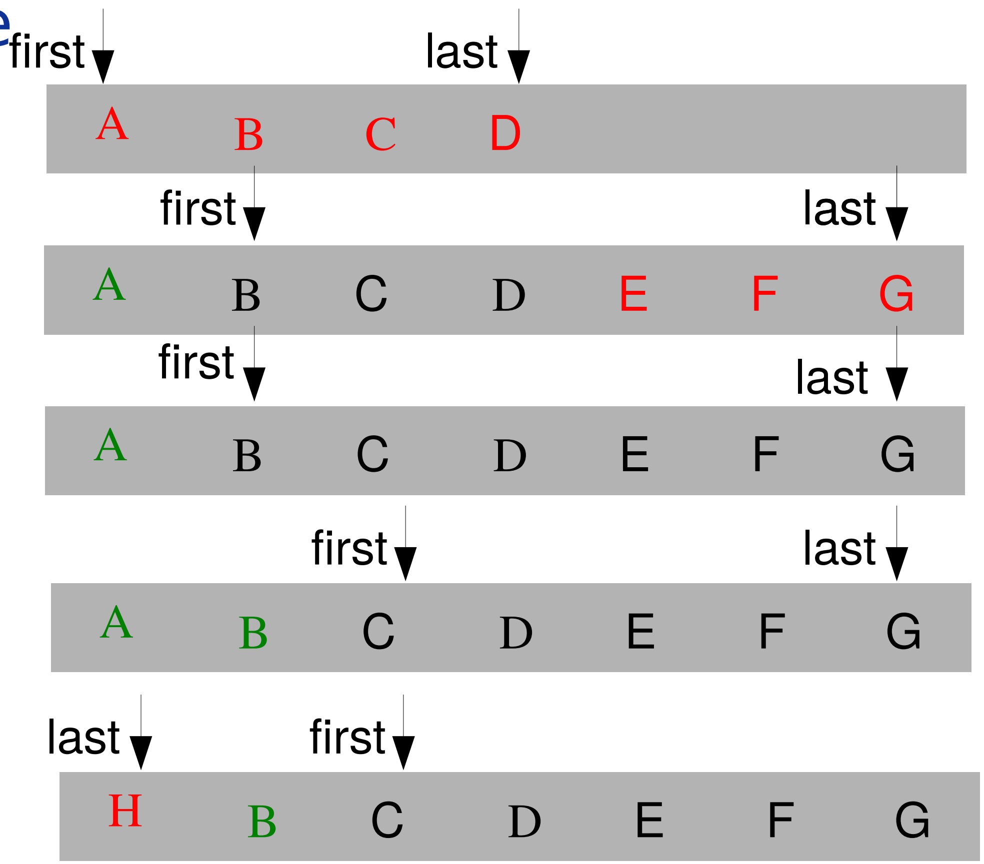
```
Queue G, H;

int main() {
      G.Join(7.1);
      printf("%f is at the front of queue G\n",
G.Front());
//   H.Join(7.1);
      if (H.isEmpty()) {
            printf("Queue H is empty\n");
      } else {
            printf("%f is at the front of queue H\n",
H.Front());
      }
}
```

Te Kunenga
ki Pūrehuroa

Queues example: circular

A, B, C, D join, A leaves, E, F, G joins, B leaves, H joins all leave



Challenge

1) Think how you can create a Queue that is dynamic in memory using linked-lists, using the same 4 basic methods (Join, Leave, Front and IsEmpty).

You need to think about:

- how similar is it with Stacks using LL?
- what type of AddNode should you use, add to head or to the tail?
- what difference does it makes to the performance if you have an extra pointer to the tail of the Queue?