



**2007-03-06**  
**Lecture #9**  
**(Material on Test 2)**

**Today's Outline**  
**Recursion**

## Recap from Last Class

### Copy Constructors

```
main( )  
{  
    Student a ;  
    Student b = a ;           // 1 Copy Constructor called for 'b'  
    Student c(b) ;          // 1 Copy Constructor called for 'c'  
    f1(a,b,c) ;             // 1 Copy Constructor called for 'a'  
}  
  
void f1(Student a, Student & b, Student & c)  
{  
    a = b ; // pass by value, so a's changes not saved ... why?  
    b = c ; // pass by reference, so a's changes saved ... why?  
}
```

## Pointers Recap

```
main( )
{
    Student a ;
    Student * ptr = new Student() ;
    f1(a,ptr) ;           // 1 Copy Constructor called for 'a'
}

void f1(Student a, Student * ptr)
{
    a = *ptr ;           // pass by value, changes not saved ... why?
    *ptr = a ;           // pass by reference, changes saved ... why?
}
```

## Recap last class

### Stack (LIFO)

- **Array vs Linked List Implementation**
  - **Pros & Cons?**
- **Push**
- **Pop**

### Queue (FIFO)

- **Array vs Linked List Implementation**
  - **Pros & Cons?**
- **Enqueue**
- **Dequeue**

### Priority Queue

- **Enqueue (Insert into sorted list)**
- **Dequeue (same as regular Queue)**

## **Recursion**

**Chapter 6 in your book (lots of interesting stuff)**

**2 ways to solve an algorithm**

**1. Iterative**

**2. Recursive**

**Iterative – uses loops (for, while, do-while)**

**Recursive – repetitive process that calls itself**

## **Recursion**

### **Pros**

- **Some algorithm are naturally recursive and thus easy & short to code**

### **Cons**

- **No way to tell how deep recursion can go before stack overflow**
- **Can easily waste time and space**
- **Function call overheads**

## **Iterative**

### **Pros**

- **Less chance of stack overflow**

### **Cons**

- **The naturally recursive algorithms can get ugly to write Iteratively**

## Simple Recursion

**Print the numbers 1 to 10**

```
main( )  
{  
    f1(1) ; // prints 1,2,3,4,5,6,7,8,9,10  
}  
  
void f1(int x)  
{  
    cout << x << endl ;  
    if(x == 10)  
        return ;  
    else  
        f1(x++) ;  
} // DRAW THIS IN THE STACK
```

## Recursive Infinite Loop (stack overflow)

**Here's an example of a stack overflow ... out of memory error  
(Equivalent of an infinite loop in an iterative algorithm)**

```
main( )
```

```
{  
    f1(20) ;          // 20,21,22,23,....goes on forever  
}
```

```
void f1(int x)
```

```
{  
    cout << x << endl ;  
    if(x == 10)  
        return ;  
    else  
        f1(x++) ;  
} // WHY DOES THIS OVERFLOW THE STACK???
```

**Iteratively?**

```
void f1(int x)  
{  
    for(int i = 1 ; i <= x ; i++)  
        cout << i << endl ;  
}
```

## Recursion

```
main( )
{
    cout << multiplication(2,3) << endl ;           // 2 * 3
}
int multiplication(int x, int y)
{
    if(y == 0)
        return 0 ;
    else
        return x + multiplication(x, y-1) ;
}

// HOW DOES THIS WORK?

// WHAT IF it said ... return multiplication(x, y-1) + x ; ???
```

**Iteratively?**

```
int multiplication(int x, int y)  
{  
    int result = 0 ;  
    for(int i = 0 ; i < y ; i++)  
        result += x ;  
    return result ;  
}
```

## Recursion

```
main( )
{
    cout << integer_division(9,3) << endl ;    // 9 / 3
}
int integer_division(int x, int y)
{
    if(x < y)
        return 0 ;
    else
        return 1 + integer_division(x-y, y) ;
}

// How does this work?
```

**Iteratively?**

```
int integer_division(int x, int y)  
{  
    int result = 0 ;  
    for(int i = x ; x > i ; i = i - y)  
        result += 1 ;  
    return result ;  
}
```

## Recursion

```
main( )
{
    cout << addition(9,3) << endl ;    // 9 + 3
}
int addition(int x, int y)
{
    if(y == 0)
        return x ;
    else
        return 1 + addition(x, y-1) ;
}

// How does this work?
```

**Iteratively?**

```
int addition(int x, int y)  
{  
    int result = x ;  
    for(int i = 0 ; i < y ; i++)  
        result += 1 ;  
    return result ;  
}
```

## Recursion

```
main( )
{
    cout << subtraction(9,3) << endl ;    // 9 - 3
}
int subtraction(int x, int y)
{
    if(y == 0)
        return x ;
    else
        return subtraction(x, y-1) - 1 ;
}

// How does this work?
```

**Iteratively?**

```
int subtraction(int x, int y)  
{  
    int result = x ;  
    for(int i = 0 ; i < y ; i++)  
        result -= 1 ;  
    return result ;  
}
```

## Recursion

### Factorials

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

```
int factorial(int x)
{
    if(x == 1) return 1 ;
    else return x * factorial(x-1) ;
}
```

### Iteratively?

```
int factorial(int x)
{
    int result = 1 ;
    for(int i = 2 ; i < x ; i++)
        result *= i ;
    return result ;
}
```

## Understanding a function call - Stackframe

**Every time a function is called ... these items get put on the stack**

- **function parameters**
- **local variables in function**
- **return statement in the function**

```
int factorial(int x)  
{  
    int stop = 1 ;  
    if(x == stop) return 1 ;  
    else return x * factorial(x-1) ;  
}  
// x is the function parameter  
// stop is the local variable  
// return statement is x * factorial(x-1)
```

## **Recursion**

**What have we seen with all our Recursive Algorithms so far?**

- **Every recursive call must either**
  - (1)Solve a part of the problem**
  - or**
  - (2)Reduce the size of the problem**
- **Typically there are 2 cases we're worried about**
  - (1)base case (no recursive call)**
  - (2)general case (does the recursive call)**
- **If there is no base case, or the base case is faulty, then we'll get a stack overflow**

## Recursive Linked List Print

```
class LinkedList{
    ...
    public:
        void Print( )
            { RecursivePrint(headptr) ; }
    private:
        void RecursivePrint(node * cur) // helper function
        {
            if(headptr == NULL) return ;
            else
            {
                cout << cur->data ;
                RecursivePrint(cur->next) ;
            }
        }
};
```

## Fibonacci Numbers

0 1 2 3 4 5 6 7 8 9

0,1,1,2,3,5,8,13,21,34, .....

Iteratively?

```
int Fibonacci(int x){
    if(x == 0) return 0 ;
    else{
        int a = 0 ;
        int b = 1 ;
        for(int i = 2 ; i < x ; i++){
            int temp = a + b ;
            a = b ;
            b = temp ;
        }
        return b ;
    }
}
```

## Fibonacci Numbers

0 1 2 3 4 5 6 7 8 9

0,1,1,2,3,5,8,13,21,34, .....

Recursively?

```
int Fibonacci(int x){  
    if(x == 0 || x == 1) return x ;  
    else return Fibonacci(x-1) + Fibonacci(x-2) ;  
}
```

WHAT'S THE DIFFERENCE?

```
int Fibonacci(int x){  
    if(x == 0 || x == 1) return x ;  
    else return Fibonacci(x-2) + Fibonacci(x-1) ;  
}
```

## Recursive Run Time

### Fibonacci Numbers

<b>Fib #?</b>	<b># Recursive Calls</b>	<b>~Time</b>	<b>Iterative Time</b>
<b>1</b>	<b>1</b>	<b>&lt; 1 second</b>	<b>&lt; 1 second</b>
<b>4</b>	<b>9</b>	<b>&lt; 1 second</b>	<b>&lt; 1 second</b>
<b>6</b>	<b>25</b>	<b>&lt; 1 second</b>	<b>&lt; 1 second</b>
<b>10</b>	<b>177</b>	<b>&lt; 1 second</b>	<b>&lt; 1 second</b>
<b>20</b>	<b>21,891</b>	<b>&lt; 1 second</b>	<b>&lt; 1 second</b>
<b>30</b>	<b>2,692,573</b>	<b>7 seconds</b>	<b>&lt; 1 second</b>
<b>35</b>	<b>29,860,703</b>	<b>1 minute</b>	<b>&lt; 1 second</b>
<b>40</b>	<b>331,160,281</b>	<b>13 minutes</b>	<b>&lt; 1 second</b>

**So, even though Fibonacci is naturally recursive, and simple to write recursively ... it's run-time because horrible because of the function calls and the stackframes used**