Preview

- Static Member in a class
  - Static variables
  - Static Functions
- Recursion
  - Memory allocation
  - Solving problem with Recursion
  - Binary Search with recursion
  - Quick Sort

Static Class Members

- Each instance of a class has its own copy of all the data members of the class.
- A static variable in a class is a variable only one copy of variable is shared by all instances of a class.

Static Class Members

Static Members

- It is possible to create an member that does not belongs to any instance of a class.
- Such a member are known as static member in a class.
- When a value is stored in a static field, it is not stored in an instance of the class.

Static Class Members

Static Field

- When a field is declared with the key word static, there will be only one copy of the field in memory, regardless of the number of instances of the class that might exist.
- A single copy of class's static field is shared by all instances of the class
- Static field initiate one once, regardless of the number of instances of class that are created.

Static Class Members

Example usage of static variable in a game.

- A **Mighty Dog** is brave enough to attack a **Small Monster**.
- But a **Mighty Dog** become brave to attack a **Big Monster** only if count of **Mighty Dog** is more than five.
- We can set a **Mighty Dog** count as static variable, it will change based on the number of **Mighty Dogs**

Static Class Members

Static Class Members

Static Class Members

```
// Mighty Dog class
class MightyDog{
public:
        MightyDog();
        Attack();
        Transform();
        Create(int);
private:
        static int DogCount;
};

int MightyDog::DogCount = 0; // static member initialization

MightyDog::MightDog()
{
        DogCount++;
}
```

Static Class Members

- Static member can be private, public or protected member type.
- Public static member can be accessed without creating a instance of class by using "::" binary scope resolution operator.
- To access a private or protected static member, a public static member function must be provided.

```
#include <iostream>
using namespace std;

class StaticExample {
public:
        static int objectCount;
        StaticExample();
};

// Initiciate a static member
int StaticExample::objectCount =0;

StaticExample::StaticExample()
{
        objectCount ++;
}

int main()
{
        // public static member can be directly accessed
        // without creating an instance of a class
        cout << StaticExample::objectCount<<endl;
        for (int i =1; i<5; i++)
                StaticExample a;
        cout << StaticExample::objectCount<<endl;
        return 0;
}
```

```
#include <iostream>
using namespace std;

class StaticExample {
private:
        static int objectCount;
public:
        StaticExample();
        static int getCount() {return objectCount;}
};

// Initiciate a static member
int StaticExample::objectCount =0;

StaticExample::StaticExample()
{
        objectCount ++;
}

int main()
{
        // public static member function can be accessed without creating an instance
        // of a class private static member can be  accessed by using public static
        // member function
        cout << StaticExample::getCount() <<endl;
        for (int i =1; i<5; i++)
                StaticExample a;
        cout << StaticExample::getCount() <<endl;
}
```

Recursive Functions

- Recursive function – a function which call itself is recursive function
- Like a loop, a recursive function must have some way to control the number of time it repeat.

$$f(n) = \begin{cases} 5 & n = 1 \\ f(n-1) + 2 * \mathrm{n} & n > 1 \end{cases}$$

Induction vs. Recursion

Ex) Prove $P(n) = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$

Proof by induction)

Base case) n = 1 LHS = 1^2 =1, RHS=$\frac{1\times2\times3}{6} = 1$

It is true since LHS =RHS

Inductive step)

Inductive hypothesis: lets assume true for *n=k-1, k > 2* such that $P(k-1) = 1^2 + 2^2 + \cdots + (k-1)^2 = \frac{(k-1)k(2(k-1)+1)}{6}$----(IH)

Now prove for n =k such that $P(k) = 1^2 + 2^2 + \cdots + k^2 = \frac{k(k+1)(2k+1)}{6}$

LHS) $1^2 + 2^2 + \cdots + (k-1)^2 + k^2 = P(k-1) + k^2$ by inductive hypothesis (IH) we can rewrite $\frac{(k-1)k(2(k-1)+1)}{6} + k^2$

$$= \frac{(k-1)k(2k-1) + 6k^2}{6} = \frac{2k^3 - 3k^2 + k + 6k^2}{6} = \frac{2k^3 + 3k^2 + k}{6}$$

$$= \frac{k(k+1)(2k+1)}{6}$$

$\therefore$LHS = RHS

Recursive functions

- Recursive procedure can have multiple activations (executions) of its body at the same time.
- Each activation needs its own storage to save temporary values in the stack.
-

Recursive functions: $f(n) = \begin{cases} 5 & n = 1 \\ f(n-1) + 2*n & n > 1 \end{cases}$

```
#include <iostream>
using namespace std;

//recursive function
int function (int n)
{
   if (n == 1)
       return 5;
   else
       return function(n - 1)+ 2*n;
}


int main()
{
        int x = 4;

        cout << function (x)<<endl;
}
```

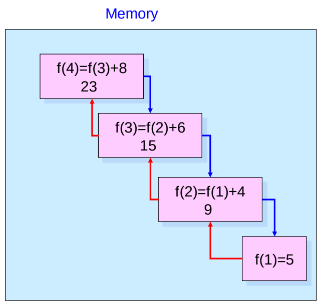
Recursive Functions

f(4) =f(3) + 2*4

f(3) =f(2) + 2*3

f(2) =f(1) + 2*2

f(1) =5



Solving Problems with Recursion

- Any problem can solve with recursion can solve with iteration.
- Usually, recursive algorithms are usually less efficient than iteration algorithms.
- Because recursive call requires several actions – each action need a memory space for saving information such as parameters, local variables.

Recursive functions: $f(n) = \begin{cases} 5 & n = 1 \\ f(n-1) + 2*n & n > 1 \end{cases}$

```
#include <iostream>
using namespace std;

//ex1:non-recursive function
int function (int n)
{
   int rval=0;
   for (int i=n; i>1; i--)
      rval = rval + 2*i;
   return rval + 5;
}


int main()
{
        int x = 4;

        cout << function (x)<<endl;
}
```

Solving Problems with Recursion

- But some repetition problems are more easily solved with recursion than with iteration.
- In some cases, the recursion method runs faster than iteration method (if we don't care about space matter).

Ex) Sorting Algorithms

- MergeSort, QuickSort running time is **O**(nlog$_2$n) in average case.
- But the running time of any sorting algorithms with iteration takes O(n^2) time.

Solving Problems with Recursion

Ex)
Factorials

$$f(n) = \begin{cases} 1 & if\ n = 1 \\ f(n) = n \times f(n-1) & if\ n > 1 \end{cases}$$

Solving Problems with Recursion

```
#include <iostream>
using namespace std;

int factorial (int);

int main()
{
        int x;

        cout <<"Enter a non-negative integer ";
        cin >> x;
        cout<< x <<"! = " << factorial (x) <<endl;
        return 0;
}
//recursive function
int factorial(int n)
{
        if (n == 1)
                return 1;   // Base case
        else
                return n * factorial(n - 1);
}
```

Solving Problems with Recursion

```
#include <iostream>
using namespace std;
int factorial (int);
int main()
{
        int x;

        cout <<"Enter a non-negative integer ";
        cin >> x;
        cout<< x <<"! = " << factorial (x) <<endl;
        return 0;
}
//non-recursive function
int factorial(int n)
{
       int rval =1;
        for (int i=n; i>1; i--)
           rval = rval * i;
        return rval;
}
```

Solving Problems with Recursion

- We can apply recursion to binary search
- In binary search problem can divide into three cases
  - If the mid value is a value looking for
  - If the mid value greater than a value looking for
  - If the mid value is less than a value looking for

Solving Problems with Recursion

```
int binarySearch(int array[], int numelems, int value)
{
   int first = 0,              // First array element
       last = numelems - 1,    // Last array element
       middle,                 // Mid point of search
       position = -1;          // Position of search value
   bool found = false;         // Flag
   while (!found && first <= last)
   {
      middle = (first + last) / 2;      // Calculate mid point
      if (array[middle] == value)       // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)   // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;            // If value is in upper half
   }
   return position;
}
```

Solving Problems with Recursion

```
int BinarySearch(int A[], int value, int low, int high)
{
        if (high < low)
                return -1; // not found
        int mid = low + (high - low) / 2;
        if (A[mid] > value)
                return BinarySearch(A, value, low, mid-1);
         else if (A[mid] < value)
                return BinarySearch(A, value, mid+1, high);
         else
             return mid; // found
    }
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int BinarySearch(int [], int, int, int);
int main()
{
            int A[10]={ 10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
            int value;
            int high = 9;
            int low =0;
            int index = -1;
            while (index <0)
            {
                        cout <<"What element are you looking for in the arrary? ";
                        cin >> value;
                        index = BinarySearch(A, value, low, high);
                        if (index == -1)
                                    cout<< "There is no such a element in the arrary, try again"<<endl;
            }
            cout <<" Index of "<< value <<" is " << index <<endl;
            return 0;
}

int BinarySearch(int A[], int value, int low, int high)
{
            if (high < low)
                        return -1; // not found
            int mid = low + (high - low) / 2;
            if (A[mid] > value)
                        return BinarySearch(A, value, low, mid-1);
             else if (A[mid] < value)
                        return BinarySearch(A, value, mid+1, high);
             else
                        return mid; // found
   }
```

Sorting with Recursion
(Quick Sort)

Description of Quicksort

- Quick sort is based on the divided-and-conquer paradigm.
- The three step divide-and-conquer process for sorting a typical subarray A[p .. r]

Sorting with Recursion
(Quick Sort)

The three step for Quicksort:

1. **Divide:** The array A[p .. r] is partitioned into two nonempty sub-arrays A[p .. q] and A[q+1 .. r] based on a pivot: elements in A[p .. q] are less than the pivot and each element of A[q+1 .. r] are greater or equal to pivot.
2. **Conquer:** The two sub-arrays A[p .. q] and A[q+1 .. r] are sorted by recursive call to quick sort.
3. **Combine:** Since the sub-arrays are sorted in place, no work is needed to combine them: the entire array A[p .. r] is now sorted.

Sorting with Recursion
(Quick Sort)

- The key to the Quick Sort is Partition which rearranges the sub-array.

```
QUICKSORT(A, p, r)
{
        If p < r
                q = Partition(A, p, r)
                QUICKSORT(A, p, q)
                QUICKSORT(A, q+1, r)
}
```

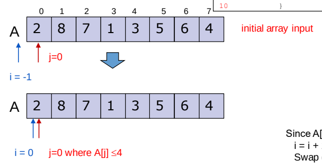
Sorting with Recursion
(Quick Sort: Partition1)

```
Partition(A, p, r)
{
        x = A[r]; // A[r] used as a pivot
        i = p -1
        for j = p to r -1
        {
                if A[j] ≤ x
                {
                        i = i + 1;
                        Swap(A[i], A[j]);
                }
        }
        Swap(A[i + 1], A[r]);
        Return i + 1;
}
```

Sorting with Recursion
(Quick Sort: Partition1)

```
Partition(A, p, r)
{
    x = A[r]; i = p - 1; // A[r] used as a pivot
    for j = p to r -1
    {
        if A[j] ≤ x
        {
            i = i + 1;
            Swap(A[i], A[j]);
        }
    }
    Swap(A[i + 1], A[r])
    Return i + 1
}
```

Partition(A, 0, 7)
p=0 r=7
x=A[7]=4
Initially i = - 1

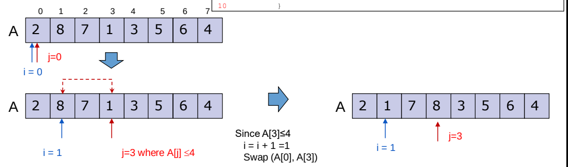


Since A[0]≤4
i = i + 1 =0
Swap (A[0, A[0]) does not change array

Sorting with Recursion
(Quick Sort: Partition1)

```
Partition(A, p, r)
{
    x = A[r]; i = p - 1; // A[r] used as a pivot
    for j = p to r -1
    {
        if A[j] ≤ x
        {
            i = i + 1;
            Swap(A[i], A[j]);
        }
    }
    Swap(A[i + 1], A[r])
    Return i + 1
}
```

Partition(A, 0, 7)
p=0 r=7
x=A[7]=4
Initially i = - 1

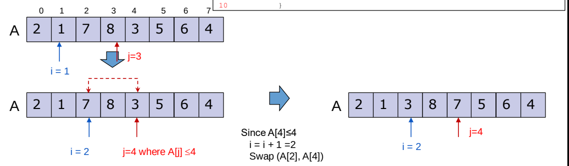


Since A[3]≤4
i = i + 1 =1
Swap (A[0], A[3])

Sorting with Recursion
(Quick Sort: Partition1)

```
Partition(A, p, r)
{
    x = A[r]; i = p - 1; // A[r] used as a pivot
    for j = p to r -1
    {
        if A[j] ≤ x
        {
            i = i + 1;
            Swap(A[i], A[j]);
        }
    }
    Swap(A[i + 1], A[r])
    Return i + 1
}
```

Partition(A, 0, 7)
p=0 r=7
x=A[7]=4
Initially i = - 1

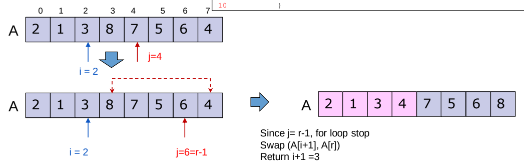


Since A[4]≤4
i = i + 1 =2
Swap (A[2], A[4])

Sorting with Recursion
(Quick Sort: Partition1)

```
Partition(A, p, r)
{
    x = A[r]; i = p - 1; // A[r] used as a pivot
    for j = p to r -1
    {
        if A[j] ≤ x
        {
            i = i + 1;
            Swap(A[i], A[j]);
        }
    }
    Swap(A[i + 1], A[r])
    Return i + 1
}
```

Partition(A, 0, 7)
p=0 r=7
x=A[7]=4
Initially i = - 1



Since j= r-1, for loop stop
Swap (A[i+1], A[r])
Return i+1 =3

```cpp
#include <iostream>
using namespace std;
int partition(int arr[], int p, int r) {
    int pivot = arr[r];  // Choose the last element as the pivot
    int i = p - 1;

    for (int j = p; j < r; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[r]);
    return i + 1;
}

void quick_sort(int A[], int p, int r) {
    if (p < r) {
        int q = partition(A, p, r);

        quick_sort(A, p, q - 1);  // Sort the left sub-array
        quick_sort(A, q + 1, r); // Sort the right sub-array
    }
}
int main() {
    int arr[] = {3, 6, 8, 10, 1, 2, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    quick_sort(arr, 0, n - 1);

    cout << "Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```