

# Lecture 2.1

## Arrays

CS112, semester 2, 2007

## How to define an array?

- An array is defined with this syntax (the same syntax is used in C and C++):

```
datatype arrayName[size];
```

Examples:

```
int ID[30];
float temperatures[31];      /* Could be used to store the daily
                           temperatures in a month */
char name[20];
int *ptrs[10]; /* An array holding 10 pointers to integer data */
```

CS112, semester 2, 2007

## What are arrays?

- Arrays are data structure.
- They are used to store a group of objects/variables.
- All elements of an array must be of the **same** data type: *int, float, char, pointer to int*.
- The elements are stored **sequentially** in memory (this allows powerful manipulation with pointers).

CS112, semester 2, 2007

## Using Arrays

- Array indexes starts from zero in C and C++.
- This means that an array of size N will be indexed from 0 to (N-1)

Example:

```
int myExample[3];
myExample[0] /* FIRST element */
myExample[1] /* SECOND element */
myExample[2] /* THIRD element */
```

CS112, semester 2, 2007

## Sample program

- This sample program calculates and stores the squares of the first one hundred positive integers:

### C++ program

```
#include <iostream.h>
#include <math.h>
#include <stdlib.h>

int main()
{
    int square[100];
    int i;

    for (i=0; i<100; i++){
        square[i] = (i+1)*(i+1);
        cout<<"Square of "<<i+1<<" is: "
             <<square[i]<<endl;
    }

    system("PAUSE");
    return 0;
}
```

### C program

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main()
{
    int square[100];
    int i;

    for (i=0; i<100; i++){
        square[i] = (i+1)*(i+1);
        printf("Square of %d is: %d\n", i+1, square[i]);
    }

    system("PAUSE");
    return 0;
}
```

CS112, semester 2, 2007

## Common specifiers for C printf/scanf

Specifier	Argument Type
%d	int
%f	float or double
%e	float or double, output in scientific notation.
%c	Character
%s	character string (char *)

CS112, semester 2, 2007

## How to initialize an array?

- An array can be initialized with an explicit **initialization list** in its definition.

Example:

- int myArray[5]={6,7,888,987,0};

CS112, semester 2, 2007

## Practical Problem: recursive functions

- Arrays can be used to generate sequence of numbers where a term of sequence depends on next or previous term.

Example: fibonacci sequence, mathematical factorial etc.

- A Fibonacci sequence is a numerical sequence such that after the second element, all numbers are equal to the sum of the previous two elements.(1,1,2,3,5,8,13,21,...)
- Factorial of non negative integer  $n$  is the product of all positive integers less than or equal to  $n$ . (1, 1, 2, 6, 24, ...)
- Write a program to calculate and print the first 20 elements of these sequences.

CS112, semester 2, 2007

## Lecture 2.2

### Arrays (cont.)

CS112, semester 2, 2007

### How to initialize an array?

#### Example

- int myArray[10];
- int i;
- can do:**
- for (i = 0; i < 10; i++)  
    myArray [ i ] = 0; // execution time
- or:**
- int myArray[10] = { 0 }; //compile time,  
                              //better  
                              //elements are initialized implicitly

CS112, semester 2, 2007

### How to initialize an array? (cont)

- An array can be initialized with an explicit **initialization list** in its definition.

Example:

- int myArray[ 5 ] = { 6, 7, 888, 987, 0 };

CS112, semester 2, 2007

### How to initialize an Array?

#### Example

- int myArray [3] = { 3, 4, 5 };
- // make sure to provide exact number
- or
- int myArray [ ] = {3 ,4, 5 };
- // creates an three-element array

CS112, semester 2, 2007

## Initializing arrays of characters

### Example

- `char charArray [6] = { 'h','e','l','l','o', '\0'};`
- or
- `char charArray1[ ] = {'h','e','l','l','o', '\0'};`
- or
- `char charArray2[ ] = "hello";`  
// size is determined by the compiler

Special character at the end of the string is known as ‘null character’ which is used to signify end of the string.

CS112, semester 2, 2007

## Initializing arrays of characters (cont)

### Example

- `char exOne[ ] = "bula";`
- `int j;`
- `for ( j = 0; j < 5; j++)`  
`cout << exOne[ j ] ;`
- **This will print**
- `bula`

CS112, semester 2, 2007

## Multidimensional arrays

- Commonly used to represent tables/matrices

### Examples

- `float temperatures[12][31];`
- /\* Used to store temperature data for a year \*/
- `char daysofweek[7][10];`

CS112, semester 2, 2007

## Accessing multidimensional arrays

- The common way is to use nested for loops.

```
#define MAXI 50;
#define MAXJ 75;
int i, j ;
float values[ MAXI ][ MAXJ ];
for (i = 0; i < MAXI; i++) {
    for (j = 0; j < MAXJ; j++) {
        values[ i ][ j ] = whatever;
    }
}
```

CS112, semester 2, 2007

## Accessing multidimensional arrays (cont.)

- Notice the expression

`values[ i ] [ j ] = whatever;`  
this is the correct form to reference a  
multidimensional array,

- The expression

`values [ i, j ]` would be **wrong!**

CS112, semester 2, 2007

## Common errors for arrays

- The most common cause of writing/reading to invalid array indexes are errors in loop limits.

```
int i;
float b[10];
for (i < 0 ; i <= 10; i++) {
    b[i] = 3.14 * i * i;
}
```

- This loop should use "<" rather than "<="

CS112, semester 2, 2007

## Initializing multidimensional arrays

- Example

```
int a[2][3] = { {1,2,3} , {4, 5, 6} },
b[2][3] = { 1,2,3,4,5},
c[2][3] = { {1,2} , {4} };
```

- Note: If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0.

CS112, semester 2, 2007

## Initializing multidimensional arrays

- Row 0 a [0][0] a[0][1] a[0][2]

- Row 1 a [1][0] a[1][1] a[1][2]

- //printing the array with the following code

```
for (int i = 0; i< 2 ; i++){
    for (into j = 0; j<3 ; j++)
        cout << a[i][j]<< ' ';
```

}

CS112, semester 2, 2007

## Initializing multidimensional arrays

- Values for array a

1 2 3  
4 5 6

- Values for array b

1 2 3  
4 5 0

- Values for array c

1 2 0  
4 0 0

CS112, semester 2, 2007

## Passing multidimensional arrays to functions

- When passing array parameters, the size of the first subscript of a multidimensional array is not required, but all subsequent subscript sizes are required.

Example:

- Function Declaration:

```
void printArray(int array[][3]);
```

- Function Call:

```
int myarray[4][3];  
printArray(myarray);
```

CS112, semester 2, 2007

# Lecture 2.3

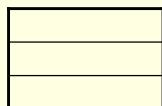
## Pointers

### Introduction to pointers

- Pointers are **variables that hold addresses** in C and C++.
- Provide much power and utility for the programmer to access and manipulate data.
- Useful for passing parameters into functions, allowing a function to modify and return values to a calling routine.
- Used in dynamic memory allocation

## Variables storage

- How values for variables are stored in computer memory?
- The computer memory could be represented like this:  
Address  
0x241FF5C  
0x241FF58  
0x241FF54
- Addresses are always represented in hexadecimal format.



### Hexadecimal numbers

- Everyday we use base 10 to represent numbers.  
0 1 2 3 4 5 6 7 8 9
- Computers use base 2 (binary) to store information  
0 1
- For memory addresses base 16 is used for the convenience of representing bigger numbers with less digits.  
0 1 2 3 4 5 6 7 8 9 A B C D E F

## Hexadecimal numbers (cont.)

- F = 15 in decimal and is represented as 1111 in binary ( $1*8+1*4+1*2+1*1$ )
- Therefore, one digit is used to represent 4 bits.
- This makes it easy to understand which bits are 1 and which ones are 0 for a number like OFF00 (What will this number be in decimal?)

## Hexadecimal numbers (cont.)

- With two digits in decimal, the maximum number you can represent is 99
- In hexadecimal this would be FF=255 ( $F*16+F*1$ )
- With four digits 9999 or FFFF=65535 ( $F*4096+F*256+F*16+F*1$ )

## How are values for variables stored in memory

- As a program is executing, all variables are stored in memory.
- Each at its own unique address or location.
- Typically a **variable** and its associated memory address contain **data values**.

## Values in memory

- For example, when you declare  
`int count = 5;`  
The value “5” is stored in memory and can be accessed by using the variable “count” Address
- 0x241FF5C      

5

      count
- 0x241FF58
- 0x241FF54

## Manipulating memory address

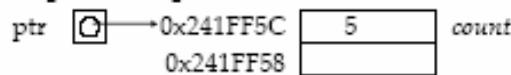
- To manipulate the memory address of a variable, we use the unary operator `&`.
- So, if you do:

```
cout << &count << endl;
```

This should print the address of the variable “count”, which for this example is 0x241FF5C.

## Pointer representation

- Usually, the address stored in the pointer is the address of some other variable.
- `int *ptr; //declares a pointer to an integer`  
`//ptr`
- `ptr = &count; //stores the address of count in ptr`
- The pointer “points to” count.

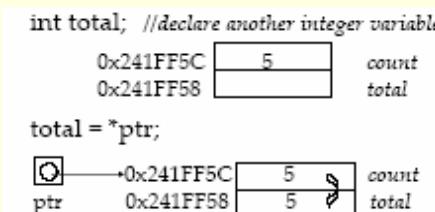


## Pointers and memory address

- A pointer is a special type of **variable** that **contains a memory address** rather than a data value.
- Data is modified when a normal variable is used.
- The value of the address stored in a pointer is modified as a pointer variable is manipulated.

## Pointers dereferencing

- To get the **value** that is stored at the memory location in the pointer.
- Dereferencing is done with the unary operator `“*”`.



## Declaration and Initialization examples

- Declaring and initializing a pointer is really easy:

```
int j=1;  
int k=2;  
int *pt1; //pointer to integer  
int *pt2; //pointer to integer  
pt1 = &j; // pt1 points to j  
pt2 = &k; //pt2 contains the address of k
```

## Declaration and initialization examples

```
float values[100];  
float results[100];  
float *pt3; //declares pointer to float  
float *pt4; //declares a float pointer  
pt3 = values; //pt3 contains the address  
//of the first element of values  
pt3 = &values[0] //This is the equivalent of the  
//above statement.  
//pt3 points to values
```

# Lecture 3.1

## Pointers (cont.)

CS112, semester 2, 2007

1

## Pointer Dereferencing

- Dereferencing allows manipulation of the **data** contained at the memory address stored in the pointer.
- Remember!
  - The pointer stores a memory **address**.
  - Dereferencing allows the **data** at that memory address to be modified.
  - The unary operator “**\***” is used to dereference.

CS112, semester 2, 2007

2

## Pointer Dereferencing (cont)

- Consider the following piece of code:

```
int j = 1;
pt1 = &j; //pt1 points to j
*pt1 = *pt1 + 2; //adds two to the value
                  //pointed to by pt1
```
- The effect of the above statement is to add 2 to j.

CS112, semester 2, 2007

3

## Pointer Dereferencing (cont 2)

- The contents of the address contained in a pointer may be assigned to another pointer or to a variable.

```
*pt2 = *pt1; // Assigns the contents of the
              //memory pointed to by pt1 to the
              //contents of the memory pointed to
              //by pt2;
```

```
k = *pt2; //Assigns the contents of the
              //address pointed to by pt2 to k.
```
- The value of k will now be 3.

CS112, semester 2, 2007

4

## Pointer Arithmetic

- Pointers can be incremented, decremented and manipulated using arithmetic expressions.

```
pt3 = &values[0]; // The address of the first element  
//of "values" is stored in pt3  
  
pt3++; // pt3 now contains the address of the  
// second element of values  
  
*pt3 = 3.1415927; // The second element of values  
//now has pie (actually pi)
```

CS112, semester 2, 2007

5

## Pointer Arithmetic (cont 2)

```
pt3 = &values[0]; // pt3 contains the address of the first  
// element of values  
pt4 = &results[0]; // pt4 contains the address of the first  
// element of results  
for (i=0; i < 100; i++)  
{  
    *pt4 = *pt3; // The contents of the address contained  
    // in pt3 are assigned to the contents of the  
    // address contained in pt4  
    pt4++;  
    pt3++;  
}
```

CS112, semester 2, 2007

7

## Pointer Arithmetic (cont)

```
pt3 += 25; // pt3 now points to the 27th element of  
//values  
*pt3 = 2.22222; // The 27th element of values is now  
//2.22222  
pt3 = values; // pt3 points to the start of values, now  
for (i = 0; i < 100; i++)  
{  
    *pt3++ = 37.0; // This sets the entire array to 37.0  
}
```

CS112, semester 2, 2007

6

## Relationship between Pointers and Arrays

- Suppose the following declarations are made.  
**float temperatures[31];**  
**float \*tmp;**
- The address of the first element of the array temperatures can be assigned to temp in two ways.  
**tmp = &temperatures[0];**  
**tmp = temperatures;**

CS112, semester 2, 2007

8

## Relationship between Pointers and Arrays (cont)

- Values for the first element can be assigned in two ways:  
**temperatures[0] = 29.3;**  
**\*tmp = 15.2;**
- Other elements can be updated via the pointer, as well.  
**tmp = &temperatures[0];**  
**\*(tmp + 1) = 19.0; /\*assigns 19 to the second element of temp \*/**

CS112, semester 2, 2007

9

## Relationship between Pointers and Arrays (cont2)

- tmp = tmp + 10; /\* tmp now has the address of the 11th element of the array \*/**
- \*tmp = 25.0; /\* temperatures[9] = 25, remember that arrays are zero based, so the tenth element is at index 9 \*/**
- tmp++; /\* tmp now points at the 12<sup>th</sup> element \*/**
- \*tmp = 40.9; /\* temperatures[11] = 40.9 \*/**

CS112, semester 2, 2007

10

## Pointers and Character Arrays

- Assigning a character literal to an array  
**char str1[] = "Hello World";**
- A character pointer can also be assigned the address of a string constant or of a character array.  
**char \*lpointer = "Hello World"; /\* Assigns the address of the literal to lpointer \*/**  
**char \*apointer = str1; /\* Assigns the starting address of str1 to apointer \*/**  
**char \*apointer = &str1[0]; /\* Assigns the starting address of str1 to apointer \*/**

CS112, semester 2, 2007

11

## Copying one array to another

- There is no direct means in the C or C++ language to copy one array to another.
- Must be done either with a standard library function or element wise in a loop.
- Example  
**#include <stdio.h>**  
**int main()**  
**{**  
    **char str1[] = "Hello World";**  
    **char str2[] = "Goodbye World";**  
    **str2 = str1;**  
    **return 0;**  
**}**

CS112, semester 2, 2007

12

# Lecture 3.2

## Pointers (cont.)

CS112, semester 2, 2007

1

## Copying using character pointers (cont)

```
printf("str1 is %s\n",str1);
printf("str2 is %s\n",str2);
printf("cpt1 is %s\n",cpt1);
printf("cpt2 is %s\n",cpt2);
cpt2 = cpt1;
printf("str1 is %s\n",str1);
printf("str2 is %s\n",str2);
printf("cpt1 is %s\n",cpt1);
printf("cpt2 is %s\n",cpt2);
```

CS112, semester 2, 2007

3

## Copying using character pointers

```
#include <stdio.h>
int main()
{
    char str1[] = "Hello World";
    char str2[] = "Goodbye World";
    char *cpt1;
    char *cpt2;
    cpt1 = &str1[0];
    cpt2 = &str2[0];
```

CS112, semester 2, 2007

2

## Copying using character pointers (cont2)

### Results:

**str1** is Hello World  
**str2** is Goodbye World  
**cpt1** is Hello World  
**cpt2** is Goodbye World  
**str1** is Hello World //contents are the same  
**str2** is Goodbye World //contents are the same  
**cpt1** is Hello World  
**cpt2** is Hello World

CS112, semester 2, 2007

4

## Point to note:

- Actually string hasn't been copied through the use of pointers. (why?????)
- Copy means copy the content and paste into different location.
- Here we are not coping the content just changing the pointer reference from one location to another.

CS112, semester 2, 2007

5

## Copying element by element

```
#include <stdio.h>
int main()
{
    int i;
    char str1[] = "Hello World";
    char str2[] = "Goodbye World";
    printf("str1 is %s\n",str1);
    printf("str2 is %s\n",str2);
    i = 0;
    while ((str2[i] = str1[i]) != '\0') {
        i++;
    }
}
```

CS112, semester 2, 2007

6

## Copying element by element (cont.)

```
...
printf("str1 is %s\n",str1);
printf("str2 is %s\n",str2);
return 0;
}
```

CS112, semester 2, 2007

7

## Copying element by element (cont.)

### Results:

```
str1 is Hello World
str2 is Goodbye World
str1 is Hello World
str2 is Hello World
```

CS112, semester 2, 2007

8

## Practice Problem 2

- Try reimplementing the above program using pointers in the copy loop.

- Hints:

```
cpt1 = &str1[0];  
cpt2 = &str2[0];
```

Use these pointers in the while loop,  
remember to dereference.

## Copying element by element using pointers

```
#include <stdio.h>  
int main()  
{  
    char str1[] = "Hello World";  
    char str2[] = "Goodbye World";  
    char *cpt1;  
    char *cpt2;
```

## Copying element by element using pointers (cont.)

```
cpt1 = &str1[0];  
cpt2 = &str2[0];  
printf("str1 is %s\n",str1);  
printf("str2 is %s\n",str2);  
while ((*cpt2 = *cpt1) != '\0') {  
    cpt2++;  
    cpt1++;  
}
```

## Code comparison

- while ((\*cpt2 = \*cpt1) != '\0') {  
 cpt2++;  
 cpt1++;
- i = 0;  
 while ((str2[i] = str1[i]) != '\0') {  
 i++; }
- while (str1[i] != '\0'){  
 str2[i]=str1[i];  
 i++; }

## Calling functions by reference

- Passing a parameter by reference means that we are “referencing” the parameter, **not** passing the **value** of the parameter.
- There are two main reasons to use pass by reference:
  - To allow functions to modify several values at a time.
  - To pass large data objects to a function and avoid the overhead of copying.

CS112, semester 2, 2007

13

## Call by reference

```
#include <iostream>
void cubeByReference (int *); //prototype
int main( ){
    int number = 5;
    ...
    cubeByReference (&number);
    ...
}
void cubeByReference (int *nPtr)
{ *nPtr = *nPtr * *nPtr * *nPtr; }
```

CS112, semester 2, 2007

15

## Call by value

```
#include <iostream>
int cubeByValue (int); //prototype
int main( ){
    int number = 5;
    ...
    number = cubeByValue (number);
    ...
}
int cubeByValue (int n)
{ return n * n * n; }
```

CS112, semester 2, 2007

14

## Arrays of Pointers

- Arrays may contain pointers.
- The most common use is to form an array of strings
- `char *names[4] = {"Atish", "Itendra", "Ron", "Vilimone"};`  
`names[0] 'A' 't' 'i' 's' 'h' '\0'`  
`names[1] 'I' 't' 'e' 'n' 'd' 'r' 'a' '\0'`  
`names[2] 'R' 'o' 'n' '\0'`  
`names[3] 'V' 'i' 'l' 'i' 'm' 'o' 'n' 'e' '\0'`
- `char [4][9]`

CS112, semester 2, 2007

16

## “sizeof” operator

- **Definition:** The sizeof operator returns the size of an object in bytes as a value of type size\_t, which is usually unsigned int.

- **Example**

```
size_t mySize;  
int x;  
mySize = sizeof (x);
```

## sizeof examples

```
int myInt;  
printf("Size of int is %d\n",sizeof(myInt)); //argument of sizeof is an  
//object  
printf("Size of int is %d\n",sizeof(int)); //argument of sizeof is a  
//data type  
printf("Size of char is %d\n", sizeof(char));  
printf("Size of short is %d\n", sizeof(short));  
printf("Size of int is %d\n", sizeof(int));  
printf("Size of long is %d\n", sizeof(long));  
printf("Size of float is %d\n", sizeof(float));  
printf("Size of double is %d\n",sizeof(double));
```

## sizeof and Objects

- **Examples:**

```
sizeof(float);  
float value;  
sizeof(value);  
class Cat {  
...  
};  
sizeof(Cat);
```

# Lecture 3.3

## Structures (structs)

CS112, semester 2, 2007

1

### How to declare a structure?

- A structure is declared by using the keyword **struct** followed by an optional structure tag followed by the body of the structure.

```
struct point {  
    int x;  
    int y;  
};
```

CS112, semester 2, 2007

3

### What are structures?

- A structure is a complex data type built by using elements of other types.
- A structure provides a way of grouping variables under a single name for easier handling and identification.
- Structures may be copied to and assigned. They are also useful in passing groups of logically related data into functions.

CS112, semester 2, 2007

2

### How can I use this?

- The struct declaration is a user defined data type.
- This means that with a struct, YOU can, define YOUR OWN data types.
- Declaring
  - point left, right;
  - point origin;is analogous to
  - float rate;

CS112, semester 2, 2007

4

## Using structures

- The individual members of a structure can be accessed using ".", the member access operator.

- Example

```
struct Dot {  
    int x;  
    int y;  
};  
.....  
Dot location;  
cout << "The x coordinate is " << location.x;  
cout << "The y coordinate is " << location.y;
```

CS112, semester 2, 2007

5

## Nesting structures

- A rectangle could be represented as follows.

```
struct Rect{  
    Dot upperLeft;  
    Dot upperRight;  
    Dot lowerLeft;  
    Dot lowerRight;  
};
```

```
Rect myRectangle;
```

- To access the members, just use the "dot" notation:

```
myRectangle.upperLeft.x=1;;  
myRectangle.lowerRight.y=3;
```

CS112, semester 2, 2007

6

## Arrays of Structures

- A common way is to define an array of structs.
- Example,  
For counting the occurrence of each letter in a string.
- One approach: declare two separate arrays
  - One would hold the letters to compare against.
  - One to hold a count of the occurrences of each letter.

```
#define NUMLETTERS 26  
....  
char letter[NUMLETTERS];  
int count[NUMLETTERS];
```

CS112, semester 2, 2007

7

## Arrays of Structures (cont)

- #define NUMLETTERS 26
- struct lettertype {  
 char letter;  
 int count;  
};
- typedef** lettertype Letter;
- The keyword **typedef** provides a mechanism for creating synonyms for previously defined data types.
- Letter alphabet[NUMLETTERS];

CS112, semester 2, 2007

8

## Arrays of structures (cont2)

- The previous declarations could also be written as follows.

```
typedef struct lettertype {  
    char letter;  
    int count;  
} Letter;  
Letter alphabet[NUMLETTERS];
```

CS112, semester 2, 2007

9

## Sample Program

```
#include <stdio.h>  
#include <string.h>  
#define NUMLETTERS 26  
struct Letter {  
    char letter;  
    int count;  
};
```

CS112, semester 2, 2007

10

## Sample Program (cont 2)

```
int main()  
{  
    int i,j;  
    int length;  
    Letter alphabet[] = {  
        'a',0,'b',0,'c',0,'d',0,'e',0,'f',0,'g',0,'h',0,  
        'i',0,'j',0,'k',0,'l',0,'m',0,'n',0,'o',0,'p',0,  
        'q',0,'r',0,'s',0,'t',0,'u',0,'v',0,  
        'w',0,'x',0,'y',0,'z',0};  
    char sampleString [] =  
        "now is the time for all good men to come to the  
        aid of their country";
```

CS112, semester 2, 2007

11

## Sample Program (cont 3)

```
/* Find length of string */  
length = strlen(sampleString);  
for (i = 0; i < length; i++){  
    for (j = 0; j < NUMLETTERS; j++) {  
        if (sampleString[i] == alphabet[j].letter){  
            alphabet[j].count++;  
        }  
    }  
}
```

CS112, semester 2, 2007

12

## Sample Program (cont 4)

```
for (j = 0; j < NUMLETTERS; j++) {  
    printf("%c found %d times\n",  
        alphabet[j].letter, alphabet[j].count);  
}  
return 0;  
}
```

# CS112

Pointers

# Pointers

- C and C++ provide a pointer data type to allow programmers to manipulate memory space.
- Pointers are **variables that hold addresses** in C and C++.
- Provide much power and utility for the programmer to access and manipulate data.
- Useful for passing parameters into functions, allowing a function to modify and return values to a calling routine.
- Used in dynamic memory allocation



# Pointers

## Careful:

- Writing in C or C++ is like running a chain saw with all the safety guards removed. *Bob Gray*
- In C++ it's harder to shoot yourself in the foot, but when you do, you blow off your whole leg. *Bjarne Stroustrup*.



# Pointers and Memory

- How values for variables are stored in computer memory?
  - Whenever a variable is declared, a memory location is associated with the variable.
  - Whenever the variable is accessed the data value is taken from that particular memory location.
  - A **variable** contains a value, but a **pointer** specifies where a value is located.
  - A **pointer** denotes the memory location of a variable.



# Memory and Addresses

- Here's a picture of RAM.
  - Every byte in RAM has an address
    - (shown in groups of four bytes)
  - Addresses are always represented in hexadecimal format.

Address	Data value
0x241FF50	
0x241FF54	
0x241FF58	
0x241FF5C	

# Hexadecimal numbers

- Everyday numbers use base 10 (decimal)
  - 0 1 2 3 4 5 6 7 8 9
- Computers use base 2 to store information (binary)
  - 0 1
- For memory addresses **base 16** is used (hexadecimal)
  - 0 1 2 3 4 5 6 7 8 9 A B C D E F
  - A=decimal 10, B=decimal 11, ..., F=decimal 15

# Converting hexadecimal to decimal

- For a decimal number 345 we have that

$$3*10^2 + 4 *10^1 + 5 * 10^0 = \\ 300 + 40 + 5 = 345$$

using decimals!

- Convert 256 hexadecimal to decimal

$$2*16^2 + 5 *16^1 + 6 * 16^0 = \\ 512 + 80 + 6 = 598$$

using decimals!

- Convert FA8 hexadecimal to decimal

$$'F'* 16^2 + 'A' * 16^1 + '8' * 16^0 = \\ 15*16^2 + 10 *16^1 + 8 * 16^0 = \\ 3840 + 160 + 8 = 4008$$

using decimals!

# Hexadecimal numbers

- Why hexadecimal?
  1. Data is binary
  2. Convenience
- One digit represents 4 bits
- FA are 8 bits binary 1111 1010
- FA is decimal 250
- Decimal to binary not as easy
- Prefix 0x is used for hexadecimals
- For example 0x241FF50

4 bits binary	hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

# Memory and Addresses

- Here's a picture of RAM.
- All variables are stored in memory, each at its own unique address or location.
- For example

```
int count = 5;
```

- The variable count is associated with memory address 0x241FF54
- The value “5” is stored in memory.
- It can be accessed by using the variable name “count”.
- **Or it be accessed by reference to its address.**

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

Every time the program is executed the memory location may change.

# Addresses and References

- Accessing memory addresses
  - To access the memory address of a variable, we use the unary operator &
  - & is called an address operator (or address-of) operator.
- Example

```
cout<< &count << endl;
```

- This prints address: 0x241FF54

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

# Addresses and Pointers

- A pointer is a special type of variable
  - **Normal variables contain a data value.**
  - **Pointer variables contains a memory address as its value.**
  - Data is modified when a normal variable is used.
  - The value of the address stored in a pointer is modified when a pointer is used.

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

# Addresses and Pointers

- How to define a pointer

  - `datatype* pointername;`

- Example

```
int* ptr;
```

- This declares a pointer variable named `ptr` of type `int*`

- Note: Some people prefer to write

```
int *ptr
```

- It's the same. Use one, consistently.

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

# Addresses and Pointers

- Usually, the address stored in the pointer is the address of some other variable.
- Example

```
int count = 5;  
int* ptr = &count;
```

- This means:
  - count is a variable with value 5
  - ptr is a pointer with value 0x241FF54

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

# Dereferencing Pointers

- To get the value that is stored at the memory location pointed by the pointer, one would need to **dereference** the pointer.
- Dereferencing means to read the data value at the address the pointer points to.
- Dereferencing allows manipulation of the **data contained at the memory address stored in the pointer**.

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

# Dereferencing Pointers

- Dereferencing is done with the unary operator \* called a **dereference operator**.
- Syntax
  - **\*variablename**

## □ Example

```
int count = 5;  
int* ptr = &count;  
cout << *ptr << endl;
```

- This prints the value at address 0x241FF54. Which is 5.

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

# Assigning to Pointers

## □ Assign an address to a pointer

```
int count = 5;  
int* ptr1 = &count;  
int* ptr2 = ptr1;
```

- ❑ Both pointers, ptr1 and ptr2, point to the same address.

## □ Set value pointed to directly

```
*ptr1 = 6;  
cout << *ptr1; prints 6  
cout << *ptr2; prints 6
```

- ❑ Because both pointers, ptr1 and ptr2, point to the same address.

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	6
	0x241FF58	
	0x241FF5C	

# Pointers and Addresses

## □ Normal variables:

- A normal variable contains a data value

```
int count = 5;
```

- Use & to get its address:

```
cout << count; prints 5
```

```
cout << &count; prints 0x241FF54
```

## □ Pointers:

- A point variable contains an address

```
int* ptr = &count;
```

- Use \* to get the value at that address

```
cout << ptr; prints 0x241FF54
```

```
cout << *ptr; prints 5
```

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

# Example

- Consider the following piece of code:

```
int j = 2;  
  
int * pt1 = &j; //pt1 points to j  
  
cout << *pt1; //prints out value in int j  
  
*pt1 = *pt1 + 2; //adds two to the value  
  
                                //pointed to by pt1 (i.e. int j)
```

- The effect of the above statement is equal to

```
int j = 2;  
  
j = j + 2;
```

Of course, usually you use pointers for more useful things. Not for making  $2 + 2$  complicated.

# Syntax of Pointers

## SYNTAX 7.1 Pointer Syntax

```
double account = 0;  
double* ptr = &account;
```

The type of ptr  
is "pointer to double".

You should always initialize  
a pointer variable, either with  
a memory address or NULL.

The & operator yields a memory address.

The \* operator accesses the location  
to which ptr points.

```
*ptr = 1000  
cout << *ptr;
```

This statement  
changes account  
to 1000.

This statement  
reads from the location  
to which ptr points.

# Pointer Syntax Examples

Table 1 Pointer Syntax Examples

Assume the following declarations:

```
int m = 10; // Assumed to be at address 20300
int n = 20; // Assumed to be at address 20304
int* p = &m;
```

Expression	Value	Comment
p	20300	The address of m.
*p	10	The value stored at that address.
&n	20304	The address of n.
p = &n;		Set p to the address of n.
*p	20	The value stored at the changed address.
m = *p;		Stores 20 into m.
🚫 m = p;	Error	m is an int value; p is an int* pointer. The types are not compatible.
🚫 &10	Error	You can only take the address of a variable.
&p	The address of p, perhaps 20308	This is the location of a pointer variable, not the location of an integer.
🚫 double x = 0; p = &x;	Error	p has type int*, &x has type double*. These types are incompatible.

# Errors Using Pointers

## Uninitialized Pointer Variables

When a pointer variable is first defined,  
it contains a random address.

Using that random address is an **error**.

# Errors Using Pointers

## Uninitialized Pointer Variables

In practice, your program will likely crash or mysteriously misbehave if you use an uninitialized pointer:

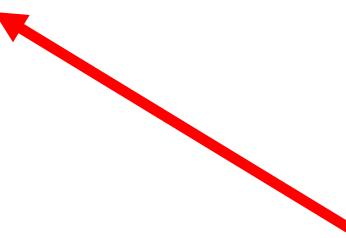
```
double* account_pointer; // No initialization  
*account_pointer = 1000;
```



Where is the 1000 going?

NO!

account\_pointer contains an unpredictable value!



# Errors Using Pointers

## Uninitialized Pointer Variables

There is a special value  
that you can use  
to indicate a pointer  
that doesn't point anywhere:

**NULL**

# NULL

- If you define a pointer variable and are not ready to initialize it quite yet, it is a good idea to set it to **NULL**.
- You can later test whether the pointer is **NULL**.
- If it is, don't use it.
- Example:

```
double* account_pointer = NULL; // Will set later
...
if (account_pointer != NULL)    // OK to use?
{
    cout << *account_pointer;
}
```

# NULL

Warning:

**Trying to access data through a NULL pointer is still illegal,  
and  
it will cause your program to crash.**

```
double* account_pointer = NULL;  
cout << *account_pointer;
```

CRASH!!!

# NULL

Warning:

**Trying to access data through a NULL pointer is still illegal,  
and  
it will cause your program to crash.**

**Accidentally dereferencing a NULL Pointer is a serious  
problem you want to avoid at all costs.**

# Common Error: Confusing Data And Pointers

A pointer is a memory address

- a number that tells where a value is located in memory.

It is a common error to confuse the pointer  
with the variable to which it points.

# Common Error: Confusing Data And Pointers

A pointer tells where a rabbit  
is located in the field.



It is a common error to confuse the pointer  
with the rabbit to which it points.



# Common Error: Where's the \*?

```
double* account_pointer = &joint_account;  
account_pointer = 1000;
```

ERROR

The assignment statement does not set the joint account balance to 1000.

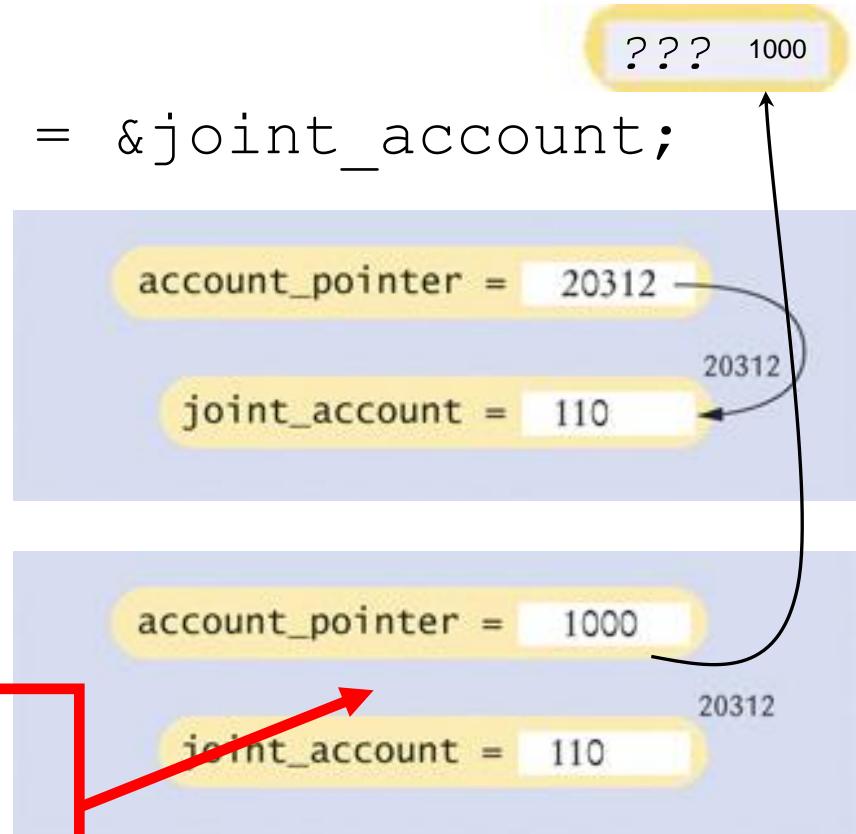
It sets the pointer variable, account\_pointer, to point to memory address 1000.

# Common Error: Where's the \*?

```
double* account_pointer = &joint_account;
```

```
account_pointer = 1000;
```

joint\_account is almost certainly  
not located at address 1000!



# Common Error: Where's the \*?

Most compilers will report an error for this kind of error.

# Confusing Definitions

It is legal in C++ to define multiple variables together, like this:

```
int i = 0, j = 1;
```

This style is confusing when used with pointers:

```
double* p, q;
```

The \* associates only with the first variable.

That is, p is a double\* pointer, and q is a double value.

To avoid any confusion, it is best to define each pointer variable separately:

```
double* p;  
double* q;
```

# Pointers and References

& == \*

?

What are you asking?

# Pointers and References

Recall that the  symbol is used for reference parameters:

```
void withdraw(double& balance, double amount)
{
    if (balance >= amount)
    {
        balance = balance - amount;
    }
}
```

a call would be:

```
withdraw(account, 1000);
```

# Pointers and References

We can accomplish the same thing using pointers:

```
void withdraw(double* balance, double amount)
{
    if (*balance >= amount)
    {
        *balance = *balance - amount;
    }
}
```

but the call will have to be:

```
withdraw(&account, 1000);
```

# Summary

- Pointer are variables that refer to addresses in memory
  - Use **datatype\*** to define pointers.
  - You obtain the value stored at the location a pointer points to by dereferencing it.
  - The dereferencing is done using the **\*** operator.
  - Careful about NULL and undefined pointers. Very careful.
  - You can get the address of a **normal variable** by using the address-of operator &
  - You can assign addresses to pointers. With appropriate type.

# Arrays and Pointers

In C++, there is a deep relationship between pointers and arrays.

This relationship explains a number of special properties and limitations of arrays.

# Arrays and Pointers

Pointers are particularly useful for understanding the peculiarities of arrays.

The name of the array denotes a pointer to the starting element.

# Arrays and Pointers

Consider this declaration:

```
int a[10];
```

(Assume we have filled it as shown.)

You can capture the  
pointer to the first  
element in the array  
in a variable:

```
int* p = a; // Now p points to a[0]
```

a	0	20300
	1	20308
	4	20316
	9	20324
	16	20332
	25	20340
	36	20348
	49	20356
	64	20364
	81	20372

**p = 20300**

# Arrays and Pointers – Same Use

You can use the array name `a` as you would a pointer:

These output statements are equivalent:

```
cout << *a;  
cout << a[0];
```



These output statements as well:

```
cout << a;  
cout << &a[0];
```



# Pointer Arithmetic

Pointer arithmetic allows you to add an integer to an array name.

```
int* p = a;
```

$p + 3$  is a pointer to the array element with index 3

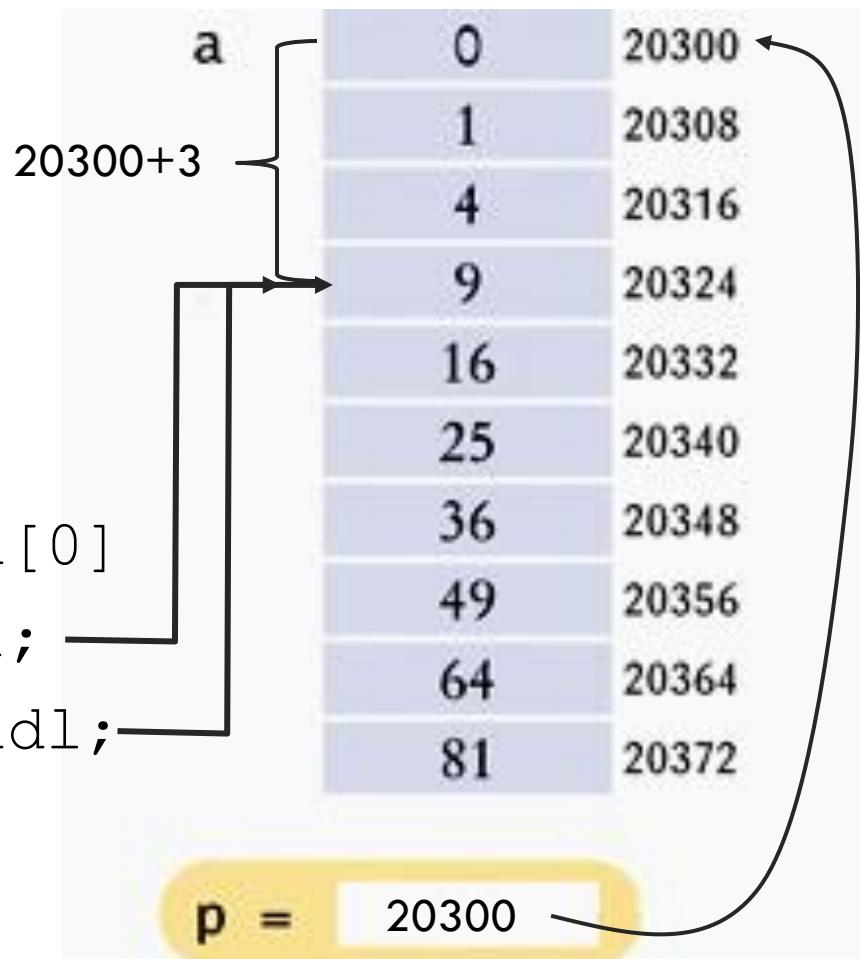
The expression:  $* (p + 3)$   
is the same as:  $a[3]$

# Pointer Arithmetic

The expression:  $* (p + 3)$   
is the same as:  $a[3]$

Really.

```
int a[10];  
int* p = a; // or &a[0]  
cout << a[3] << endl;  
cout << * (p+3) << endl;
```



# The Array/Pointer Duality Law

The array/pointer duality law states:

**$a[n]$  is identical to  $*(\mathbf{a} + n)$**

where **a** is a pointer into an array  
and **n** is an integer offset.

# The Array/Pointer Duality Law

This law explains why all C++ arrays start with an index of zero.

The pointer `a` (or `a + 0`) points to the starting element of the array.

That element must therefore be `a[0]`.

You are adding 0 to the start of the array, thus correctly going nowhere!

a	0	20300
	1	20308
	4	20316
	9	20324
	16	20332
	25	20340
	36	20348
	49	20356
	64	20364
	81	20372

`p = 20300`

# The Array/Pointer Duality Law

Now it should be clear why array parameters  
are different from other parameter types.

(if not, we'll show you)

# The Array/Pointer Duality Law

Consider this function that computes  
the sum of all values in an array:

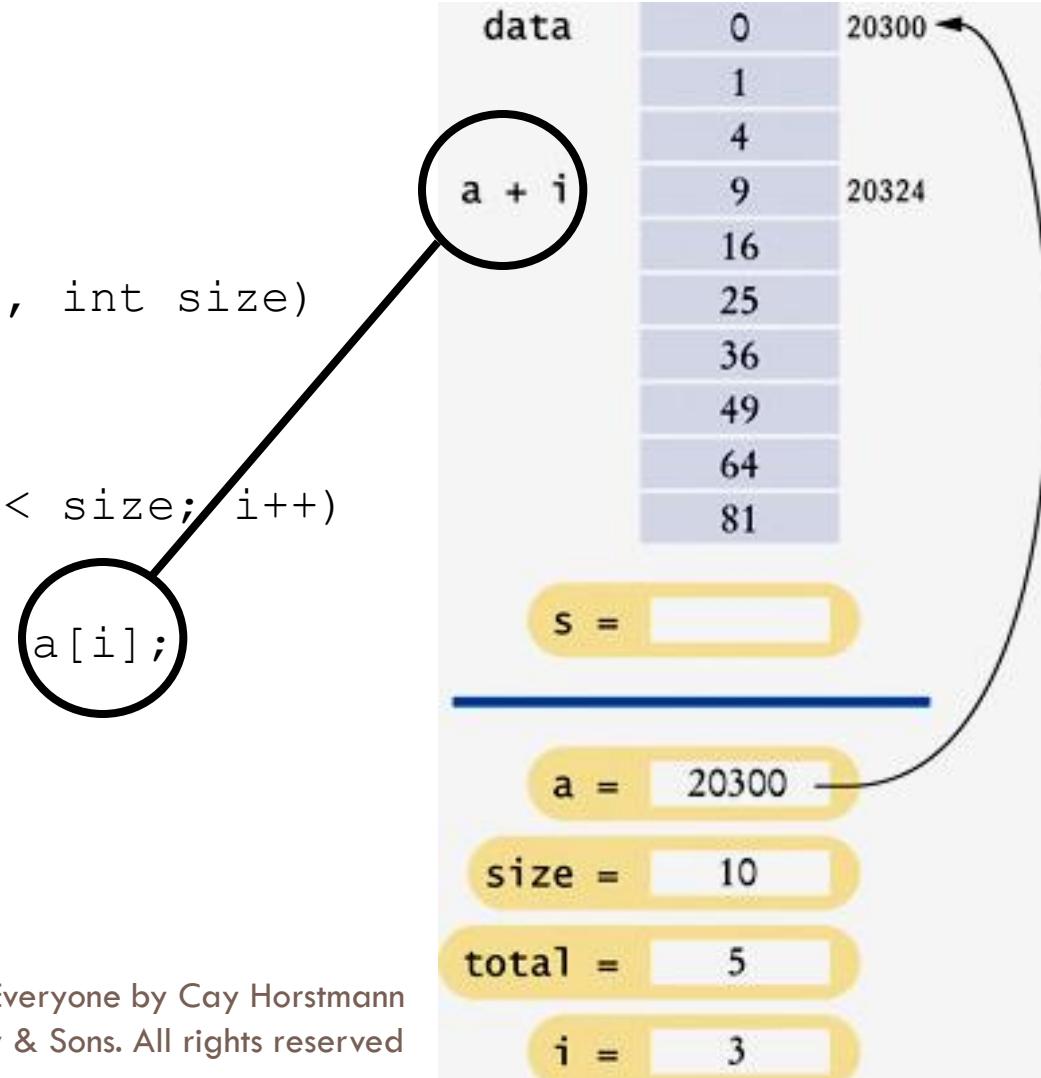
Look at this

```
double sum(double a[], int size)
{
    double total = 0;
    for (int i = 0; i < size; i++)
    {
        total = total + a[i];
    }
    return total;
}
```

# The Array/Pointer Duality Law

After the loop has run  
to the point when  $i$  is 3:

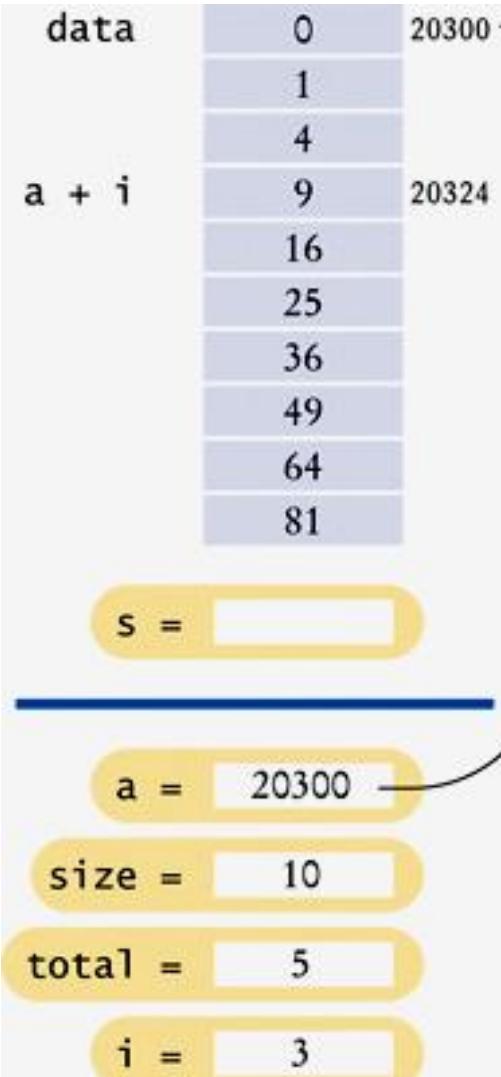
```
double sum(double a[], int size)
{
    double total = 0;
    for (int i = 0; i < size; i++)
    {
        total = total + a[i];
    }
    return total;
}
```



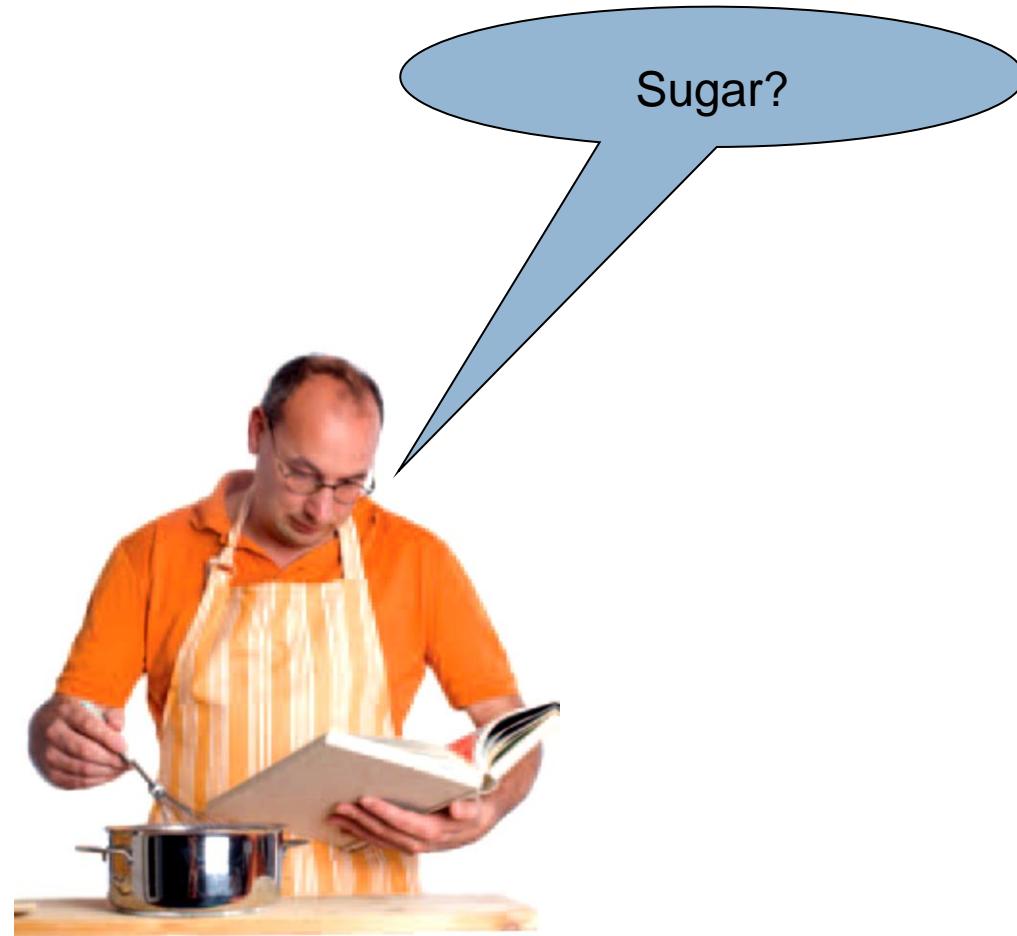
# The Array/Pointer Duality Law

The C++ compiler considers  
a to be a pointer, not an array.

The expression `a[i]`  
is syntactic sugar  
for `*(a + i)`.



# Syntactic Sugar



# Syntactic Sugar

Computer scientists use the term

“syntactic sugar”

to describe a notation that is easy to read for humans  
and that masks a complex implementation detail.

Yum!

# Syntactic Sugar



# Syntactic Sugar

That masked complex implementation detail:

`double sum(double* a, int size)`

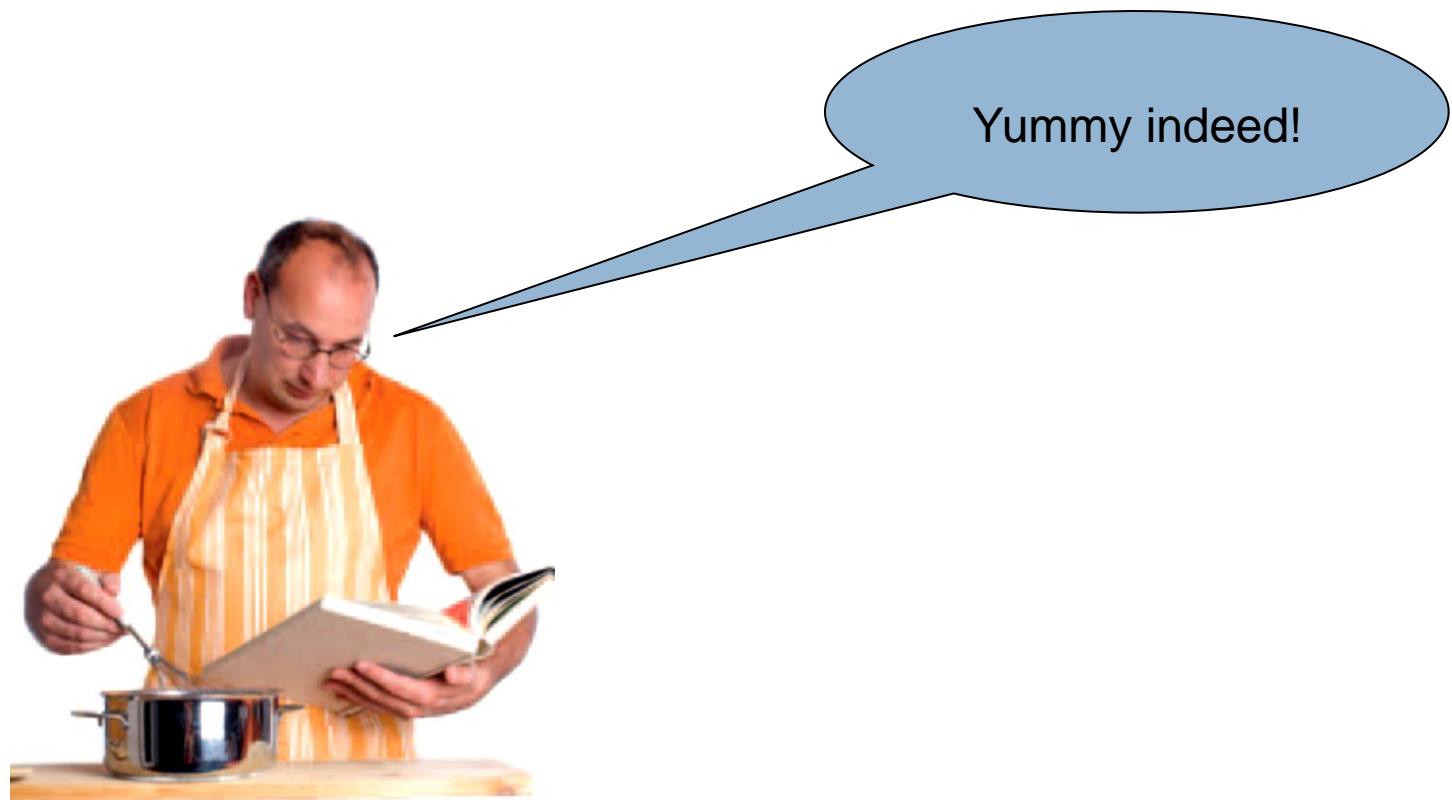
is how we should define the first parameter

but

`double sum(double a[], int size)`

looks a lot more like we are passing an array.

# Syntactic Sugar



# Syntactic Sugar

The **is what the function would look like using pointer notation:**

```
double sum(double* a, int size)
{
    double total = 0;
    for (int i = 0; i < size; i++)
    {
        total = total + *(a+i);
    }
    return total;
}
```

# Arrays and Pointers

**Table 2 Arrays and Pointers**

Expression	Value	Comment
a	20300	The starting address of the array, here assumed to be 20300.
*a	0	The value stored at that address. (The array contains values 0, 1, 4, 9, ....)
a + 1	20308	The address of the next double value in the array. A double occupies 8 bytes.
a + 3	20324	The address of the element with index 3, obtained by skipping past $3 \times 8$ bytes.
*(a + 3)	9	The value stored at address 20324.
a[3]	9	The same as *(a + 3) by array/pointer duality.
*a + 3	3	The sum of *a and 3. Since there are no parentheses, the * refers only to a.
&a[3]	20324	The address of the element with index 3, the same as a + 3.

# Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
    }
    return total;
}
```

a	0	20300
	1	20308
	4	20316
	9	20324
	16	20332
	25	20340
	36	20348
	49	20356
	64	20364
	81	20372

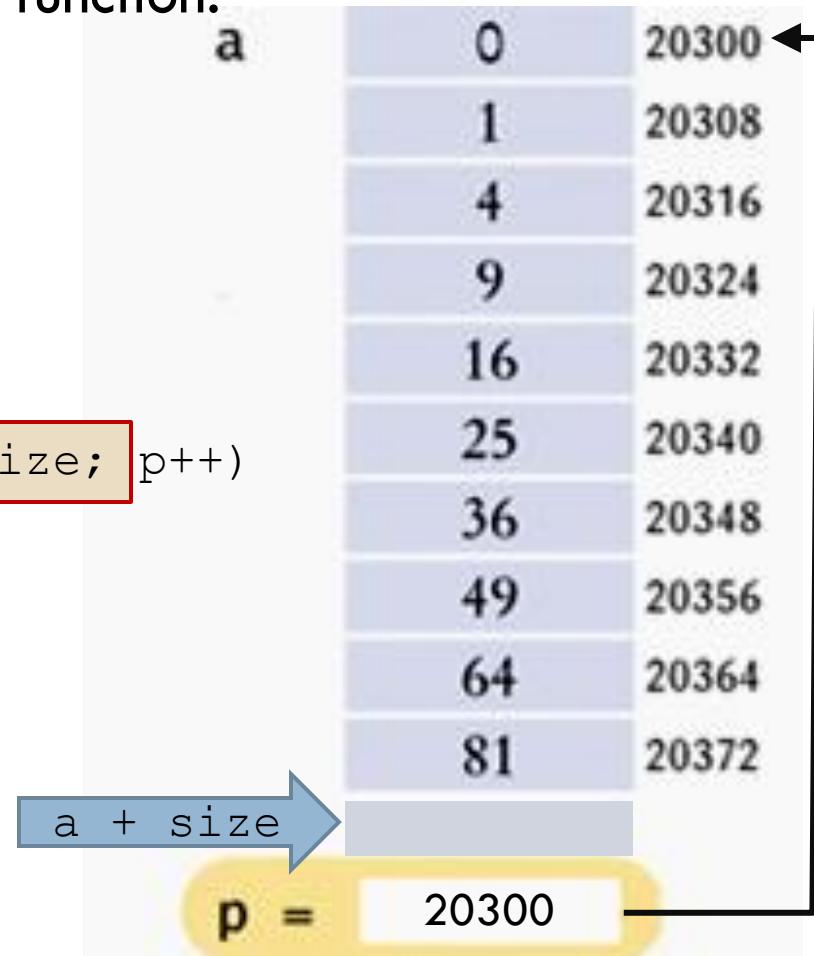
p = 20300

# Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
    }
    return total;
}
```

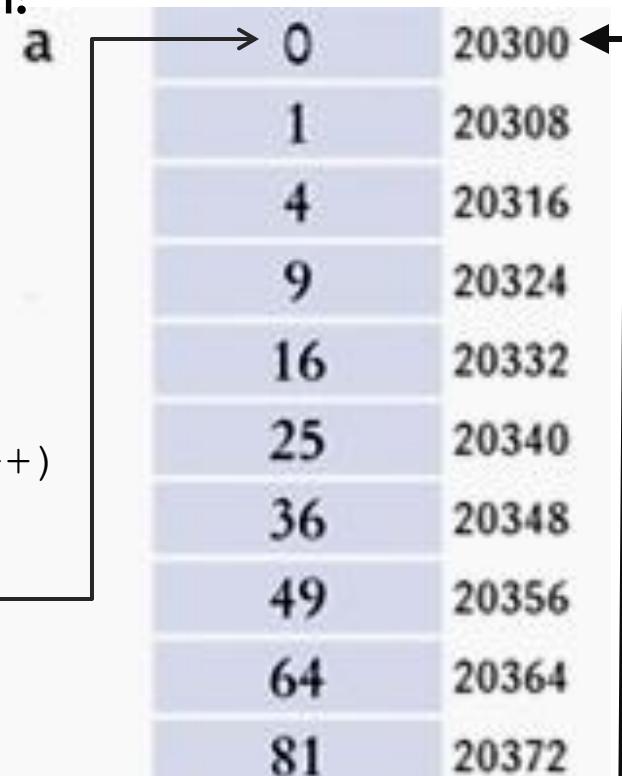


# Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
    }
    return total;
}
```



**p** = 20300

# Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
    }
    return total;
}
```

a	0	20300
	1	20308
	4	20316
	9	20324
	16	20332
	25	20340
	36	20348
	49	20356
	64	20364
	81	20372

p = 20300

# Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
    }
    return total;
}
```

a	0	20300
	1	20308
	4	20316
	9	20324
	16	20332
	25	20340
	36	20348
	49	20356
	64	20364
	81	20372

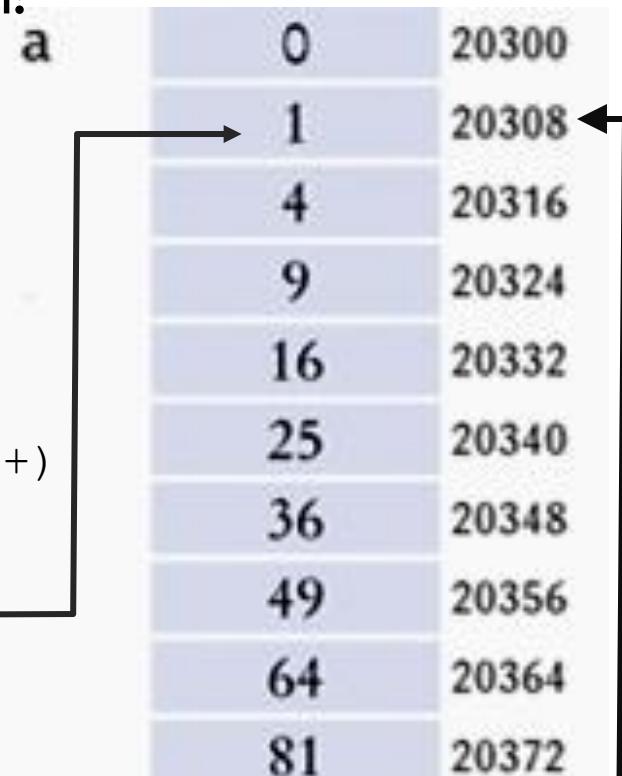
p = 20300

# Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
    }
    return total;
}
```



p = 20300

# Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
    }
    return total;
}
```

a	0	20300
	1	20308
	4	20316
	9	20324
	16	20332
	25	20340
	36	20348
	49	20356
	64	20364
	81	20372

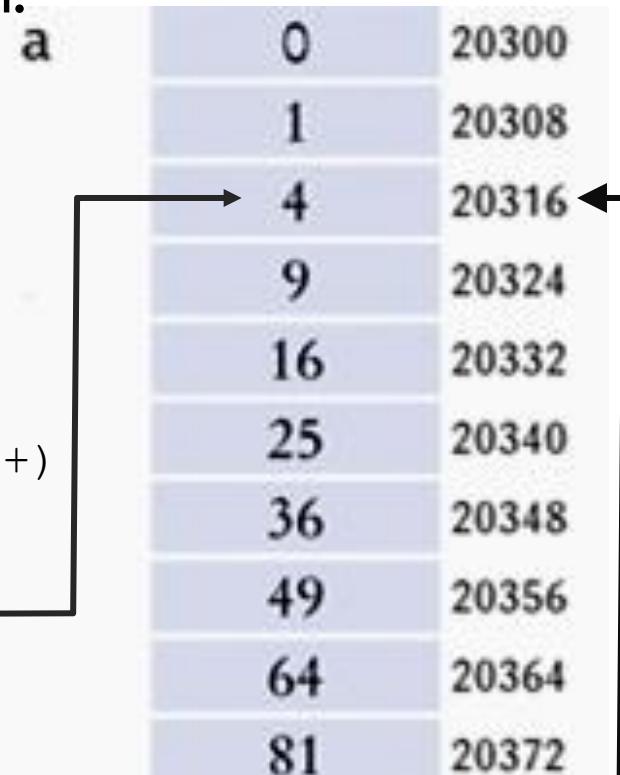
p = 20300

# Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
    }
    return total;
}
```



Etcetera

p = 20300

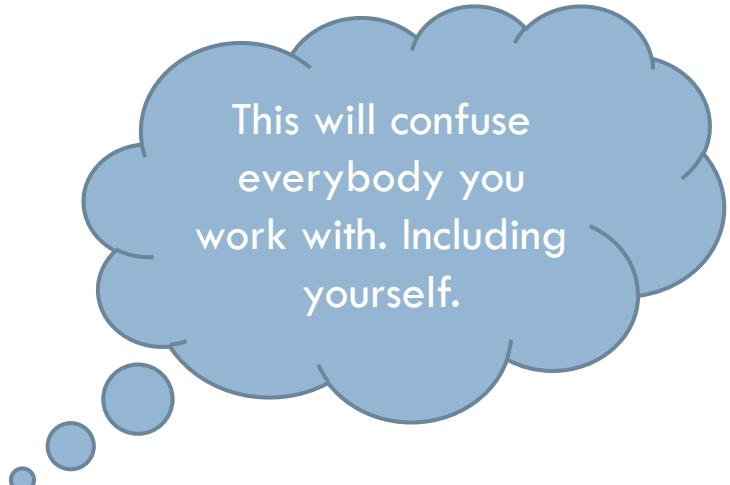
# Program Clearly, Not Cleverly

Some programmers take great pride  
in minimizing the number of instructions,  
even if the resulting code is hard to understand.

```
while (size > 0)
{
    total = total + *p;
    p++;
    size--;
}
```

could be written as:

```
while (size-- > 0)
    total = total + *p++;
```



This will confuse  
everybody you  
work with. Including  
yourself.

# Program Clearly, Not Cleverly



**Please do not use this programming style.**

**Your job as a programmer is not to dazzle other programmers with  
your cleverness,  
but to write code that is easy  
to understand and maintain.**

# Common Error: Returning a Local Pointer

Consider this function that tries to return a pointer to an array containing two elements, the first and last values of an array:

```
double* firstlast(double a[], int size)
{
    double result[2];
    result[0] = a[0];
    result[1] = a[size - 1];
    return result;
}
```

Local memory is invalid after the function call has ended!

What would the value the caller gets be pointing to?

# Common Error: Returning a Local Pointer

A solution would be to pass  
in an array to hold the answer:

```
void firstlast(double a[], int size,  
               double result[])  
{  
    result[0] = a[0];  
    result[1] = a[size - 1];  
}
```

# Common Error: Returning a Local Pointer

Or a pointer.

Remember the duality:

```
void firstlast(double* a, int size,  
               double* result)  
{  
    *result = *a;  
    *(result+1) = *(a + size - 1);  
}
```

Not nearly as nice. Use some syntactic sugar.

# Summary

- The name of an arrays is a pointer
- It points to the first element of the array.
- $a[n]$  is identical to  $*(a + n)$ , where  $a$  is a pointer into an array and  $n$  is an integer offset.
- Don't try to be too clever.

# C and C++ Strings

More things we didn't tell you before:

C++ has two mechanisms for manipulating strings.

# C and C++ Strings

C++ has two mechanisms for manipulating strings.

- The string class
  - Supports character sequences of arbitrary length.
  - Provides convenient operations such as concatenation and string comparison.
- C strings
  - Provide a more primitive level of string handling.
  - Are from the C language (C++ was built from C).
  - Are represented as arrays of char values.

# Recap on characters

- The type `char` is used to store an individual character.
- Some of these characters are plain old letters and such as  
`'y'`, `'n'`, `'3'`, `'?'`
- Some of them are escape characters such as:  
`'\n'`, `'\t'`, `'\a'`
- And then there is the special character `'\0'` to denote the end of a string, the **null terminator**.

# Characters

**Table 3 Character Literals**

'y'	The character y
'0'	The character for the digit 0. In the ASCII code, '0' has the value 48.
' '	The space character
'\n'	The newline character
'\t'	The tab character
'\0'	The null terminator of a string
 "y"	<b>Error:</b> Not a char value

# C strings

- C *strings* are arrays of characters.
- **Include** #include <cstring>
- The null always the last character in a C string.
- Literal strings are always stored as character arrays
- Example:
  - "CAT" is really this sequence of characters: 'C' 'A' 'T' '\0'
  - The null terminator character indicates the end of the C string
  - The literal C string "CAT" is actually an array of four chars stored somewhere in the computer.

# Pop Quiz #1.

Q:

Is "C string" a string?

Yes

...wait...

No

...wait...

# Pop Quiz #1

Answer:

"C string" is NOT an object of **string** type.

"C string" IS an **array of chars** with a null terminator character at the end.

“C string” is a **C string**.

# Character Arrays as Storage for C Strings

As with all arrays, a string literal can be assigned to a pointer variable that points to the initial character in the array:

```
char* char_pointer = "Harry"; // Points to the 'H'
```

char\_pointer = 320300

Points to 'H'

null terminator

'H'	[0]	320300
'a'	[1]	320301
'r'	[2]	320302
'r'	[3]	320303
'y'	[4]	320304
'\0'	[5]	320305

# Using the Null Terminator Character

Functions that operate on C strings rely on this terminator.

The `strlen` function returns the length of a C string.

```
#include <cstring>
int strlen(const char s[])
{
    int i = 0;
    // Count characters before the null terminator
    while (s[i] != '\0') {
        i++;
    }
    return i;
}
```

# Using the Null Terminator Character

The call `strlen("Harry")` returns 5.

The null terminator character is not counted as part of the “length” of the C string – but it’s there.

Really, it is.

# C String Functions

Table 4 C String Functions

In this table, `s` and `t` are character arrays; `n` is an integer.

Function	Description
<code>strlen(s)</code>	Returns the length of <code>s</code> .
<code>strcpy(t, s)</code>	Copies the characters from <code>s</code> into <code>t</code> .
<code>strncpy(t, s, n)</code>	Copies at most <code>n</code> characters from <code>s</code> into <code>t</code> .
<code>strcat(t, s)</code>	Appends the characters from <code>s</code> after the end of the characters in <code>t</code> .
<code>strncat(t, s, n)</code>	Appends at most <code>n</code> characters from <code>s</code> after the end of the characters in <code>t</code> .
<code>strcmp(s, t)</code>	Returns 0 if <code>s</code> and <code>t</code> have the same contents, a negative integer if <code>s</code> comes before <code>t</code> in lexicographic order, a positive integer otherwise.

# C String Functions

- Warning:
  - Many C string function have to be used with care.
  - If used incorrectly, they can allow attackers to write to your memory.
  - Many organizations advise to avoid functions such as strcpy altogether.
  - See for example:
    - <http://cwe.mitre.org/data/definitions/676.html>
    - <http://cwe.mitre.org/data/definitions/120.html>
    - <http://cwe.mitre.org/data/definitions/170.html>

# C++ strings

- C++ has a `<string>` library defining the string class
- Include it in your programs when you wish to use strings:

```
#include <string>
```

- This library makes string processing easier than in C
- Notice there is no “.h” in the C++ string header;  
`<string.h>` is used for C-style strings
- You define a string object as follows:

```
string first_name = "Pete";
```



What's  
an  
object?



We'll tell  
in a few  
weeks

# C++ strings

- The string library provides many useful functions for
  - manipulating string data,
  - comparing strings,
  - searching strings for characters and other strings,
  - tokenizing strings (separating strings into logical pieces),
  - determining the length of strings.

# C++ strings

## □ Example: Concatenation

- To concatenate two strings, we use the “+” operator

```
string str1= "Hi";  
  
string str2= "5";  
  
string str3 = str1 + str2;  
  
cout<<"str3 = "<<str3<<endl; //displays Hi5  
  
str3 += "!";  
  
cout<<"str3 = "<<str3<<endl; //displays Hi5!
```

# C++ strings

- Strings are compared using the following operators:  
==, !=, <, <=, >, >=
- The comparison uses the alphabetical order
- The result is a Boolean value: *true* or *false*
- The comparison works as long as at least one of the two arguments is a string object. The other string can be a string object, a C-style string (char array).
- Example:

```
if(str == "Hi5!") { ... }
```

# Comparing <cstring> and <string>

C Library Functions	C++ string operations
strcpy	=
strcat	+=
strcmp	= =, !=, <, >, <=, >=
strlen	.size( )
str[i]	str.substr(i,1);
...	...

# Converting Between C and C++ Strings

- To convert a C++ string object to a C string you can use the member function `c_str` (use dot notation)
- Example

```
string cppstr = "Welcome";
char cstr[8]; // 7 for 'Welcome', plus 1 for '\0'
strcpy (cstr, cppstr.c_str());
```

# Converting Between C and C++ Strings

- Converting from a C string to a C++ string is very easy:
  - You can just assign a C string (char array) to a string object.
  - Example: `string name = "Harry";`
  - `name` is initialized with the C string "Harry".

# Summary

C++ has two mechanisms for manipulating strings.

- The string class
  - Supports character sequences of arbitrary length.
  - Provides convenient operations such as concatenation and string comparison.
- C strings
  - Provide a more primitive level of string handling.
  - Are from the C language (C++ was built from C).
  - Are represented as arrays of char values.

# Lecture 4.1

## Classes and Objects

CS112, semester 2, 2007

1

## Top-Down Design Approach

- In this approach the problem is successively broken down into smaller tasks or functions.
- The result is a tree structure where the top is main task and the nodes further down the tree are the sub-tasks and functions.

CS112, semester 2, 2007

3

## Object-oriented design

- The system is divided into objects which interact with each other by sending messages.
- Object-oriented programming (OOP) *encapsulates* data (attribute) and functions (behavior) into packages called *classes*.
- Each object has a set of properties/ attributes and behavior/methods.
- In C++ these properties are like variables and behaviors are like functions.

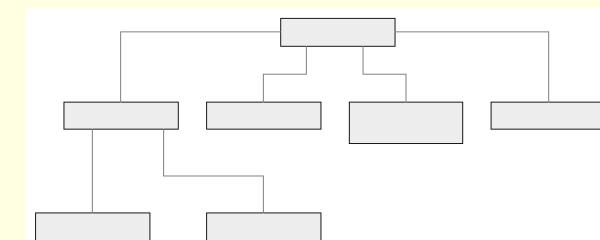
CS112, semester 2, 2007

2

## Top-Down Design Approach

### Example:

- A program to calculate the final grade can be broken down in the following sub-tasks (functions)



CS112, semester 2, 2007

4

## Advantages

- Programs are modular
- Easy to keep track of tasks and easy to maintain programs
- Objects can be **reused** in other projects

## Classes and object (cont)

- Each class contains data as well as the set of functions that manipulate the data.
- The data components of a class are called data members or *properties*.
- The function components of a class are called member functions or *methods* or *behaviors*.

## Classes and object

- Classes are like blueprints of which objects are created. In other words the class specifies the attributes and methods which every object of that particular class will contain.
- In simple terms classes are user defined data types and objects are just variables ( I will say complex variables).
- Just as an instance of a built-in type such as int is called a variable, and instance of a user-defined data type is called an object.

## Example

- What are some attributes/properties of a car?
  - A car has color, model, registration number, engine capacity
- What are some actions you can perform on a car?
  - You can start it, drive it, stop it.

## Example

- The car class describes just about any car in the world. Now consider a specific car, say James Bond's car.
  - It is grey BMW with 3L V8 engine and of course the registration number is 007.
  - You can start, drive and stop his car.
- In programming terms Mr. bond's car is an object or an instance of the car class.



## What are classes?

- Classes are a software construct that can be used to emulate a real world object.
- Classes encapsulate data (attributes) and abilities (functions/methods).

```
int x; //Declares x to be a variable of type int.  
Car pajero; //Declares pajero to be an object of class  
//Car.
```

## Structs and Classes

- So what is the difference between classes and structs?
- Both are user defined types
- Both are used for data encapsulation
- Both creates objects (variables)
- Structs are used to store data only but classes are used to store data as well as functions (methods).
- Classes are just enhanced structs.

## Lecture 4.2

### Classes and Objects

CS112, semester 2, 2007

1

### How to define a class (cont)

```
class Dog {  
    public:  
        void setAge(int age);  
        int getAge();  
        void setWeight(int weight);  
        int getWeight();  
        void speak();  
    private:  
        int age;  
        int weight;  
};
```

CS112, semester 2, 2007

3

### How to define a class?

- Using the keyword **class** followed by a programmer-specified name followed by the class definition in braces.
- The class definition contains the class members (data) and the class methods (functions).

CS112, semester 2, 2007

2

### Some important concepts

- **private** indicates that the two members, age and weight, cannot be directly accessed from outside of the class.
- **public** indicates that the methods, can be called from code outside of the class. They may be called from other parts of a program.
- Allowing access and manipulation of data members only through methods is referred to as **data hiding**.

CS112, semester 2, 2007

4

## Method implementation

- The methods were declared but not defined. That is, an implementation for each method must be written.

```
void Dog::setAge(int age)  
{ this->age = age; }
```

```
int Dog::getAge() { return age; }
```

## Some important things!

- The methods are implemented outside of the class definition, they must be identified as belonging to that class.
- This is done with the scope resolution operator, "::".
- Every object has a special pointer call "this", which refers to the object itself.
- The members of the Dog class can be referred to as this->age or this->weight, as well as, age or weight.

## Method implementation (cont)

```
void Dog::setWeight(int weight)  
{ this->weight = weight; }
```

```
int Dog::getWeight()  
{ return weight; }
```

```
void Dog::speak()  
{ cout << "BARK!!" << endl; }
```

## And some more important things (cont)

- If there is no ambiguity, no qualification is required.
- In the getWeight method, "weight" can be used instead of "this->weight".  

```
int Dog::getWeight()  
{ return weight; }
```
- Here, the scope resolution operator must be used.

```
void Dog::setWeight(int weight)  
{ this->weight = weight; }
```

## Data Abstraction

- Describing the functionality of a class independent of its implementation is called data abstraction and c++ classes define so-called abstract data types.
- We make project in Dev C++ to separate declaration of methods with their implementations. Technique of strictly hiding the implementations is outside the scope of this course.
- This is done to hide the implementation details from the clients of the classes.

## Destructor

- Another special method, the destructor, is called when an object is destroyed.
- An object is destroyed when it goes out of scope.
- If an object is created within a function, it will go out of scope when the function exits.

## Constructors

- Each class also has a special method, the constructor, which is called when an object of the class is instantiated (created).
- The constructor can be used to initialize variables, dynamically allocate memory or setup any needed resources.

## Destructor (cont)

- Since your program is the "main" function, all its objects go out of scope when the program ends.
- The destructor is used to free any memory that was allocated and possibly release other resources.

# Lecture 4.3

## Classes and Objects

CS112, semester 2, 2007

1

## Constructor facts (cont)

- It is possible to have multiple constructors that differ in their number and/or type of parameters.
- The constructor that is used is based on the arguments used in its invocation. This is referred to as function or method overloading.

```
Dog::Dog()  
{ age = 0;  
    weight = 0; }  
  
Dog::Dog(weight)  
{ age = 0;  
    this->weight=weight; }
```

CS112, semester 2, 2007

3

## Example of constructor and destructor

```
class Dog {  
public:  
    Dog(); //Constructor  
    ~Dog(); //Destructor  
    void setAge(int age); ...  
}  
Dog::Dog(){  
    age = 0;  
    weight = 0  
    cout << "Dog Constructor Called" << endl;  
}  
Dog::~Dog()  
{ cout << "Dog Destructor Called" << endl; }
```

CS112, semester 2, 2007

2

## Destructor facts

- The destructor has the same name as the class prefixed by a tilde, "~".
- The destructor might free memory that was allocated, release some resources or perform some other clean up activity.

CS112, semester 2, 2007

4

## Using Objects

- The following program declares objects of the Dog class.
- For simplicity, all code will be contained in a single source file
- In larger projects classes are usually kept in separate files from the main program.

CS112, semester 2, 2007

5

## Using Objects Example (cont)

```
public:  
    Dog(); //Constructor  
    ~Dog(); //Destructor  
    void setAge(int age);  
    int getAge();  
    void setWeight(int weight);  
    int getWeight();  
    void speak();  
};
```

CS112, semester 2, 2007

7

## Using Objects Example

```
#include <iostream.h>  
  
class Dog {  
private:  
    int age;  
    int weight;
```

CS112, semester 2, 2007

6

## Example (cont2)

```
Dog::Dog()  
{  
    age = 0;  
    weight = 0;  
    cout << "Dog Constructor Called" << endl;  
}  
Dog::~Dog()  
{  
    cout << "Dog Destructor Called" << endl;  
}
```

CS112, semester 2, 2007

8

## Example (cont3)

```
void Dog::setAge(int age){  
    this->age = age;}  
int Dog::getAge(){  
    return age;}  
void Dog::setWeight(int weight){  
    this->weight = weight;}  
int Dog::getWeight(){  
    return weight;}  
void Dog::speak(){  
    cout << "BARK!!" << endl;}
```

CS112, semester 2, 2007

9

## Example (cont5)

```
cout << "Rover is " << rover.getAge() << " years old.";  
cout << "He weighs " << rover.getWeight() << " lbs.";  
  
cout << "Fido is " << fido.getAge() << " years old.";  
cout << "He weighs " << fido.getWeight() << " lbs.";  
  
cout << "Setting Fido to be the same as Rover" ;  
fido = rover; //NOTE: Object assignments are not  
//recommended
```

CS112, semester 2, 2007

11

## Example (cont4)

```
int main()  
{  
    Dog fido;  
    Dog rover;  
    cout << "Rover is " << rover.getAge() << " years old." ;  
    cout << "He weighs " << rover.getWeight() << " lbs." ;  
    cout << "Updating Rover's Age and Weight" << endl;  
    rover.setAge(1);  
    rover.setWeight(10);
```

CS112, semester 2, 2007

10

## Example (cont6)

```
cout << "Fido is " << fido.getAge() << " years old." ;  
cout << "He weighs " << fido.getWeight() << " lbs." ;  
  
rover.speak();  
fido.speak();  
  
return 0;  
}
```

CS112, semester 2, 2007

12

## Question

---

- write a class definition for circle.
- First step is to think of all the properties that a circle has in 2D cartesian plane.
- To draw or make a circle you need 2 things: center and radius.
- So you have 2 properties of a circle.

## Lecture 4.2

### C-Style Structs

CS112, semester 2, 2011

1

### What are structures?

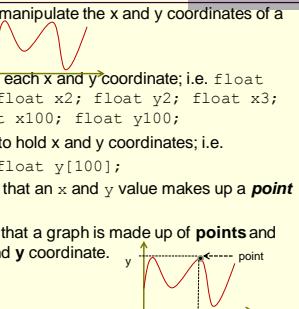
- A structure is a complex data type built by using elements of other types.
- A structure provides a way of grouping variables under a single name for easier handling and identification.
- Structures may be copied to and assigned. They are also useful in passing groups of logically related data into functions.

CS112, semester 2, 2011

2

### Why use structures?

- Consider a program to manipulate the x and y coordinates of a graph.
- Two known ways:
  - Create variables for each x and y coordinate; i.e. float x1; float y1; float x2; float y2; float x3; float y3;...float x100; float y100;
  - Use parallel arrays to hold x and y coordinates; i.e. float x[100]; float y[100];
- It could be easily noted that an x and y value makes up a **point** on the graph.
- Hence, it could be said that a graph is made up of **points** and each **point** has an x and y coordinate.



CS112, semester 2, 2011

3

### How to declare a structure?

- A structure is declared by using the keyword **struct** followed by a **structure tag** followed by the body of the structure containing the structure **members**. E.g.

```
struct keyword
struct point {
    structure tag - defines the
    float x;           name of the structure type
    float y;
    structure members - defines the characteristics
};                      or attributes of the structure; i.e. a point has an
                        x and y coordinate part
```

CS112, semester 2, 2011

4

### How can I use this?

- The preceding structure definition does not reserve any space in memory; rather the definition just creates a new data type that is used to declare variables.
- The **struct** declaration is a **user-defined data type**.
- This means that with a **struct**, YOU can, define YOUR OWN data types.
- Structure variables are declared like variables of other types:
  - point left, right; // **left** and **right** are **objects** (or // **instances**) of type **point**
- is analogous to
  - float rate; // **rate** is an **instance** (**object**) of type **float**

CS112, semester 2, 2011

5

### Accessing Members of the Structure

- The individual members of a structure can be accessed using the **dot operator** **"."**, which is a special type of a **member access operator**.

#### Syntax

object.membername;  
Object or instance of structure      Dot operator      Name of the variable member to be accessed

E.g.

```
point location;
location.x = 2.5;
location.y = 3.5;
cout << "The x coordinate is " << location.x;
cout << "The y coordinate is " << location.y;
```

CS112, semester 2, 2011

1

## Arrays of Structures

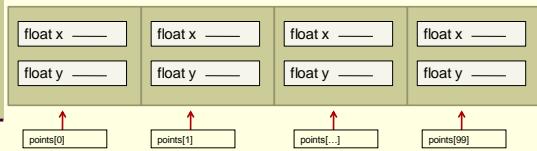
- Let's refer to our program to manipulate the x and y coordinates of a graph.
- Instead of creating two arrays for the x and y coordinates, we could create an array of points, since, we now have a data structure that could hold two floating point number in one structure.
- That is declare

```
point points[100]; //this creates an array of size 100 of
//type point; i.e. 100 point elements
```

7

CS112, semester 2, 2011

## Memory representation of Struct



CS112, semester 2, 2011

8

## Accessing Members of Array

- Arrays are grouping of objects of the same type. Thus, an array of points, e.g. `point points[100];` will have a **point object** in each element of the array, and each point object has an **x** and **y** data member.
- E.g. to display all points

```
for (int i=0; i < 100; i++){
    cout << "x" << i << " is " << points[i].x << endl;
    cout << "y" << i << " is " << points[i].y << endl;
}
```

points[i] reference the point object stored in the 1<sup>st</sup> index of points array. ".y" accesses the y data member

CS112, semester 2, 2011

9

## Nesting structures

- A structure could also be composed of another structure
- A rectangle could be represented as follows.

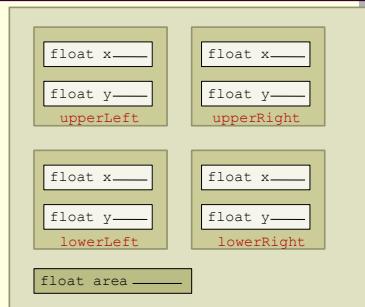
```
struct Rect
{
    point upperLeft;
    point upperRight;
    point lowerLeft;
    point lowerRight;
    float area;
};
```
- This means that each `Rect` object will have 5 components
- To instantiate a `Rect` object we do:

```
Rect myRectangle;
```

CS112, semester 2, 2011

10

## Memory representation of nested Struct



myRectangle

CS112, semester 2, 2011

11

## Accessing nested structs

- To access the members of the struct within a struct, we use the "dot" operator in the nested manner:

```
myRectangle.upperLeft.x = 1.0;
myRectangle.lowerRight.y = 3.0;
```

myRectangle is the object that we want to access. Remember each `Rect` object has 5 components, thus we need to use the dot operator

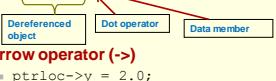
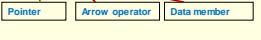
lowerRight is the member object that we want to access within the `myRectangle` object. Remember `lowerRight` is a `point` object and each `point` object has 2 components

y is the member object that we want to access within the `lowerRight` object. Remember y is a `float` object and, thus, has no parts.

CS112, semester 2, 2011

12

## Pointers to Structs

- Structure instances could be manipulated via pointers to the structure type.
  - E.g. `point location;`  
`point *ptrloc = &location; //ptrloc pointing //to object location`
- Two ways to access data members:
  - **Dot operator (.)**
    - `(*ptrloc).x = 1.0;`  

  - **Arrow operator (->)**
    - `ptrloc->x = 2.0;`  


CS112, semester 2, 2011

13

## Pointers to Structs

### E.g.

```
point location;  
point *ptrloc = &location;  
location.x = 2.5;  
ptrloc->y = 3.5;  
cout << "The x coordinate is " << location.x << endl;  
cout << "The y coordinate is " << (*ptrloc).y << endl;
```

CS112, semester 2, 2011

14

## Structs to Functions

- Since structs are data types (even though user-defined), they could be passed to functions in the same manner as the primitive data types (such as int or float) are passed to functions.
- Pointers to structs are passed to functions in the similar fashion as with the primitive types

CS112, semester 2, 2011

15

## Structs to Functions

```
void print_point(point p){ //receiving object  
    cout << "The x coordinate is " << p.x << endl;  
    cout << "The y coordinate is " << p.y << endl;  
}  
  
void print_point(point *ptr){ //receiving pointer  
    cout << "The x coordinate is " << ptr->x << endl;  
    cout << "The y coordinate is " << ptr->y << endl;  
}  
  
int main(){  
    point location;  
    point *ptrloc = &location;  
    location.x = 2.5;  
    ptrloc->y = 3.5;  
  
    print_point(location); //object passed  
    print_pointptr(ptrloc); //pointer passed  
    print_pointptr(&location); //address of object location passed  
}
```

CS112, semester 2, 2011

16

# Lecture 4.1

## Data Structures

CS112, semester 2, 2010

1

## Data Type and Data Structures

- Computers store and operate upon data. These data can normally be categorized into **types**.
- A typical computer or computer language has certain types that are **native** to it – that is, exist as part of the computer or language (*primitive* or *built-in* types).
- C/C++ has the simple types:

- int
- char
- bool

Variables and constants of these types take on values that allow computer programs to reason, calculate, search, display and so on

CS112, semester 2, 2010

2

## Data Type and Data Structures

- Each type has an associated set of values. These values constitute the type's **domain**.
- Each type has a defined set of **operators** that operate on values of the type.

### E.g. in C/C++

Type	Domain
bool	true (1), false (0)
char	ASCII characters {depends on implementation}
int	-INT_MIN to INT_MAX {depends on implementation}

Type	Operators
bool	&&,   , !, =, ==, ...
char	=, ==, !=, <, >, ...
int	=, ==, !=, <, >, +, -, *, ...

CS112, semester 2, 2010

3

## Data Type and Data Structures

- These two features form the essence of a data type, which could be defined as follows:

### A **data type** is:

1. A **domain** of allowed values, and
2. A set of **operations** on those values.

- All C/C++ examples of data types shown in the previous slide have the property that they are somehow elementary or simple.

- What makes them so?

CS112, semester 2, 2010

4

## Data Type and Data Structures

- Notice for all of them, we normally consider their values to be **atomic**; that is, we consider each to have no parts.
- For e.g. the bool values are *true* and *false* (analogous to bit values, respectively, 1 and 0) or the character value 'a'.
  - They are not decomposable, they have no parts.
- Integer or real values are slightly different. E.g. value 154 is normally considered to be a single atomic quantity; we don't worry about any components.
- But we could decompose value 154 into a sequence of base 10 digits of the following form (if we wished):
  - $154 = 1 * 10^2 + 5 * 10^1 + 4 * 10^0$
- For these types, we can decompose their values, but we usually choose not to do so.

CS112, semester 2, 2010

5

## Data Type and Data Structures

- Thus, the atomic types are those that we consider to have no component parts.
  - A value of an atomic data type is regarded as non-decomposable.
- Notice that for each of the types, C/C++ provides no operators that directly allow us to access a component part of these values.

CS112, semester 2, 2010

6

## Array Data Structure

- Arrays are fundamentally different from the native types.
- Arrays are usually called **structured types**.
- E.g. bool sample[2];
  - sample is a data type in exactly the same way that the native types are.
  - It has a domain of possible values and a set of operations on those values.
  - Domain of sample is:

Type	Possible domain values			
sample[0]	true	true	false	false
sample[1]	true	false	true	false

CS112, semester 2, 2010

7

## Array Data Structure

- These 4 values form its domain. They are different from the values of the simple types in that each value has parts or elements.
- E.g. the array value sample[0] (true), sample[1] (false); has 2 elements (parts) and each has a value taken from the domain of type bool.
- Thus, array sample is called a **structured type**. Each of its values has component elements or parts and these elements are arranged in a pattern with respect to each other; that is, in some **structure**.

CS112, semester 2, 2010

8

## Data Structure

- We can, thus, define a structured data type or data structure as follows:

A **data structure** is a **data type** whose values

  1. Can be decomposed into set of component **elements** each of which is either simple (atomic) or another data structure;
  2. Include a set of associations or relationships (**structure**) involving the component elements.

■ A data structure is a special kind of a data type. Since it is a data type, it must, like any other data type, have a domain of allowable values and a set of operations.

■ We look at structures in more detail in the next lecture.

CS112, semester 2, 2010

9

# Lecture 5.1

## Object Oriented Concepts

CS112, semester 2, 2007

1

### Association (“has a”) relationship

- **Association** represents the ability of one instance to send a message to another instance. This is typically implemented with a pointer or reference instance variable, although it might also be implemented as a method argument, or the creation of a local variable.
- Example:  
A house "has" furniture. Association

CS112, semester 2, 2007

3

### Relationships between classes

Three main types:

- Association
  - “has a” relationship.
  - Two or more classes that interact in some manner.
- Aggregation
  - “has a part” relationship.
  - One class is **constructed from** another.
- Generalization (Inheritance)
  - “is a” relationship.
  - One class is **derived from** another (base class)

CS112, semester 2, 2007

2

### Aggregation (“has a part”) relationship

- **Aggregation** is the typical whole/part relationship. One class is **constructed from** another. There is not much **difference** in the way that the association and aggregation are implemented.
- Example:  
A house "has a part" roof. Aggregation

CS112, semester 2, 2007

4

## Generalization (Inheritance)

- Derived class has more specialization than the base class.
- May **override** methods of the base
- May **add new** methods.

CS112, semester 2, 2007

5

## Example of Inheritance

```
class Animal {  
    private:  
        int itsAge;  
        float itsWeight;  
    public:  
        //Accessor methods  
        void setAge (int value) { itsAge=value; }  
        void setWeight (float value) { itsWeight = value; }  
        int getAge () { return itsAge; }  
        float getWeight () { return itsWeight; }  
        // General methods  
        void move () { cout << "Animal Moving\n" ; }  
        void speak() { cout << "Animal Speaking\n" ; }  
        void eat() { cout << "Animal Eating\n" ; }};
```

CS112, semester 2, 2007

6

## Example of Inheritance (cont)

```
class Duck: public Animal {  
    private:  
        int beakSize;  
    public:  
        // Accessor methods are not included for simplification  
        // this method is said to override the move  
        // method from Animal  
        void move() { cout << "Waddle\n"; }  
        //this method overrides speak from Animal  
        void speak() { cout << "Quack\n"; }  
};
```

CS112, semester 2, 2007

7

## Example of Inheritance (cont2)

- The Duck still has a weight, an age and can still eat, because it **inherits** these from Animal.

```
int main () {  
    Duck ducky;  
    cout << "ducky age before setting = " << ducky.getAge() << endl;  
    ducky.setAge (2);  
    cout << "ducky age after setting = " << ducky.getAge() << endl;  
    cout << "EAT inherited from animal = ";  
    ducky.eat ( );  
    cout << "overriden MOVE = " ;  
    ducky.move ( );  
    cout << "overriden SPEAK = " ;  
    ducky.speak ( );}
```

CS112, semester 2, 2007

8

## Output

```
ducky age before setting = 0
ducky age after setting = 2
EAT inherited from animal = Animal Eating
overridden MOVE          = Waddle
overridden SPEAK         = Quack
Press any key to continue . . .
```

CS112, semester 2, 2007

9

## Private vs Protected

```
class Animal {.....  
    private:  
        float itsWeight;  
};  
  
class Duck { .....  
void print_weight ()  
{ cout<<itsWeight;  
};
```

**Can't**

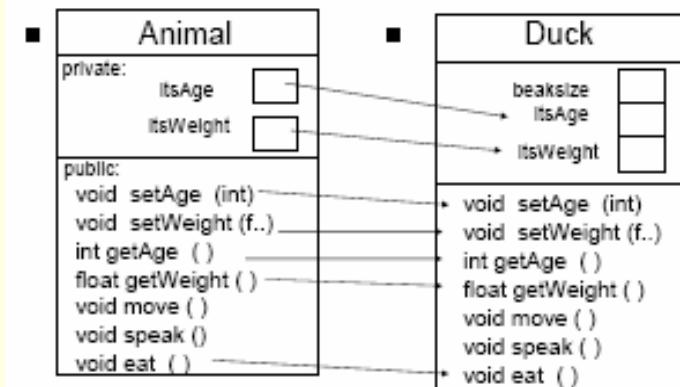
```
class Animal {.....  
    protected:  
        float itsWeight;  
};  
  
class Duck{ .....  
void print_weight ()  
{ cout<<itsWeight;  
};
```

**Can**

CS112, semester 2, 2007

11

## What's happening?



CS112, semester 2, 2007

10

## Inheritance (Another Example)

```
class Pet {  
public:  
    Pet () { weight = 1; food = "Pet Chow";}  
    ~Pet () {}  
    void setWeight (int w) { weight = w; }  
    int getWeight () { return weight; }  
    void setfood (string f) { food = f; }  
    string getFood () { return food; }  
    //General Methods  
    void eat () {cout<<"eating "<<food<<endl;}  
    void speak () {cout<<"Growl"<<endl;}  
protected: // these data can be seen by derived classes  
    int weight;  
    string food;  
};
```

CS112, semester 2, 2007

12

## Lecture 5.2

### Object Oriented Concepts

CS112, semester 2, 2007

1

### Constructor and Destructor inheritance

- Destructors clean up their respective parts.
- Let's see an example with print statements in constructors and destructors to better see how objects are created and destroyed.

CS112, semester 2, 2007

3

### Constructor and Destructor Inheritance

- Constructors, and destructors are NOT inherited.
- Each subtype has its own constructor and destructor.
- Each object of a subtype consists of multiple parts, a base class part and a subclass part.
- The base class constructor forms the base class part.
- The subclass constructor forms the subclass part.

CS112, semester 2, 2007

2

### Protected Members of a Class

- Private members of a class
  - Are private
  - Cannot be accessed outside of the class
  - Even a derived class cannot directly access the private members of a base class
- Protected members –
  - Base class to give access to a member to its derived class

CS112, semester 2, 2007

4

## Friend functions of classes

- Function defined outside the scope of a class
- Is a non-member function of the class, but has access to class's private data members.
- The reserved word **friend** precedes the function prototype in class definition
- Normally used in operator overloading

## friend functions of classes

- Declaration can be placed within private, protected or public part of the class

```
class Student {  
    friend void IcanAccessPrivateMembers(...);  
    ..  
    ..  
};
```

## Definition of a friend function

- Name and scope resolution operator of the class do not precede the name of the friend function heading
- Also the word friend does not appear in the heading of the function.  
*void IcanAccessPrivateMembers(...);*
- However it is implemented in the implementation file

# Lecture 5.3

## Object Oriented Concepts

CS112, semester 2, 2007

1

## Example of same signature

```
class Pet { ...
    void makePetJump (int times) { }
};

class Dog { ...
    void makePetJump (int times) { }
};
```

CS112, semester 2, 2007

3

## Overriding Basics

- A derived class can use the methods of its base class(es), or it can override them
- The method in the derived class must have exactly the **same signature** as the base class method to override.
- The signature is number and type of arguments and the constantness (const, non- const) of the method.
- The **return type** must match the base class method to override.

CS112, semester 2, 2007

2

## Overriding Basics (cont)

- When an object of the base class is used, the base class method is called. When an object of the subclass is used, its version of the method is used if the method is overridden.

CS112, semester 2, 2007

4

## Example

```
int main ( ) {  
    Animal myAnimal;  
    Duck myDuck;  
  
    myAnimal.walk( ); // This will display  
                      // "Animal Walking"  
    myDuck.walk( ); //This will display "Waddle"
```

CS112, semester 2, 2007

5

## Overriding an Overloaded method (cont)

- If the subclass, Cat, defined only  
void speak();
- speak() would be overridden.  
speak(string s) and  
speak(string s, int loudness) would be **hidden**.

CS112, semester 2, 2007

7

## Overriding an Overloaded method

- What is an **overloaded** method?
- A method defined more than once with the same name but different signature.
- For instance, suppose the Pet class had defined several speak methods.

```
void speak();  
void speak(string s);  
void speak(string s, int loudness);
```

CS112, semester 2, 2007

6

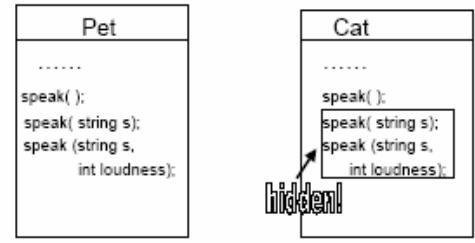
## Overriding an Overloaded method (cont2)

- using Cat fluffy; we could call:  
fluffy.speak();
- But the following would cause compilation errors.  
fluffy.speak("Hello");  
fluffy.speak("Hello", 10);

CS112, semester 2, 2007

8

## Overriding Overloaded methods (cont3)



CS112, semester 2, 2007

9

## Important!

- If you override an overloaded base class method
  - either override every one of the overloads, or carefully consider why you are not.

CS112, semester 2, 2007

10

## Polymorphism

- Definition: The ability for objects of different classes **related by inheritance** to respond differently to the overridden member call.
- Virtual** keyword enables polymorphism.

Example:

```
class base{
public:
    void override() //NOTE: this is NOT a virtual method
    {cout<<"override() from base is called" << endl;}
    virtual void virtual_override()
    {cout<<"virtual_override() from base is called" << endl;}
};
```

CS112, semester 2, 2007

11

## Polymorphism example (cont)

```
class derived: public base{
public:
    void override()
    {cout<<"override() from derived is called" << endl;}
    virtual void virtual_override()
    {cout<<"virtual_override() from derived is called" << endl;}
};
```

CS112, semester 2, 2007

12

## Polymorphism example (cont)

```
void without_virtual_polymorphism(base * b){ b->override();}  
void virtual_polymorphism(base * b){ b->virtual_override();}  
  
int main()  
{  
    derived d1;  
    without_virtual_polymorphism(&d1); //no polymorphism  
    virtual_polymorphism(&d1); //polymorphism  
    return 0;  
}
```

**NOTE:** polymorphism works only though pointers or pass by reference, NOT though pass by value.

## Brief info about Abstract Classes

- Cannot be instantiated.
- You can't declare an object of that type.
- A class is made abstract by declaring one or more of its virtual function to be "pure". A pure virtual function is one with an initializer of = 0 in its declaration as in  
virtual float speak() = 0;
- Hence a derived class must override pure virtual function. If function implementation of pure virtual function in derived class is not provided than derived class would also become abstract class. (why?)

### Example

If Pet is abstract you could NOT declare:

```
Pet myPet;
```

# Lecture 6.1

## The Preprocessor and Linking

CS112, semester 2, 2007

1

Program is created in the editor and stored on disk.

Preprocessor program processes the code.

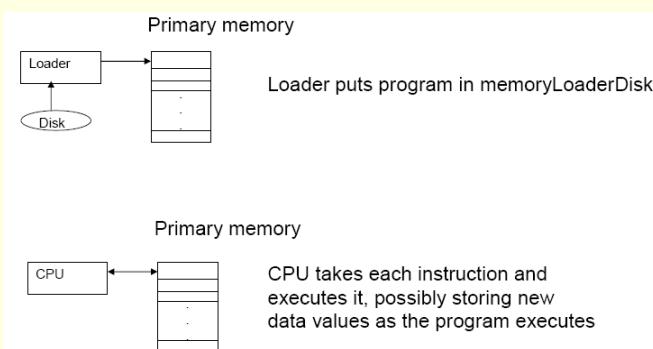
Compiler creates **object code** and stores it on disk.

Linker links the object code with the libraries, creates the executable file and stores it on disk.

CS112, semester 2, 2007

2

## A typical C++ environment (cont)



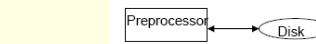
CS112, semester 2, 2007

3

## A typical C++ environment



Program is created in the editor and stored on disk.



Preprocessor program processes the code.



Compiler creates **object code** and stores it on disk.



Linker links the object code with the libraries, creates the executable file and stores it on disk.

## What is the preprocessor?

- Program that executes automatically **BEFORE** the compiler's translation phase begins.
- Obeys special commands called ***preprocessor directives***
- Indicate that certain manipulations need to be performed on the program before compilation.

CS112, semester 2, 2007

4

## What is the preprocessor? (cont)

- These manipulations usually consist of including other text files in the file to be compiled and performing various text replacements.
- The preprocessor is invoked by the compiler before the program is converted to machine language.

## Loading, the next phase

- Before a program can be executed, the program must first be placed in memory.
- Done by the loader, which takes the executable image from disk and transfers it to memory.
- Additional components from shared libraries that support the program are also loaded.

## What is linking?

- C++ programs typically contain references to functions defined elsewhere
- The object code produced by the C++ compiler typically contains “holes” due to these missing parts.
- A linker links the object code with the code for the missing functions to produce an executable image (with no missing pieces).

## Execution phase, the end

- The computer, under the control of its CPU, executes the program one instruction at a time.

# Preprocessor directives

## #include

- Causes a *copy* of a specified file to be included in place of the directive.
- #include <filename> (< and >) are used for a standard library header file. The preprocessor searches for the file in a implementation – dependent predefined location.

CS112, semester 2, 2007

9

## #include

- #include “filename”
  - Used for programmer defined header files.
- The preprocessor searches for the file first in the same directory as the file being compiled, then in the in the same location as the standard library header files.

CS112, semester 2, 2007

10

## What happens when you use #include?

```
***** main file *****/
#include "myExample.h"

int main( ){
    int x;
    myExample one;
    x=one.calculateX( );
}
****end of main file*****/

*****myExample.h*****
/**this is myExample declaration/public interface**/
class myExample{
    private:
        int a,b;
    public:
        int getA( );
        int getB( );
        void setA (int value);
        void setB (int value);
        int calculateX( );
};
*****end of myExample.h*****
```

CS112, semester 2, 2007

11

## After preprocessing

```
class myExample{
    private:
        int a,b;
    public:
        int getA( );
        int getB( );
        void setA (int value);
        void setB (int value);
        int calculateX( );
};

int main( ){
    int x;
    myExample one;
    x=one.calculateX( );
}
```

CS112, semester 2, 2007

12

## What happened to the class implementation?

- So far it has not been included in the code
- This code can be compiled, but it will have some holes that will need to be filled by the linker in the next phase.

CS112, semester 2, 2007

13

## MyExample implementation

(MyExample.cpp)

```
#include "myExample.h" ← Since this is a separate file, the header must be included here too. This file (myExample.cpp) will be compiled separately from the main file.

myExample::myExample() { a = 0; b = 0; }
int myExample::int getA() { return a; }
int myExample::int getB() { return b; }
void myExample::void setA (int value); { a = value; }
void myExample::void setB (int value); { b = value; }
int myExample:: calculateX(); { return (a + b); }
```

CS112, semester 2, 2007

15

## Class Implementations

- Class implementations should be in a **separate** file.
- Traditionally the file has the same name as the class with a **.cpp** extension
- Just as the header file is usually named with the same name of the class with a **.h** extension

CS112, semester 2, 2007

14

## More about implementations

- Implementations should NOT be included in header files
- their **.cpp** files should NEVER be included as headers
- Classes are meant to be reused.
- You don't want to give the public access to your code.
- You want to let other people USE your class, not modify it.

CS112, semester 2, 2007

16

## Even more about implementations

- This implementation file CAN be compiled on its own, without the need of a main (or driver) file.
- The compiler will generate an **object** file if the compilation was successful.
- This object file will have the same name as the .cpp file except that it will have a .o extension.

For example

compiling myExample.cpp will produce  
myExample.o

## Even more about implementations (cont)

- The linker the one that generates the executable file.
- If we only compile myExample.cpp there will be NO executable file.
- The executable file will be generated once the linker **links** the object code of the main file with the object code of myExample file

## Linking and Dev C++

- In DevC++ you usually don't have to worry about specifying that you want to link several files together
- You have to include them as part of a project.
- When compiling the project the linking will be done automatically.

## #define

- This preprocessor directive creates symbolic constants.
- Symbolic constants are normal *constants* represented by symbols instead of being declared with a data type.
- For example,  
`#define SIMB_CONST 99`  
is equivalent to,  
`const int SIMB_CONST = 99`

## #define (cont)

- Traditionally capital letters are used for constant identifiers.
- There's no = sign after SIMB\_CONST used with #define.

Otherwise every time you use SIMB\_CONST in your program, it would be replaced by =99 instead of just 99.

## Advantages of using #define over const

- You can check whether a symbolic constant has been defined or not (with the use of conditional compilation directives)

## Conditional Compilation

- Enables the programmer to control the execution of preprocessor directives and the compilation of program code.
- Each conditional preprocessor directive evaluates a constant integer expression that will determine if the code will be compiled.

## Conditional Compilation (cont)

- They work pretty much like a normal *if* statement.

For example

```
#ifndef X //or #if ! defined X  
#define X  
..... // definition of X  
#endif
```

## Conditional Compilation Example

```
#ifndef MYEXAMPLE_H
#define MYEXAMPLE_H

class myExample{
    private:
        int a,b;
    public:
        int getA( );
        int getB( );
        void setA (int value);
        void setB (int value);
        int calculateX( );
};

#endif
```

CS112, semester 2, 2007

25

## Final recommendation

- Good programming practice!

Use the name of the header file with the period replaced by an underscore when you are using conditional compilation preprocessor directives.

CS112, semester 2, 2007

26

## Lecture 6.2

### Dynamic Memory Allocation

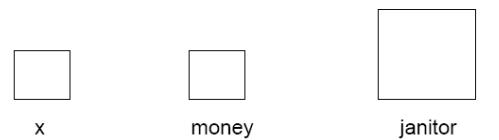
CS112, semester 2, 2007

1

### Statically allocated?

- Statically allocated objects are those we have been using through out this course.

```
int x;  
float money;  
Employee janitor;
```



CS112, semester 2, 2007

3

### What is memory allocation?

- To allocate memory is to reserve a space in memory for the variables we are declaring.
- In C++, space in memory for variables may be either statically or dynamically allocated.

CS112, semester 2, 2007

2

### Sizes in bytes of different variable types

<b>Size of int</b>	<b>4</b>
<b>Size of char</b>	<b>1</b>
<b>Size of short</b>	<b>2</b>
<b>Size of long</b>	<b>4</b>
<b>Size of float</b>	<b>4</b>
<b>Size of double</b>	<b>8</b>
<b>Size of ptr</b>	<b>4</b>
<b>Size of *ptr</b>	<b>4</b>

CS112, semester 2, 2007

4

## Statically allocated objects

- The compiler arranges the required space as it turns source code into a binary or executable program.
- Statically allocated objects that are of local scope are put into a memory space known as the stack.
- Statically allocated objects of global scope live in the global address space.

CS112, semester 2, 2007

5

## What happens if we don't know the size of an array?

- We could try to size the buffer or array to be large enough to hold the worst case (big enough to hold anything we should encounter)

CS112, semester 2, 2007

6

## What's wrong with this strategy?

- It consumes memory unnecessarily.
  - This is less of an issue than in the past when memory was more limited, but still impacts overall system performance.
- No matter how much memory is statically set aside for our object, we can never be sure it will be large enough.

CS112, semester 2, 2007

7

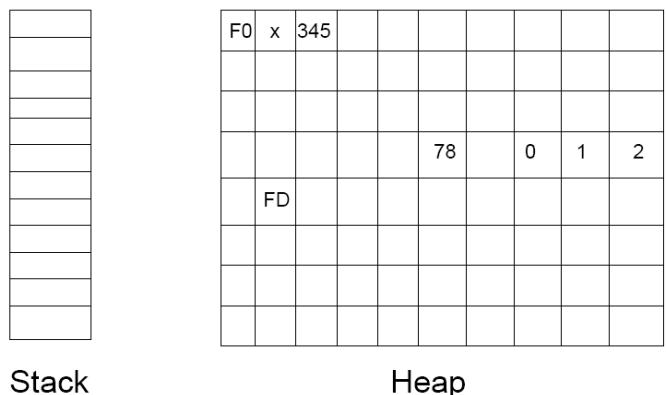
## Dynamically allocate an object?

- The memory for the object comes from a pool of memory known as the **heap** or **free store** instead of the stack.
- The memory is allocated when the program is run instead of when it is compiled.

CS112, semester 2, 2007

8

## Stack and Heap



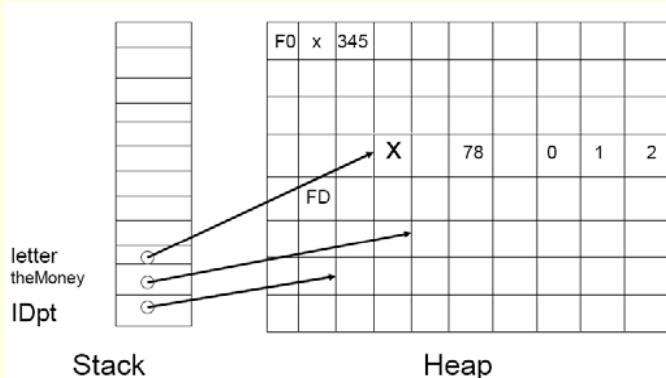
CS112, semester 2, 2007

9

CS112, semester 2, 2007

10

## Stack and Heap with dynamic memory allocation



CS112, semester 2, 2007

11

## What is the syntax for doing this?

- You will have to use two operators:
  - new
  - delete

### Example:

```
int *IDpt = new int;  
float *theMoney = new float;  
char *letter = new char;
```

## The “new” operator

- The "new" operator returns the address to the start of the allocated block of memory.
- This address must be stored in a pointer.
- "new" allocates a block of space of the appropriate size in the heap for the object.

CS112, semester 2, 2007

12

## The “new” operator (cont)

- int \*myPtr = new int;
- Notice that the reserved block of memory is anonymous; it has no identifier (name).
- dynamically allocated memory is accessed indirectly via a pointer.
- If there's no memory available “new” will fail.
- “new” will throw a "bad\_alloc" exception in this case.

CS112, semester 2, 2007

13

## Initializing a dynamically allocated Object (cont)

- There is another syntax we can follow when using the “new” operator:

```
int *IDpt = new int(5); //Allocates an int  
//object and initializes it to value 5  
char *letter = new char('J');
```

CS112, semester 2, 2007

15

## Initializing a dynamically allocated Object

- Dynamically allocated objects contain whatever random bits happen to be at their memory location.
- Therefore a value must be assigned **before** use.

```
int *IDpt = new int;  
*IDpt = 5;
```

CS112, semester 2, 2007

14

## “delete” operator

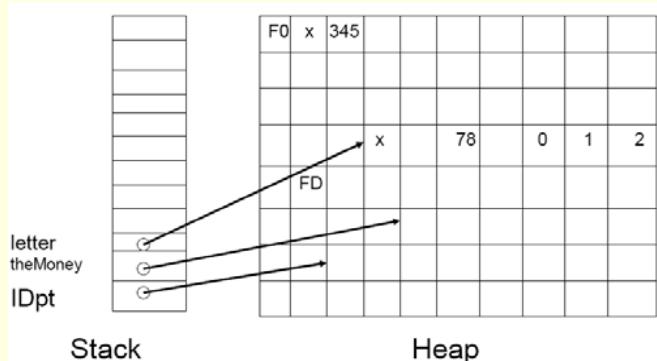
- Dynamically allocated objects must be explicitly deleted when no longer used by a program.
- “delete” releases the memory used by the object.
- That memory is then available for reuse.  

```
delete IDpt;  
delete theMoney;  
delete letter;
```

CS112, semester 2, 2007

16

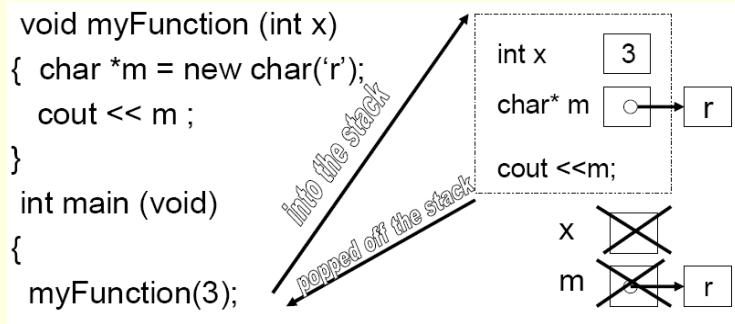
## Stack and Heap with dynamic memory allocation



CS112, semester 2, 2007

17

## What happens with statically allocated objects?

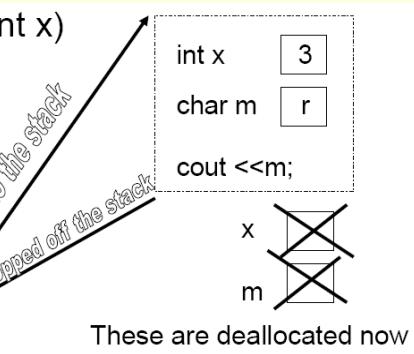


CS112, semester 2, 2007

19

## What happens with statically allocated objects?

```
void myFunction (int x)
{ char m = 'r';
cout << m ;
}
int main (void)
{
myFunction(3);
}
```



CS112, semester 2, 2007

18

## What is the problem?

- The dynamically allocated object still exists!
- And now we no longer have a pointer to it so we cannot release it!
- This is known as a **memory leak**

CS112, semester 2, 2007

20

## Function with static memory allocation

```
void Function ( int amount) {  
    float money = 78.56;  
    char myArray[3]={0}  
}  
  
int main (void) {  
    Function ( 3567);  
    return 0;  
}
```

CS112, semester 2, 2007

21

## Stack and Heap when function is compiled

myArray 2	0
myArray 1	0
myArray 0	0
money	78.56
amount	3567

Stack

F0	x	345				
			78	0	1	2
FD						

Heap

CS112, semester 2, 2007

22

## Stack and Heap when function goes out of scope

CS112, semester 2, 2007

23

## Function with dynamic memory allocation

```
void myFunction (int x){  
    char *m = new char('r');  
    cout << m ;  
}  
  
int main (void)  
{  
    myFunction(3);  
}
```

CS112, semester 2, 2007

24

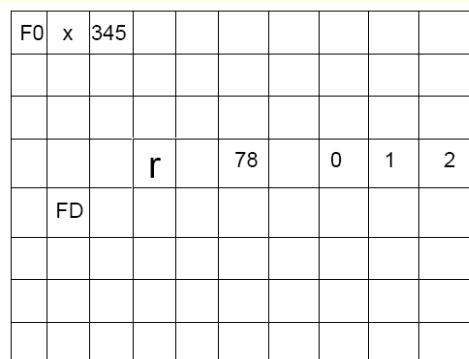
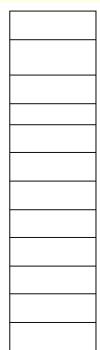
# Lecture 6.3

## Dynamic Memory Allocation

CS112, semester 2, 2007

1

## Stack and Heap when function goes out of scope



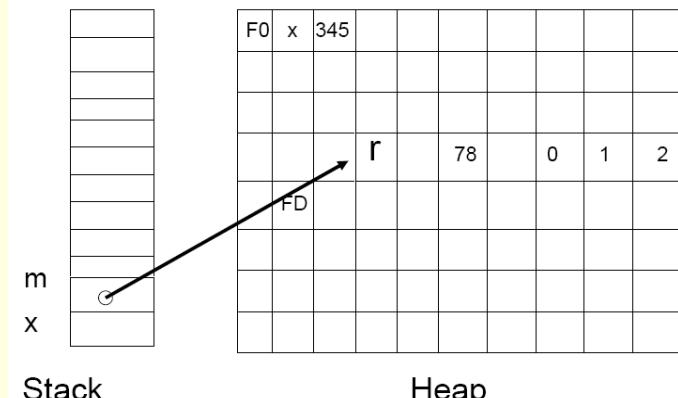
Stack

Heap

CS112, semester 2, 2007

3

## Stack and Heap with dynamic memory allocation



CS112, semester 2, 2007

2

## What happens if you call the function 15 times?

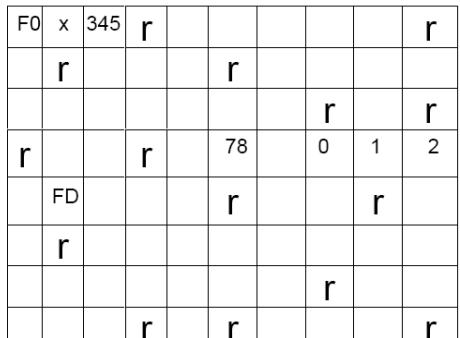
```
void myFunction (int x){  
    char *m = new char('r');  
    cout << m ;  
}
```

```
int main (void){  
    for (int i = 0; i < 15; i++)  
        myFunction(3);  
}
```

CS112, semester 2, 2007

4

## Stack and Heap when function goes out of scope **without** delete



Stack

Heap

CS112, semester 2, 2007

5

## What's the problem with this?

- As the program continued to operate, more and more memory will be lost from the heap (free store).
- If the program runs long enough, eventually no memory will be available, and the program will no longer operate.

CS112, semester 2, 2007

6

## What's the problem with this? (cont)

- Even if we don't run out of memory, the reduced pool of available memory affects system performance.
- The moral of all this: **Be sure to delete.**
- Every new should be paired with a delete in your code to avoid memory leaks.

CS112, semester 2, 2007

7

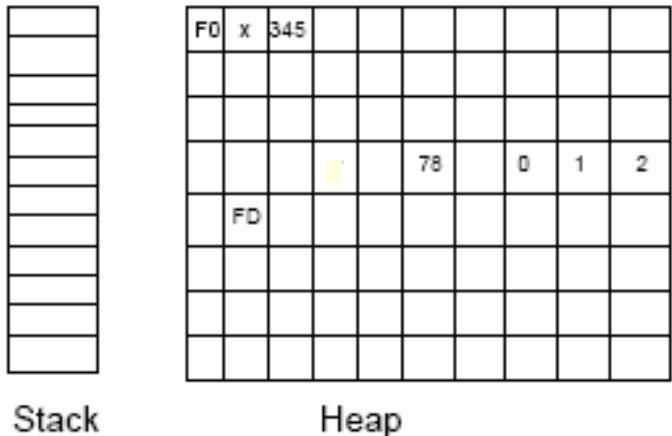
## If you “free” the memory with delete

```
void myFunction (int x){  
    char *m = new char('r');  
    cout << m ;  
    delete m;  
}  
  
int main (void){  
    for (int i = 0; i < 15; i++)  
        myFunction(3);  
}
```

CS112, semester 2, 2007

8

## Stack and Heap when function goes out of scope **with** delete



CS112, semester 2, 2007

9

CS112, semester 2, 2007

10

## Dynamically Allocating Arrays

- Arrays of built-in and user-defined data types may be dynamically allocated.
- User-defined data types include classes.

```
int *pt = new int [ 1024 ]; //allocates an array  
//of 1024 ints
```

```
double *myBills = new double [ 10000 ];  
//allocates array of 10000 doubles
```

## Important! Do not get confused!

- Note the difference between:

```
int *pt = new int [ 1024 ]; //allocates an array  
//of 1024 ints
```

```
int *pt = new int ( 1024 ); //allocates a single  
//int with value 1024
```

CS112, semester 2, 2007

11

## Initializing a dynamically allocated array

```
int *buff = new int [ 1024 ];  
for ( i = 0; i < 1024; i++ )  
{  
    *buff = 52; //Assigns 52 to each element;  
    buff++;  
}
```

CS112, semester 2, 2007

12

## Initializing a dynamically allocated array (cont)

- Or

```
int *buff = new int [ 1024 ];  
for ( i = 0; i < 1024; i++ )  
{  
    buff [ i ] = 52; //Assigns 52 to each  
    //element;  
}
```

CS112, semester 2, 2007

13

## So what is different from what we are used to?

- Use new to allocate memory and always assign it to a pointer of the same type of our allocated memory
- ```
int *buff = new int [ 1024 ];
```
- The pointer can be used just as we are used to ( with an index like an array or with the \* to indicate "value")
- ```
buff [ 1 ] = 43;  
or  
buff++;  
*buff = 43;
```
- Use delete when we are done.
- ```
delete [ ] buff;
```

CS112, semester 2, 2007

15

## How to use “delete” with a dynamically allocated array

- The syntax of the “delete” operator for dynamically allocated arrays is slightly different from what we saw for single objects.

```
delete [ ] pt;  
delete [ ] myBills;
```

- The square brackets after the delete tell the compiler to delete a dynamic array rather than a single object.

CS112, semester 2, 2007

14

## Dangling Pointers

- Take a look at this snippet of code.

```
1 int *myPointer;  
2 myPointer = new int(10);  
3 cout << *myPointer << endl;  
4 delete myPointer;  
5 *myPointer = 5;  
6 cout << *myPointer << endl;
```

CS112, semester 2, 2007

16

## What's wrong with this?

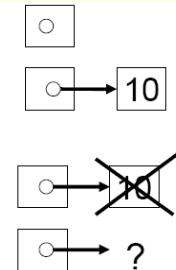
- myPointer is dangling!
- We've released the memory of the object whose address myPointer holds and then continued to use it.

CS112, semester 2, 2007

17

## What's happening?

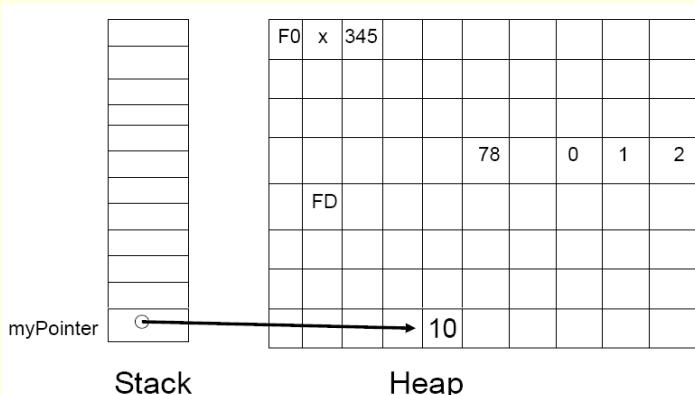
- 1 int \*myPointer;
- 2 myPointer = new int(10)
- 3 cout << \*myPointer;
- 4 delete myPointer;
- 5 \*myPointer = 5;
- 6 cout << \*myPointer;



CS112, semester 2, 2007

18

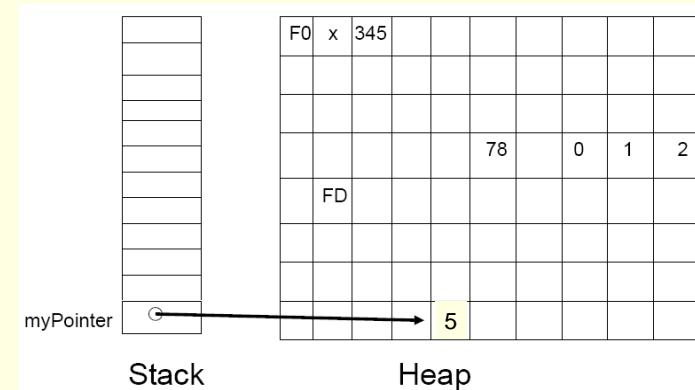
## Stack and Heap before dangling pointer



CS112, semester 2, 2007

19

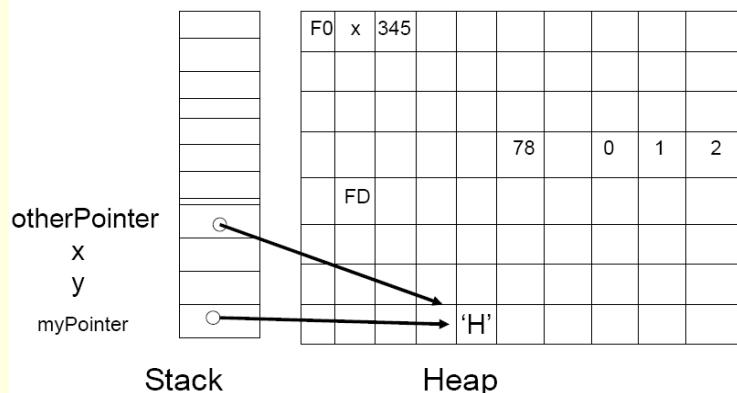
## Stack and Heap with dangling pointer



CS112, semester 2, 2007

20

## Stack and Heap with mem location used by other pointer



CS112, semester 2, 2007

21

## What is the problem with this?

- Although the program may run, this section of memory may be used by another dynamic object allocated after the delete.
- The values in that object will be corrupted by the continued use of `myPointer`.
- This is a very subtle programming bug and is very difficult to isolate.

CS112, semester 2, 2007

22

## Can you prevent it?

- To avoid this bug, always set a pointer to `NULL`, after the `delete` is called.
- Subsequent attempts to use the pointer will result in a run-time exception.
- This will immediately allow the bug to be identified and fixed.

CS112, semester 2, 2007

23

## Corrected code

```
int *myPointer;  
myPointer = new int(10);  
cout << *myPointer << endl;  
delete myPointer;  
myPointer = NULL;  
*myPointer = 5; //This statement will cause a  
//run-time exception, now.  
cout << *myPointer << endl;
```

CS112, semester 2, 2007

24

# Lecture 7.1

## Recursion

CS112, semester 2, 2007

1

## Factorial Example

### Problem:

- Mathematically, the factorial of an integer is defined as:-
- $0! = 1$
- $n! = n * (n-1)!$ , if  $n > 0$

$$1! = 1$$

$$2! = 2 * 1$$

$$3! = 3 * 2 * 1$$

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

CS112, semester 2, 2007

3

## What is Recursion?

- Recursion is defined as a function calling itself.
- It is in some ways similar to a loop because it repeats the same code
- It requires passing in the looping variable from one function call to another.
- Problem solving by reducing the problem to a smaller versions of itself.

CS112, semester 2, 2007

2

## Factorial Example

### So

$$\begin{aligned}4! &= 4 * (4-1)! \\3! &= 3 * (3-1)! \\2! &= 2 * (2-1)! \\1! &= 1 * (1-1)! \\0! &= 1\end{aligned}$$

- This gives us a recursive definition (smaller versions of itself)

CS112, semester 2, 2007

4

## Recursive definition

- Recursive functions must have
  - **Base case** ( $0! = 1$ )
  - **General case** ( $3!, 2!..$ etc i.e.  $n! = (n-1)! \cdot n$ )
  - Recursive function must have one or more base case and one general case.
  - Base case stops the recursion

CS112, semester 2, 2007

5

## Recursive definition

- Recursive algorithm
  - Algorithm that finds the solution to a given problem by reducing the problem to a smaller version of itself.
- Recursive function
  - A function that calls itself

CS112, semester 2, 2007

6

## Recursive function

```
int factorial ( int n ) {  
    if (n > 1)  
        return n * factorial ( n - 1 ) ;  
    else  
        return 1;  
}
```

CS112, semester 2, 2007

7

## How Recursion Looks Like

```
void recurse( )  
{  
    recurse ( ); //Function calls itself  
}  
  
int main ( )  
{  
    recurse ( ); //Sets off the recursion  
    return 0; //Rather pitiful, it will never be reached  
}
```

CS112, semester 2, 2007

8

## Recursion Example

```
void recurse ( int count ) //The count variable is initialized
{ // by each function call
    cout<<count ;
    recurse(count+1); // It is not necessary to increment count
} // each function's variables
// are separate (so each count will be
// initialized one greater)

int main ()
{
    recurse(1); //First function call, so it starts at one
    return 0;
}
```

CS112, semester 2, 2007

9

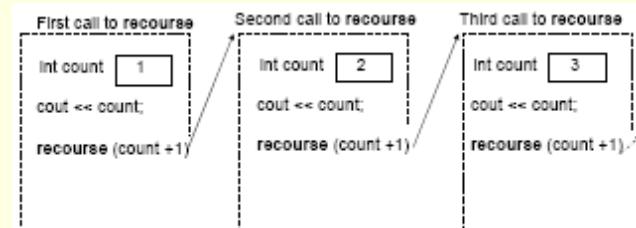
## Base Case of the function

- The condition where the function will not call itself.
- It is an if-statement that checks some variable for a condition (such as a number being less than zero, or greater than some other number)
- If that condition is true, it will not allow the function to call itself again.

CS112, semester 2, 2007

11

## What's happening?



- Will go on until the computer stack gets full,
- Or until we control it with the help of a variable

CS112, semester 2, 2007

10

## Example of base case

```
void doll ( int size )
{
    if (size == 0 ) // No doll can be smaller than 0
        return; // Return does not have to return
                  // something, it can be used to exit a
                  // function
    cout<<size<<endl;
    doll (size-1); // Decrements the size variable so the next
                  // doll will be smaller.
}

int main ( )
{
    doll (10); //Starts off with a large doll
    return 0; //Finally, it will be used }
```

CS112, semester 2, 2007

12

## Important Info!

- If our base case is not properly set up, it is possible to have a base case that is always true (or always false).
- Once a function has called itself, it will be ready to go to the next line after the call.
- It can still perform operations.

CS112, semester 2, 2007

13

## Solution

```
void printnum ( int begin )
{
    cout<<begin;
    if ( begin < 9 )
        printnum ( begin + 1 );
        // The base case is when begin is greater
        // than 9, it will not recurse after the if
        // statement
    cout<<begin; } // Outputs the second begin, after the
        // program has gone through and
        // output the numbers from begin to 9.
```

CS112, semester 2, 2007

15

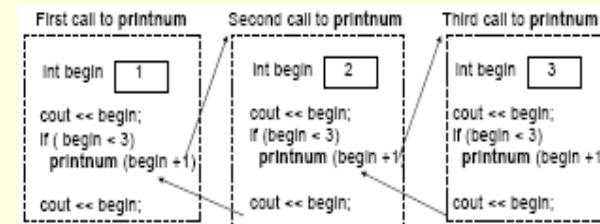
## Performing operations **after** the call

- Problem:
- Write a function to print out the numbers 123456789987654321.
- How can you use recursion to write a function to do this?

CS112, semester 2, 2007

14

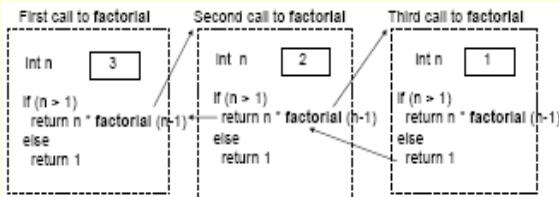
## What's happening?



CS112, semester 2, 2007

16

## What's happening?

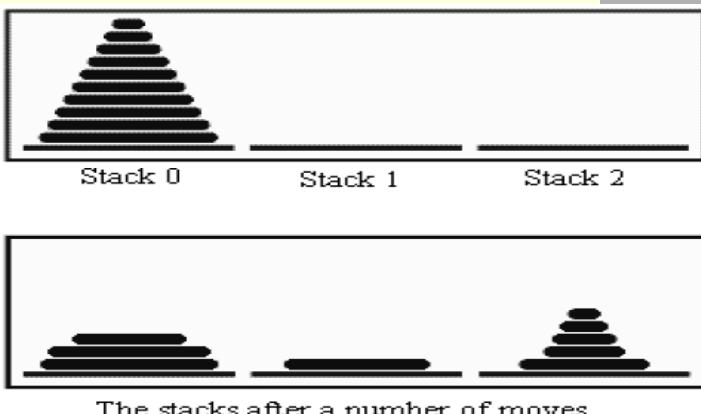


- Third call will return 1
- Second call will return  $2 * 1 = 2$
- First call will return  $3 * 2 = 6$

CS112, semester 2, 2007

17

## Towers of Hanoi representation



CS112, semester 2, 2007

19

## Towers of Hanoi

- This problem involves a stack of various sized disks, piled up on a base in order of decreasing size.
- The object is to move the stack from one base to another, subject to two rules:
  - Only one disk can be moved at a time
  - No disk can ever be placed on top of a smaller disk.
- There is a third base that can be used as a "spare".

CS112, semester 2, 2007

18

## Towers of Hanoi Solution

- The base case is when there is only one disk to be moved.
- The solution in this case is trivial: Just move the disk in one step.

CS112, semester 2, 2007

20

## Towers of Hanoi Solution

- The base case is when there is only one disk to be moved.
- The solution in this case is trivial: Just move the disk in one step.

CS112, semester 2, 2007

21

## Towers of Hanoi Implementation

```
void TowersOfHanoi ( int disks, int from,
                     int to, int spare )  
{  
    if (disks == 1) // base case  
        cout << "Move a disk from stack number "  
             << from << " to stack number " << to  
             << endl;  
}
```

CS112, semester 2, 2007

22

## Towers of Hanoi Implementation (cont)

```
else {  
    // Move all but one disk to the spare  
    // stack, then move the bottom disk, then  
    // put all the other disks on top of it.  
    TowersOfHanoi ( disks- 1, from , spare , to);  
    cout << "Move a disk from stack number "  
         << from << " to stack number " << to << endl;  
    TowersOfHanoi(disks- 1, spare, to, from);  
}
```

CS112, semester 2, 2007

23

# Lecture 8.1

## Sorting and Searching

CS112, semester 2, 2007

1

## Sorting with Bubble Sort

- Compares two values next to each other and exchanges them if necessary to put them in the right order.
- Many variations on the order in which the pairs are examined.

CS112, semester 2, 2007

2

## Algorithm Analysis

- The BIG – O - Notation
- Some algorithms may take very little computer time to compute while others may take a considerable amount of time
- Example

```
c= 0;  
sum = 0;  
cin>>num;  
while (num !=-1)  
{sum+=num;c++;cin>>num;}  
average=sum/count;  
cout<<"Average is"<<average;
```

CS112, semester 2, 2007

3

## The BIG – O - Notation

- The algorithm
  - has 3 operations before the while loop
  - 4 operations within the while loop
  - 2 operations after the while loop
- In total (if loop executes 10 times)
  - $10*4 + 3+2$
- In total (if loop executes 100 times)
  - $100*4 + 3+2$
- We can generalize it to  $4n+5$  ( $n$  = loops)
- For large values of  $n$ , the term  $4n$  becomes the dominating term

CS112, semester 2, 2007

4

## The BIG – O - Notation

- Growth Rate of various functions
- Function grows as n (problem size grows)

| n  | $\log_2 n$ | $n \log_2 n$ | $n^2$ | $2^n$  |
|----|------------|--------------|-------|--------|
| 1  | 0          | 0            | 1     | 2      |
| 2  | 1          | 2            | 2     | 4      |
| 4  | 2          | 8            | 16    | 16     |
| 8  | 3          | 24           | 64    | 256    |
| 16 | 4          | 64           | 256   | 65,536 |

CS112, semester 2, 2007

5

## Growth Rate

| n  | $f(n)=n$ | $f(n)=\log_2 n$ | $f(n)=n \log_2 n$ | $f(n)=n^2$ | $f(n)=2^n$ |
|----|----------|-----------------|-------------------|------------|------------|
| 10 | 0.01μs   | 0.03μs          | 0.033μs           | 0.1μs      | 1μs        |

- Time for  $f(n)$  instructions on a computer that executes 1 billion instructions per second
- Definition – we say that  $f(n)$  is **Big-O** of  $g(n)$  written  $f(n) = O(g(n))$  if there exists positive constants  $c$  and  $n_o$  such that  $f(n) \leq cg(n)$  for all  $n \geq n_o$

CS112, semester 2, 2007

6

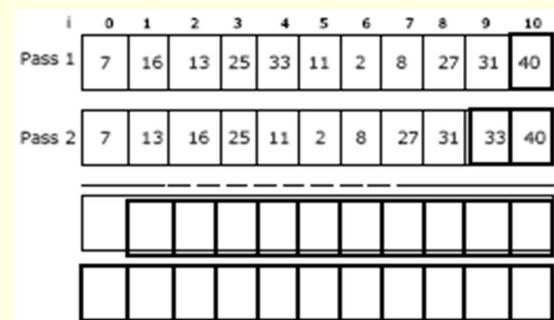
## Bubble Sort facts

- Efficient aspect of bubble sorts,
  - can quit early if the elements are almost sorted.
- Bubble sorts are  $O(N^2)$  on the average,
- Can have an  $O(N)$  best case.

CS112, semester 2, 2007

7

## $O(N^2)$



CS112, semester 2, 2007

8

## BS-fixed number of passes

```
void bubbleSort1 ( int x [ ] , int n )
{
    for (int pass=1; pass<n; pass++){ //count
        //how many times
        //This next loop becomes
        //shorter and shorter
        for ( int i=0; i < n - pass; i++ ) {
            if ( x [ i ] > x [ i+1 ] ) {
                //exchange elements
                int temp = x [ i ];
                x [ i ] = x [ i+1 ];
                x [ i+1 ] = temp;}}
    }
}
```

CS112, semester 2, 2007

9

## BS-fixed number of passes facts

- Fixed number of passes =length of the array - 1
- Each inner loop is one shorter than the previous one
- Disadvantage:
  - Always makes  $n-1$  passes over the array,
  - Can't stop early if the array is already sorted.

CS112, semester 2, 2007

10

## BS—stop when no exchanges

```
void bubbleSort2 ( int x [ ] , int n ) {
    bool exchanges;
    int temp;
    do {
        exchanges = false; // assume no exchanges
        for ( int i=0; i < n-1; i++ ) {
            if ( x [ i ] > x [ i+1 ] ) {
                temp = x[ i ];
                x [ i ] = x[ i+1 ];
                x [ i+1 ] = temp;
                exchanges = true;}} // after
                           //exchange,
                           // must look again
    }while (exchanges);
```

CS112, semester 2, 2007

11

## BS—stop when no exchanges facts

- Continues making passes over the array as long as there were any exchanges.
- If the array already sorted, sort will stop after only one pass.
- Disadvantage:
  - Doesn't shorten the range each time by 1 as it could.

CS112, semester 2, 2007

12

## BS-stop when no exchanges, shorter each time

```
void bubbleSort3 ( int x [ ] , int n ) {  
    bool exchanges;  
    int temp;  
    do {  
        n--; //make loop smaller each time  
        exchanges = false; // assume this is last pass over array  
        for ( int i=0; i < n; i++ ) {  
            if (x [ i ] > x [ i+1 ] ) {  
                temp = x[ i ];  
                x [ i ]= x [ i+1 ];  
                x [ i+1 ]= temp;  
                exchanges = true; // after exchange must look again  
            } } }  
    while (exchanges); }
```

CS112, semester 2, 2007

13

## Linear Search

- Straightforward loop comparing every element in the array with the *key* (the item we are looking for).
- As soon as an equal value is found, it returns
- If loop finishes without finding a match, a -1 is returned.
- Good solution for small arrays

CS112, semester 2, 2007

14

## Linear Search code

```
int linearSearch ( int a [ ] , int first, int last, int key)  
{  
    for ( int i = first; i <= last; i++ ) {  
        if ( key == a [ i ] )  
            return i;  
    }  
    return -1; // failed to find key  
}
```

CS112, semester 2, 2007

15

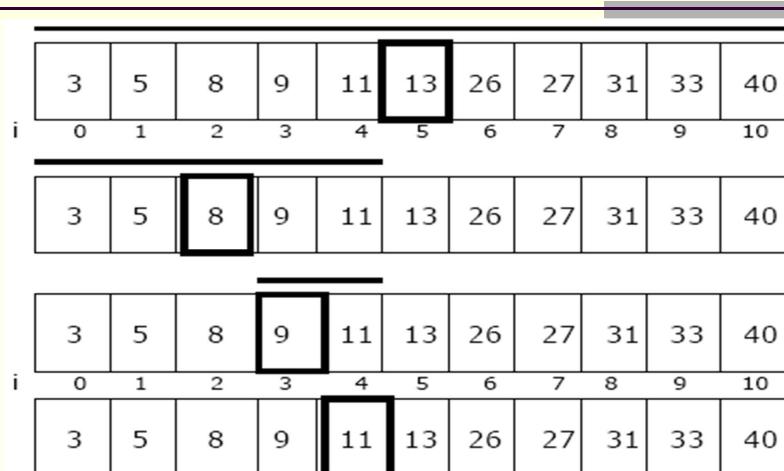
## Binary Search

- Fastest way to search a sorted array.
- Look at the element in the middle.
  - If the key is equal to that, search is finished.
  - If the key is less than the middle element, do a binary search on first half.
  - If it's greater, do a binary search of the second half.

CS112, semester 2, 2007

16

## Binary Search (cont)



CS112, semester 2, 2007

17

## Binary search code

```
int binarySearch ( int sortedArray [ ] , int first, int last, int key ) {  
    while ( first <= last ) {  
        int mid =( first + last ) / 2; //compute mid point.  
        if ( key > sortedArray [ mid ] )  
            first = mid + 1; // repeat search in top half.  
        else if ( key < sortedArray [ mid ] )  
            last = mid - 1; // repeat search in bottom half.  
        else return mid; // found it. return position  
    }  
    return - ( first + 1 ); // failed to find key  
}
```

CS112, semester 2, 2007

18

## Lecture 8.2

### Recursive sorting and searching

CS112, semester 2, 2007

1

### Recursive Binary Search

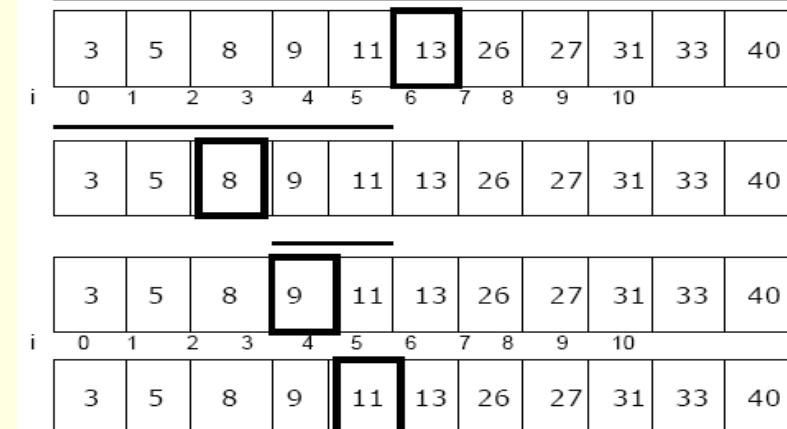
```
int binarySearch (int array [ ] , int first, int last, int key)
{
    int middle;

    // base case
    if ( first > last)
        return - 1; // key was not found
```

CS112, semester 2, 2007

3

### Binary Search



CS112, semester 2, 2007

2

### Recursive Binary Search (cont)

```
else {
    middle = ( first + last ) / 2;
    if ( key == array [ middle ] )
        return middle;
    else if ( key < array [ middle ] )
        return binarySearch ( array , first, middle - 1, key );
    else
        return binarySearch ( array, middle + 1, last, key );
}
```

CS112, semester 2, 2007

4

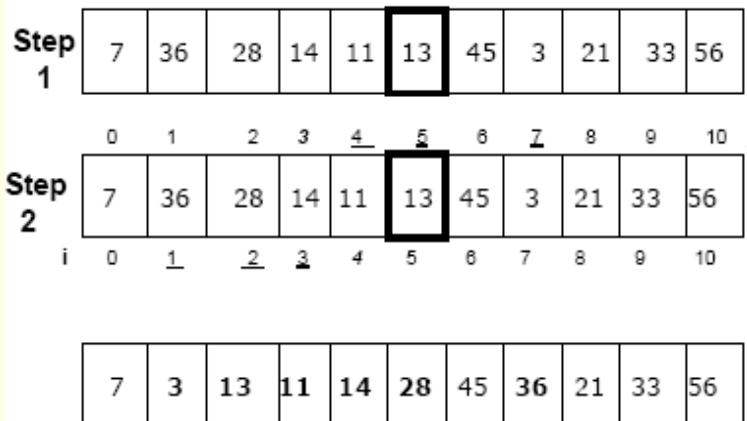
# Quicksort

- One of the fastest and simplest sorting algorithms. It works recursively by a divide-and-conquer strategy.
- Divide the array of items to be sorted into two partitions and then call the quicksort procedure recursively to sort the two partitions,
- ie we *divide* the problem into two smaller ones and *conquer* by solving the smaller ones.

CS112, semester 2, 2007

5

## How Quicksort works



CS112, semester 2, 2007

7

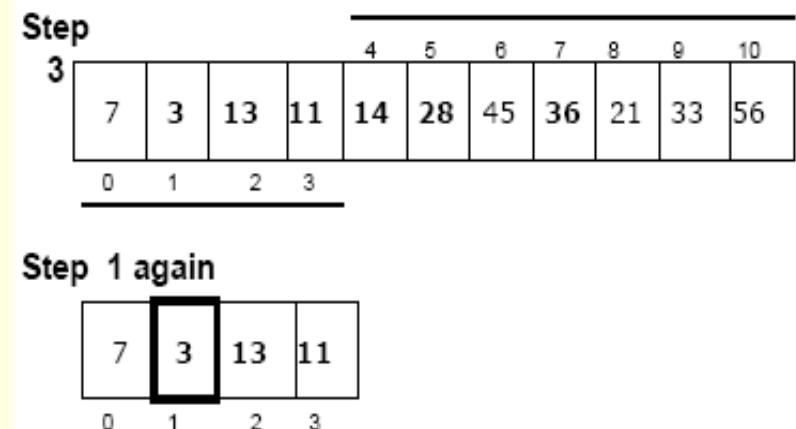
# Quicksort Algorithm

1. pick one element in the array, which will be the *pivot*.
2. make one pass through the array, called a *partition step*, re-arranging the entries so that:
  - entries smaller than the pivot are to the left of the pivot.
  - entries larger than the pivot are to its right.
  - the pivot is in its proper place.
3. recursively apply quicksort to the part of the array that is to the left of the pivot, and to the part on its right.

CS112, semester 2, 2007

6

## How Quicksort works



CS112, semester 2, 2007

8

## Quicksort Implementation

```
void quicksort ( int array [ ], int lo, int hi ) {  
    // lo is the lower index,  
    // hi is the upper index  
    // of the region of array a that is to be  
    // sorted  
    int i = lo, j = hi, temp;  
    //step 1, x will be the pivot  
    int x = array [ ( lo + hi ) / 2 ];
```

CS112, semester 2, 2007

9

## Quicksort Implementation (cont 2)

```
// partition //step  
do {  
    while (array [ i ] < x ) //check until we find an  
    //element bigger than the pivot in the  
    i++; // lower part of the array.  
    while (array [ j ] > x ) // check until we find an  
    //element smaller than the pivot in the  
    j- - ; // higher part of the array.
```

CS112, semester 2, 2007

10

## Quicksort Implementation (cont 3)

```
if (i <= j)  
{  
    temp = array [ i ]; //swap  
    array [ i ] = array [ j ];  
    array [ j ] = temp;  
    i++;  
    j- - ;  
}  
} while (i <= j);
```

CS112, semester 2, 2007

11

## Quicksort Implementation (cont 4)

```
// recursion // step 3  
if ( lo < j ) quicksort ( array , lo, j );  
if ( i < hi ) quicksort ( array , i, hi );  
}
```

CS112, semester 2, 2007

12

## Quicksort Analysis

- The *best-case* behavior of the Quicksort algorithm occurs when in each recursion step the partitioning produces two parts of equal length.
- In order to sort  $n$  elements, in this case the running time is in  $O(n \log(n))$ .
- This is because the recursion depth is  $\log(n)$  and on each level there are  $n$  elements to be treated.

## Quicksort Analysis (cont)

- The worst case occurs when in each recursion step an unbalanced partitioning is produced.
- When one part consists of only one element and the other part consists of the rest of the elements .
- Then the recursion depth is  $n-1$  and Quicksort runs in time (  $n^2$  ).

# Lecture 9.1

## Linked List

CS112, semester 2, 2007

1

## Considerations

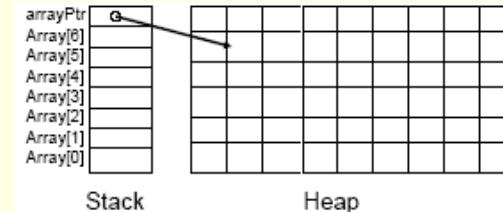
- What happens if we needed a smaller array than the one declared?
- What happens if we needed a bigger array than the one declared?
- What if there is not a continuous block of memory big enough to allocate the array?
- We need a dynamic structure!

CS112, semester 2, 2007

3

## What we know about arrays

- Arrays need a continuous block of memory
- `int Array[7]={0};`
- `int *arrayPtr = new int[7];`



CS112, semester 2, 2007

2

## Dynamic data structures

- This kind of structures can grow and shrink during execution.
- Linked Lists
- Stacks
- Queues
- Trees

CS112, semester 2, 2007

4

## What are linked lists?

- Way to store data with structures so that the user can automatically create a new place to store data whenever necessary.
- The programmer writes a struct or class definition that contains variables holding information about something, and then has a pointer to a struct of its type.

CS112, semester 2, 2007

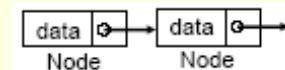
5

## What are linked lists? (cont)

- Each of these individual struct or classes in the list is known as a node and each node contains whatever data you are storing along with a pointer (a link) to another node

Example:

```
struct Node{  
    int data;  
    Node * next;  
};
```



CS112, semester 2, 2007

6

## How Linked Lists work?

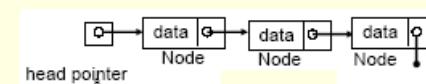
- Think of it like a train.
- The programmer always stores the first node of the list. This would be the engine of the train (head).
- The pointer is the connector between cars of the train.
- Every time the train adds a car, it uses the connectors to add a new car.
- The programmer uses the keyword `new` to create a pointer to a new struct or class.

CS112, semester 2, 2007

7

## Traversing the list

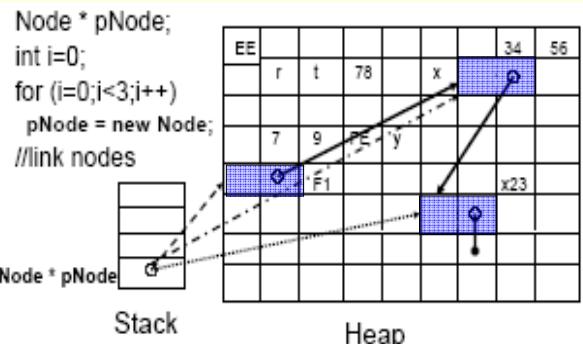
- By locating the node referenced by the head pointer and then doing the same with the pointer in that new node and so on, you can traverse the entire list.
- At the end, there is nothing for the pointer to point to, so it does not point to anything. It should be set to "NULL" to prevent it from accidentally pointing to a totally arbitrary and random location in memory (which is very bad).



CS112, semester 2, 2007

8

## How's the memory for a linked list allocated?



CS112, semester 2, 2007

9

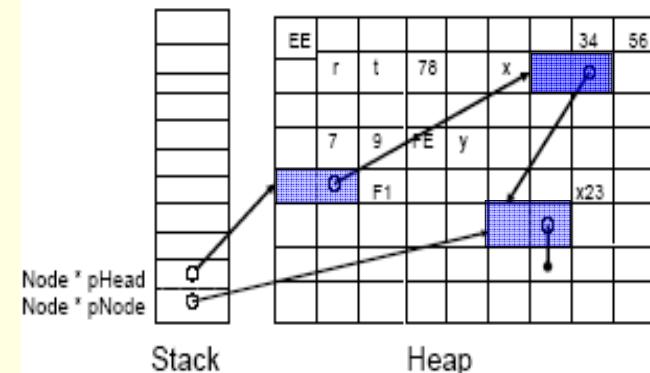
## Linked Lists: Some Properties

- Linked list basic operations:
  - Search the list to determine whether a particular item is in the list
  - Insert an item in the list
  - Delete an item from the list

CS112, semester 2, 2007

11

## Keeping track of the head



CS112, semester 2, 2007

10

## Inserting Items into the list

- New nodes can be inserted into the list at any place:
  - before first node
  - after the final node
  - between two existing nodes
  - into an empty list

CS112, semester 2, 2007

12

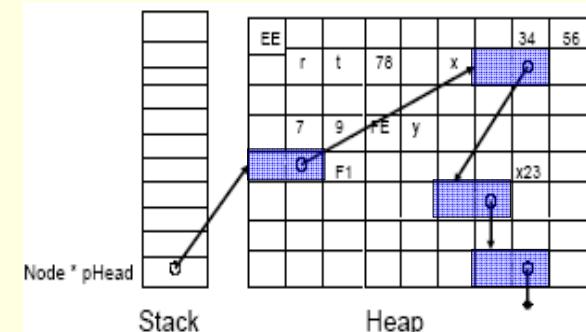
## Removing items from the list

- Nodes may be removed from any place along the chain
  - the first node in the chain
  - final node in the chain
  - between two existing nodes

CS112, semester 2, 2007

13

## List before remove

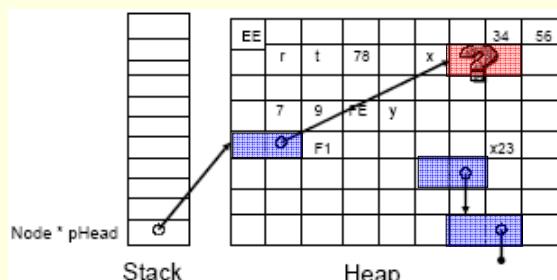


CS112, semester 2, 2007

14

## Removing example

- Can you remove a node without any consequences?

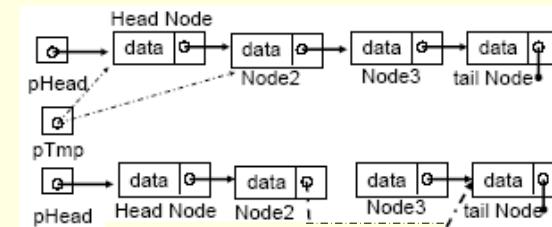


CS112, semester 2, 2007

15

## Removing example:

- To remove Node3 we need to know what node is before Node3 to change the “next” pointer.

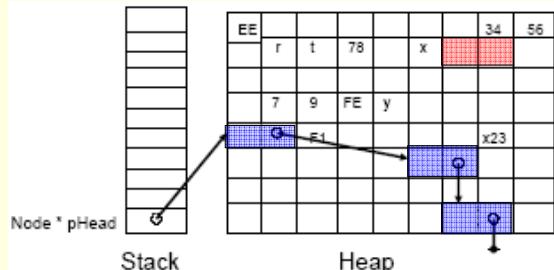


CS112, semester 2, 2007

16

## Removing example

- Keeping track of the previous node would make it easy to find the next node.



CS112, semester 2, 2007

17

## Doubly linked list

- Makes it easy to keep track of **previous** and **next** nodes.

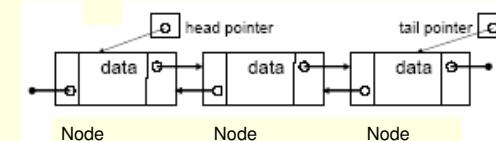
```
struct Node {
```

```
    int Data;
```

```
    Node *prev;
```

```
    Node *next;
```

```
};
```



CS112, semester 2, 2007

18

## Doubly linked lists (cont)

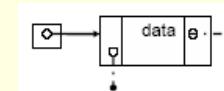
- These lists still need to code for special cases of inserting at the front, in the middle or after the last node in the chain.
- Similar consideration is required for removing a node from the list.

CS112, semester 2, 2007

19

## Inserting into an empty list.

- Create new node
- Make head pointer point to new node
- Make "prev" of new node point to null.
- Make "next" of new node point to null.

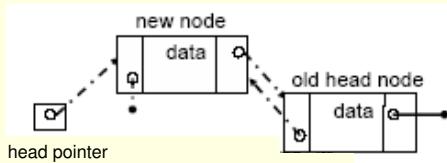


CS112, semester 2, 2007

20

## Insert before the first node

1. Create new node
2. Make head node “prev” point to new node
3. Make new node “next” point to head node
4. Make head pointer point to new node
5. Make new node “prev” point to null

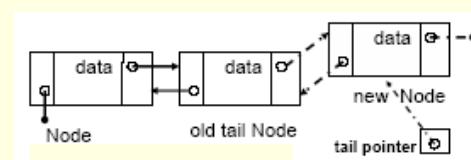


CS112, semester 2, 2007

21

## Append at the end of the list

1. Create new node
2. Make tail node “next” point to new node
3. Make new node “prev” point to tail node
4. Make tail pointer point to new node
5. Make new node “next” point to null.

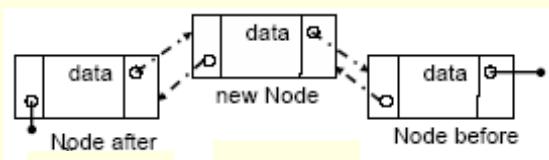


CS112, semester 2, 2007

22

## Insert between two existing nodes

1. Create new node
2. Make new node “next” point to “next” of “node after”
3. Make new node “prev” point to “node after”
4. Make “prev” of “node before” point to new node
5. Make “next” of “node after” point to new node

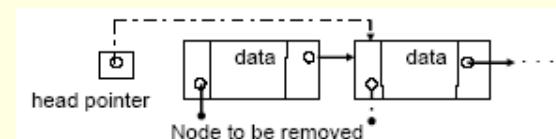


CS112, semester 2, 2007

23

## Removing the first node in the list

1. Make head pointer point to “next” of node to be removed
2. Make new head node “prev” point to null
3. Remove the node

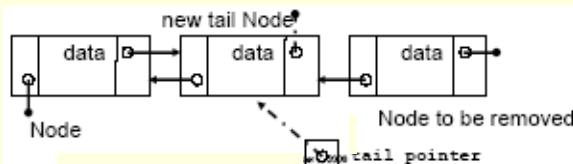


CS112, semester 2, 2007

24

## Removing the last node in the list

- Make tail pointer point to the “prev” of node to be removed
- Make “next” of new tail node point to null.
- Remove the node

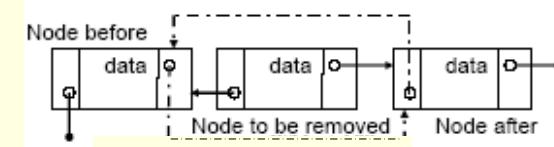


CS112, semester 2, 2007

25

## Removing a node between two existing nodes

- Make “next” of “node before” point to “next” of node to be removed
- Make “prev” of “node after” point to “prev” of node to be removed
- Remove the node



CS112, semester 2, 2007

26

## Advantages of a Linked List

- You can quickly insert and delete items in a linked list.
- Inserting and deleting items in an array requires you to either make room for new items or fill the "hole" left by deleting an item.
- With a linked list, you simply rearrange those pointers that are affected by the change.

CS112, semester 2, 2007

27

## Disadvantages of a Linked List

- They are quite difficult to sort.
- You cannot immediately locate, say, the hundredth element in a linked list the way you can in an array. Instead, you must traverse the list until you've found the hundredth element.

CS112, semester 2, 2007

28

# Lecture 9.2

## Linked List Implementation with Structure

CS112, semester 2, 2007

1

## Linked List Implementation

```
struct NODE {  
    NODE *pNext;  
    NODE *pPrev;  
    int nData;  
};  
  
//declare head and tail of list  
//originally the list is empty
```

```
NODE *pHead = NULL;  
NODE *pTail = NULL;
```

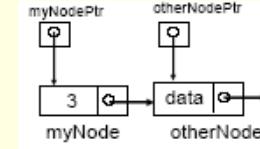
CS112, semester 2, 2007

3

## Node structure

```
struct Node{  
    int Data;  
    Node * pNode;  
};  
  
Node myNode, otherNode;  
myNode.Data = 3;  
OtherNode.Data = 5;  
myNode.pNode = &otherNode;
```

```
Node * myNodePtr;  
myNodePtr->Data = 3;  
myNodePtr->pNode = &otherNode;  
  
Node *otherNodePtr;  
myNodePtr->pNode = otherNodePtr;
```



CS112, semester 2, 2007

2

## Linked List implementation

```
void AppendNode ( NODE *pNode);  
  
void InsertNode ( NODE *pNode, NODE *pAfter);  
  
void RemoveNode ( NODE *pNode);  
  
void DeleteAllNodes ( );
```

CS112, semester 2, 2007

4

## LL implementation (cont 2)

```
int main()
{
    NODE *pNode; //declare the pointer for
    // the dynamically allocated memory
```

CS112, semester 2, 2007

5

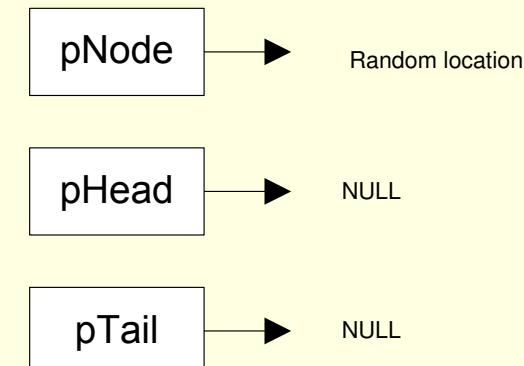
## LL Implementation

```
for (int i = 0; i < 100; i++) {
    pNode = new NODE; //allocate
    // memory for each node and make
    // pointer point to the
    // dynamically allocated memory
    pNode->nData = i; //put some data in the node
    AppendNode(pNode); //add node to list
}
```

CS112, semester 2, 2007

7

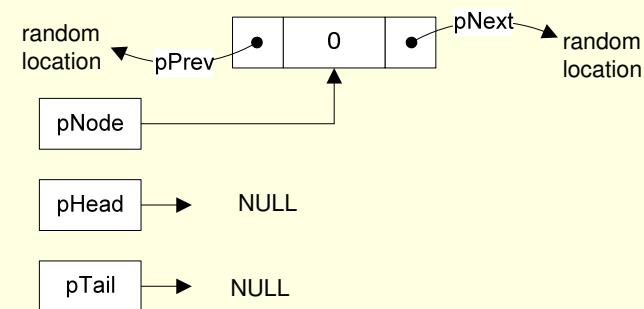
## LL Implementation (so far)



CS112, semester 2, 2007

6

Allocating new node, making **pNode** point to new node and assigning **nData** of new node to 0.



CS112, semester 2, 2007

8

## Appending pNode

```
void AppendNode ( NODE *pNode )
{
    if (pHead == NULL) { //if list is empty
        pHead = pNode; //make head point to pNode
        pNode->pPrev = NULL;
    }
    else {
        pTail->pNext = pNode; //make tail point to pNode
        pNode->pPrev = pTail;
    }

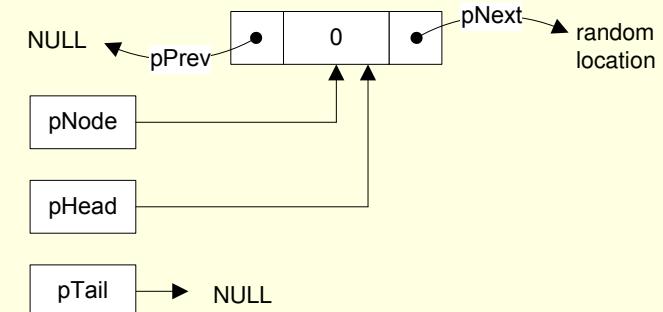
    pTail = pNode; //tail is now pNode
    pNode->pNext = NULL; //pNode next now points to NULL
}
```

CS112, semester 2, 2007

9

## Appending a new node

```
if ( pHead == NULL ) {
    pHead = pNode;
    pNode->pPrev = NULL;
}
```

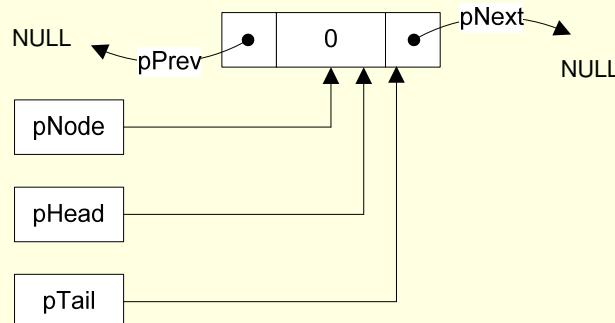


CS112, semester 2, 2007

10

## Appending a new node

```
pTail = pNode;
pNode->pNext = NULL;
```

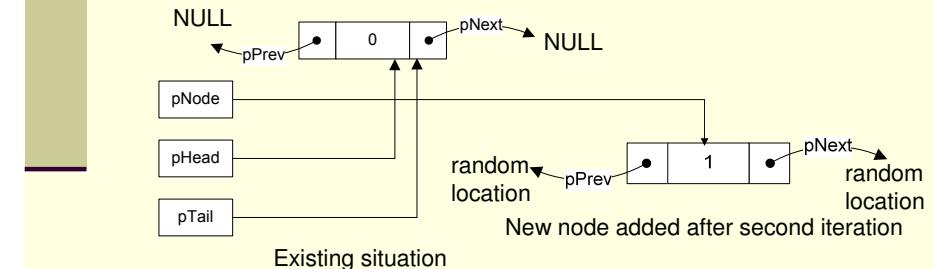


CS112, semester 2, 2007

11

## Second iteration of for loop

- Allocating another new node, making pNode point to new node and assigning nData of new node to 1.

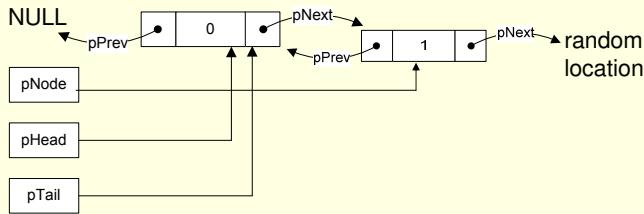


CS112, semester 2, 2007

12

## Appending a new node

```
pTail->pNext = pNode; //newly added node  
pNode->pPrev = pTail;
```

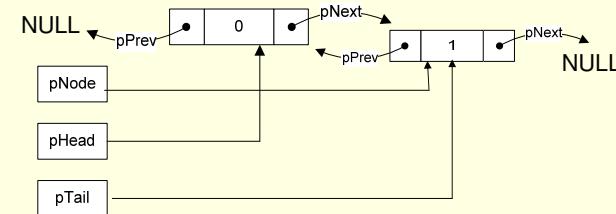


CS112, semester 2, 2007

13

## Appending a new node (cont)

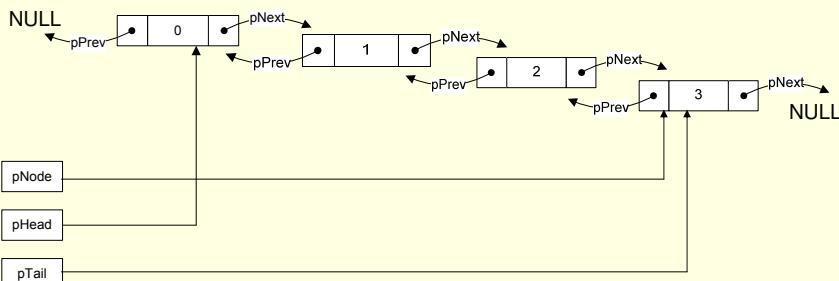
```
pTail = pNode; //newly added node  
pNode->next = NULL;
```



CS112, semester 2, 2007

14

## After 4 iterations



CS112, semester 2, 2007

15

## Inserting a node

```
void InsertNode(NODE *pNode, NODE *pAfter)  
{  
    pNode->pNext = pAfter->pNext;  
    pNode->pPrev = pAfter;  
  
    if (pAfter->pNext != NULL)  
        pAfter->pNext->pPrev = pNode;  
    else  
        pTail = pNode;  
  
    pAfter->pNext = pNode;  
}
```

CS112, semester 2, 2007

16

## Removing Nodes

```
void RemoveNode(NODE *pNode) {  
    if (pNode->pPrev == NULL)  
        pHead = pNode->pNext;  
    else  
        pNode->pPrev->pNext = pNode->pNext;  
  
    if (pNode->pNext == NULL)  
        pTail = pNode->pPrev;  
    else  
        pNode->pNext->pPrev = pNode->pPrev;  
  
    delete pNode;  
}
```

CS112, semester 2, 2007

17

## Delete all Nodes

```
void DeleteAllNodes()  
{  
    while (pHead != NULL) //keep on  
        //removing until the  
        //head points to NULL  
    RemoveNode(pHead);  
}
```

CS112, semester 2, 2007

18

## Traversing the list

```
for (pNode = pHead; pNode != NULL; pNode = pNode->pNext)  
    cout << pNode->nData << endl;
```

CS112, semester 2, 2007

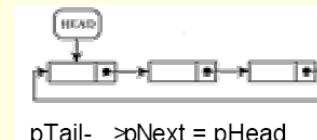
19

# Lecture 9.3

## Linked List (continuation)

CS112, semester 2, 2007

1



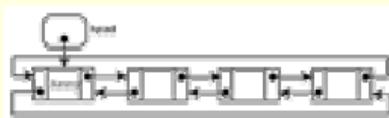
pTail->pNext = pHead

- The last node links back to the first.
- Traversing the list stops when you arrive back at the head of the list.

CS112, semester 2, 2007

2

## Circular doubly linked lists



- Reduces all special cases to inserting or removing between nodes

CS112, semester 2, 2007

3

## Implementing Linked Lists with classes

- Convert your struct into a class.
- Create a class for your list, and the functions of the previous implementation into methods for the list class.

CS112, semester 2, 2007

4

## The Node class

```
class Node{  
    friend class List; // gives List class access to private  
                      // members of Node class  
  
private:  
    int nData;  
    Node* pNext;  
    Node* pPrev;  
  
public:  
    Node ( int data ) { nData = data;  
                      pNext = NULL;  
                      pPrev = NULL; }  
    int getData( ) { return nData; }  
};
```

CS112, semester 2, 2007

5

## The List class

```
class List{  
private:  
    Node* pHead;  
    Node* pTail;  
    Node* createNode ( int data ) ;  
  
public:  
    List ();  
    ~List();  
    void appendNode ( int value );  
    void insertNode ( int value, Node *pAfter );  
    void removeNode (Node *pNode);  
    bool isEmpty ( );  
    void printList ( );  
};
```

CS112, semester 2, 2007

6

## List Constructor and Destructor

```
List::List () {  
    pHead=NULL;  
    pTail=NULL;  
}  
List::~List () {  
    while ( !isEmpty ( ) ) // keep on removing until the  
                          // head points to NULL  
        removeNode( pHead );  
    cout << "List deleted\n";  
}
```

CS112, semester 2, 2007

7

## Extra methods

```
Node* List::createNode ( int data ) {  
    //allocate memory for new node and  
    //intialize value to data  
    Node* pNode = new Node ( data ) ;  
    return pNode;  
}  
  
bool List::isEmpty ( ) {  
    return pHead == NULL;  
}
```

CS112, semester 2, 2007

8

## Changes to Append method

```
void List::appendNode ( int value )
{
    Node* pNode = createNode ( value ) ;
    if ( isEmpty ( ) ) { // if list is empty
        pHead = pNode; // make head point to pNode
        pNode->pPrev = NULL;
        ...
        ...
    }
}
```

CS112, semester 2, 2007

9

## The main

```
int main()
{
    List sampleList;
    // Add items to linked list
    for (int i = 0; i < 100; i++)
        sampleList.appendNode ( i ); //add node to list
                                    //print the list
    sampleList.printList();
    system ("pause");
    return 0;
}
```

CS112, semester 2, 2007

10

# Lecture 9.4

## Templates

CS112, semester 2, 2007

1

CS112, semester 2, 2007

2

## What is template

- The programmer writes a single function template definition. Based on the argument types provided in calls to this function, the compiler automatically generates separated object code functions to handle each type of call appropriately.

CS112, semester 2, 2007

3

## Templates

- Templates can be used with classes as well as with functions.
- Function templates are used when data types of arguments or return types are required to be generic.

## Problem without template

- Suppose you have been asked to create linked list of integers, floats and strings.

|                                                                   |                                                                     |                                                                      |
|-------------------------------------------------------------------|---------------------------------------------------------------------|----------------------------------------------------------------------|
| Struct iNode{<br>iNode * pNext;<br>iNode * pPrev;<br>int nData;}; | Struct fNode{<br>fNode * pNext;<br>fNode * pPrev;<br>float nData;}; | Struct sNode{<br>sNode * pNext;<br>sNode * pPrev;<br>string nData;}; |
|-------------------------------------------------------------------|---------------------------------------------------------------------|----------------------------------------------------------------------|

- You will have to rewrite entire code for node 3 times.

CS112, semester 2, 2007

4

## Solution

- Write just one templated node struct.

```
template<class T>
Struct Node{
    Node * pNext;
    Node * pPrev;
    T nData;
};
```

CS112, semester 2, 2007

5

## Implementation of AppendNode

```
template <class T>
void AppendNode(NODE<T> * pNode)
{
    if (pHead == NULL) { //if list is empty
        pHead = pNode; //make head point to pNode
        pNode->pPrev = NULL;
    }
    else {
        pTail->pNext = pNode; //make tail point to pNode
        pNode->pPrev = pTail;
    }
    pTail = pNode; //tail is now pNode
    pNode->pNext = NULL; //pNode next now points to NULL
}
```

CS112, semester 2, 2007

7

## Function implementation with template

```
template <class T>
void AppendNode(NODE<T> * pNode);

template <class T>
void InsertNode(NODE<T> * pNode, NODE<T> * pAfter)

template <class T>
void InsertNodeAt(NODE<T> * pNode, NODE<T> * pAfter);

template <class T>
void RemoveNode(NODE<T> * pNode);

template <class T>
void DeleteAllNodes();
```

CS112, semester 2, 2007

6

## Calling templated node struct from main

```
int main( )
{
    NODE <int> * iNode;
    NODE <string> * sNode;
    NODE <float> * fNode;

    //your code...
}
```

CS112, semester 2, 2007

8

## Dev C++ project

---

- We need a trick to include templated files in dev c++ project.
- There are two ways for this:
- Put the implementation of methods in the header file.
- Or
- Call implementation file together with header file. Eg.

```
#include "node.h"  
#include "node.cpp"
```

# Lecture 9.5

## Pointer to pointer (\*\*)

CS112, semester 2, 2007

1

CS112, semester 2, 2007

2

## Pointer to a variable

Example:

```
int b; // b is a variable  
b = 5;  
int * a; //pointer a, initially pointing to nowhere  
a = &b; // pointer a is pointing to b which is an  
//integer variable
```

CS112, semester 2, 2007

3

## What you already know about pointer

- pointers are just one memory block that store the address of another object/variable.

## Pointer to pointer

- It is not necessary that pointer just points to variable or object. Pointer can also point to another pointer.

Example:

```
int c = 5;  
int d = 10;
```

```
int *b = &c; //b is pointing to c
```

CS112, semester 2, 2007

4

## Example (cont)

```
//int **a; //pointer to pointer  
//a = &b; //a is pointing to b which is a pointer.
```

//or you can also use the following syntax

```
int**a = &b; //a is pointing to b which is a pointer itself  
//hence, *a is an address of b  
// and **a is a value of b i.e. where b  
//is pointing
```

## Example (cont)

- Changing location of \*b reflects on \*\*a, similarly changing the location of \*\*a would change the location of \*b;

```
*a = &c;  
cout <<"value of *b after **a is changed: "  
<<*b<<endl;
```

## Example (cont)

```
cout<<"intial value of **a: "<<**a<<endl;
```

```
b = &d; //pointer is changed. now b is pointing  
//to d
```

```
cout <<"value of **a after b* is changed: "  
<<**a<<endl;
```

- conclusion: \*\*a always points to the location where b\* points to.

# Lecture 10.1

## Stack data structure

CS112, semester 2, 2007

1

## Dynamic memory allocation in C

- **malloc** returns a void pointer to the allocated buffer.
- This pointer must be *cast* (converted) into the proper type to access the data to be stored in the buffer.
- On failure, malloc returns a null pointer. The return from malloc should be tested for error as shown below.

```
char *cpt;  
...  
if ( ( cpt = (char *) malloc ( 25 ) ) == NULL )  
{  
    printf ( "Error on malloc\n" );  
}
```

CS112, semester 2, 2007

3

## Dynamic memory allocation in C

- Just like in C++ we allocate memory dynamically with the **new** operator, in C there is a function called **malloc** used to allocate memory.

- Its prototype is:

```
void* malloc ( size_t nbytes );
```

CS112, semester 2, 2007

2

## Remember sizeof ( )?

```
cout<< "The size of an int is " << sizeof ( int )<< endl;  
cout<< "The size of a float is " << sizeof ( float ) << endl;  
cout<< "The size of a char is " << sizeof ( char ) << endl;  
cout<< "The size of a double is "<< sizeof ( double ) << endl;
```

```
The size of an int is 4  
The size of a float is 4  
The size of a char is 1  
The size of a double is 8
```

CS112, semester 2, 2007

4

## Dynamic memory allocation in C

- To free the memory, just like using **delete** in C++, in C you call the **free** function

```
void free ( void *pt );
```

- It is not necessary to cast the pointer argument back to void. The compiler handles this conversion.

- Example of use:

```
free cpt;
```

## What is a Stack?

- A stack is another way to store data.
- It is generally implemented with only two principle operations.
  - push // put in
  - pop // take out

## Another Example

C++ code

```
Node * pNode = new Node;
```

```
pNode = pHead ...
```

```
.
```

```
.
```

```
.
```

```
delete pNode
```

C code

```
if (
```

```
Node * pNode =  
( Node* ) malloc ( sizeof (Node) ) )  
== NULL )
```

```
printf ("Error on malloc");
```

```
pNode = pHead ...
```

```
.
```

```
.
```

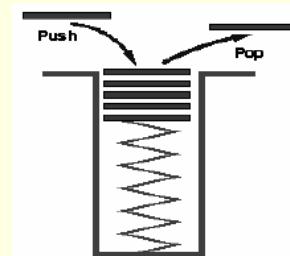
```
.
```

```
free (pNode);
```

## How does a stack work?

- A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.

- Stacks form Last-In-First-Out (LIFO) queues and have many applications.



## LIFO?

- LIFO = Last In, First Out.  
The last element in is the first element out.
- With lists we can implement different orders:  
FIFO = First In, First Out.  
FILO = First In, Last Out
- Stacks have many applications from the parsing of algebraic expressions to keeping track of variables and return address values for function calls

CS112, semester 2, 2007

9

## About dynamic allocation

- In most operating systems, allocation and deallocation of memory is a relatively expensive operation, there is a penalty for the flexibility of linked list implementations.

CS112, semester 2, 2007

11

## About the stack implementation

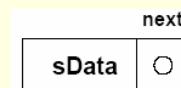
- A linked list implementation of a stack is possible  
(adding and deleting from the head of a linked list produces exactly the LIFO semantics of a stack)
- However the most common applications for stacks have a space restraint so that using an array implementation is a natural and efficient one

CS112, semester 2, 2007

10

## Stack Implementation with linked lists

```
typedef struct stack_node{  
    int sData;  
    stack_node* next;  
}sNode;
```



- `sNode* top = NULL;`

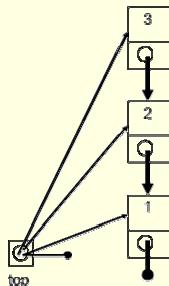


CS112, semester 2, 2007

12

## Push

```
void push ( int data ) {
    sNode* nNode = ( sNode * ) malloc ( sizeof ( sNode ) );
    nNode->sData = data;
    if ( top == NULL )
    {
        top = nNode;
        nNode->next = NULL;
    }
    else
    {
        nNode->next = top;
        top = nNode;
    }
}
```

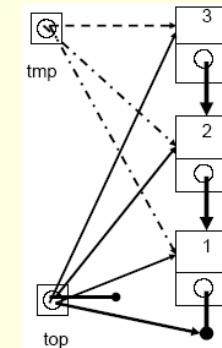


CS112, semester 2, 2007

13

## Pop

```
bool pop ( int &data )
{
    if ( top == NULL)
        return false;
    else
    {
        sNode* tmp = top;
        data = top->sData;
        top = top->next;
        free (tmp);
        return true;
    }
}
```



CS112, semester 2, 2007

14

## Printing the Stack

```
void printStack ( ) {
    sNode* tmp = top;
    while ( tmp != NULL)
    {
        cout << "\t" << tmp- > sData << endl;
        tmp = tmp- > next;
    }
}
```

CS112, semester 2, 2007

15

## The output

- The output will **always** be in inverse order as the input.
- If we push the numbers:
  - 1 2 3 4 5 6 7
- In that order, we will pop:
  - 7 6 5 4 3 2 1

CS112, semester 2, 2007

16

## Calling the methods

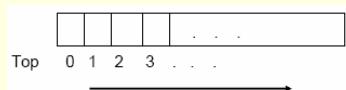
```
int main ()  
{  
    int dData;  
    for ( int i = 0; i <= 5; i++ )  
        push ( i );  
    cout << "Printing Stack \n";  
    printStack ( ) ;  
    for ( int i = 5; i >= 0; i-- ){  
        if ( pop ( dData ) ){  
            cout <<"popping :"<<dData<<endl; .....  
    }
```

CS112, semester 2, 2007

17

## Method implementation

```
template <class Type> bool Stack<Type>::push(Type val)  
{  
    // Check stack is not full, increment index, then store  
    if ( !isFull ( ) ){  
        sPtr [ ++top ] = val;  
        return true;  
    }  
    return false;  
}
```



CS112, semester 2, 2007

19

## Stack implementation with arrays and template classes

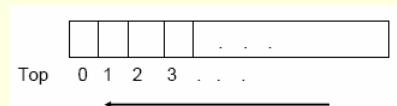
```
template <class Type> class Stack {  
private:  
    int size;  
    Type* sPtr;  
    int top;  
public:  
    Stack (int s) { size = s;  
        top = -1;  
        sPtr = new Type [ size ]; //allocate the array for the stack.  
    }  
    bool push( Type val );  
    bool pop ( Type &data );  
    bool isEmpty( ) { return top == -1; }  
    bool isFull ( ) { return top == ( size -1 ); }  
    int getIndex () { return top; }  
};
```

CS112, semester 2, 2007

18

## Method implementation (cont)

```
template <class Type> bool Stack<Type>::pop(Type &data )  
{  
    // Retrieve, then decrement index  
    if ( !isEmpty ( ) )  
    {  
        data= sPtr [ top-- ];  
        return true;  
    }  
    return false;  
}
```



CS112, semester 2, 2007

20

## Calling the methods

```
int main () {  
    int size;  
    cout << "Please enter a size for your stack " << endl;  
    cin >> size;  
    Stack <int> stack(5);  
    int num;  
    int val;  
    for (int i = 0; i < size; i++) {  
        cout << "Enter num: ";  
        cin >> num;  
        stack.push ( num );  
    }  
}
```

CS112, semester 2, 2007

21

## Calling the methods (cont)

```
while ( stack.pop ( val ) ) {  
    cout << "popped " << val << endl;  
    cout << "index " << stack.getIndex() << endl;  
}  
  
system("PAUSE");  
return 0;  
}
```

CS112, semester 2, 2007

22

CS112, semester 2, 2007

23

## Lecture 10.2

### Queue structure

CS112, semester 2, 2007

1

### Sample Queue Applications

- Processor queue:
  - Entries of the user are placed on a queue and executed in the order they were entered.
- Keyboard queue:
  - Characters are placed on a queue and displayed in the order they were entered.
- Printer queue
- Router queue:
  - Sends packets in the order they arrive

CS112, semester 2, 2007

3

## Queues

- A queue is much like a check-out line queue where the first element comes out first and the last element comes out last.
- Nodes are removed only from the front of the queue.
- Nodes are inserted only at the tail of the queue.

CS112, semester 2, 2007

2

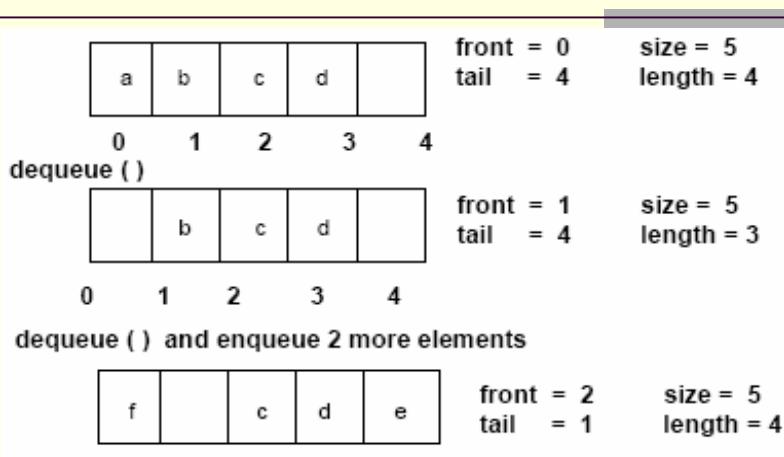
### Dequeue facts

- When using an array to implement the queue, every time we remove an element, there will be a space left free in the array.
- We will want to use those spaces when we reach the end of the array.

CS112, semester 2, 2007

4

## Queue implementation



CS112, semester 2, 2007

5

CS112, semester 2, 2007

6

## The enqueue() function

```
bool enqueue (int val)
{
    // Check queue is not full, store then increment tail
    if ( length != size )
    {
        qPtr [ tail ] = val;
        tail = ( tail+1 ) % size;
        length++;
        return true;
    }
    return false;
}
```

## The dequeue( ) function

```
bool dequeue ( int & data )
{
    //Retrieve then increment front
    if (length != 0)
    {
        data= qPtr [ front ];
        front=( front+1 )%size;
        length--;
        return true;
    }
    return false;
}
```

CS112, semester 2, 2007

7

## The main ( )

```
#include <iostream.h>
#include <stdlib.h>

bool enqueue (int val);
bool dequeue (int &data);
//declare these as global variables so they are visible to the
// functions
int* qPtr;
int front = 0, tail = 0;
int size = 0, length = 0;
```

CS112, semester 2, 2007

8

## The main () (cont)

```
int main()
{
    int size = 100;
    qPtr = new int [size];
    int num;
    int val;
```

CS112, semester 2, 2007

9

## The main() (cont 2)

```
for (int i = 0; i < size ; i++)
{
    cout << "Enter num: ";
    cin >>num;
    if ( enqueue ( num ) )
        ...
}
```

CS112, semester 2, 2007

10

## The main ( ) (cont 3)

```
while( dequeue(val) ) {
    cout << "dequeued :" << val << endl;
    ...
    system ("pause");
    return 0;
}
```

CS112, semester 2, 2007

11

# Queues

This lesson is borrowed from the following:

Reference

CS 367 – Introduction to Data Structures

*<http://pages.cs.wisc.edu/~mattmcc/cs367/notes/Queues.ppt>*

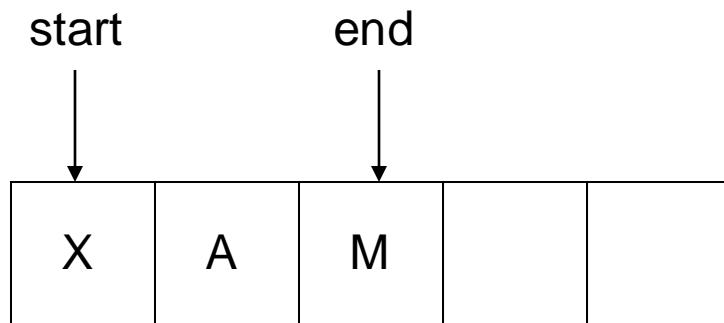
# Queue

- A queue is a data structure that stores data in such a way that the last piece of data stored, is the last one retrieved
  - also called First-In, First-Out (FIFO)
- Only access to the stack is the first and last element
  - consider people standing in line
    - they get service in the order that they arrive

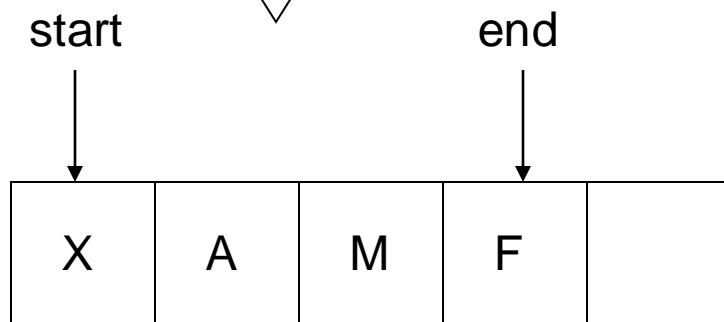
# Queues

- *Enqueue*
  - operation to place a new item at the tail of the queue
- *Dequeue*
  - operation to remove the next item from the head of the queue

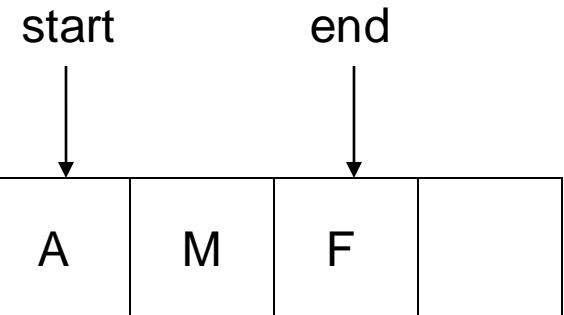
# Queue



enqueue(F)



item = dequeue()  
item = X



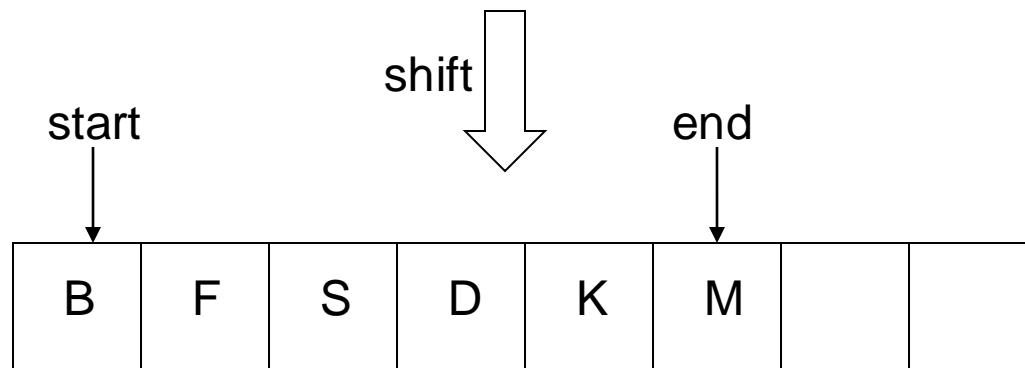
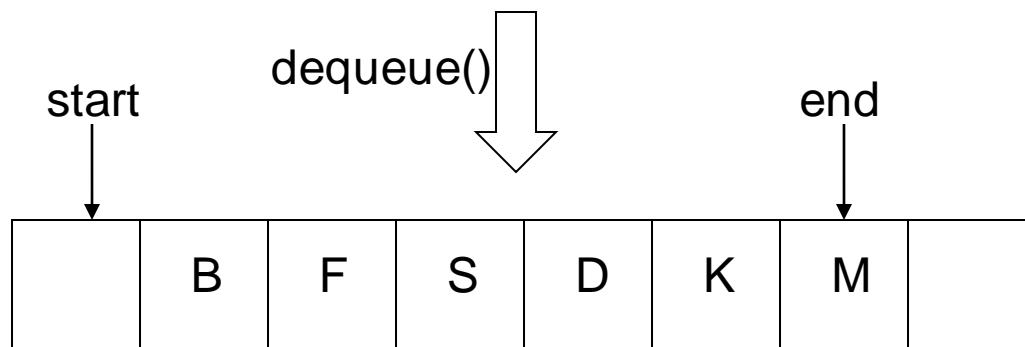
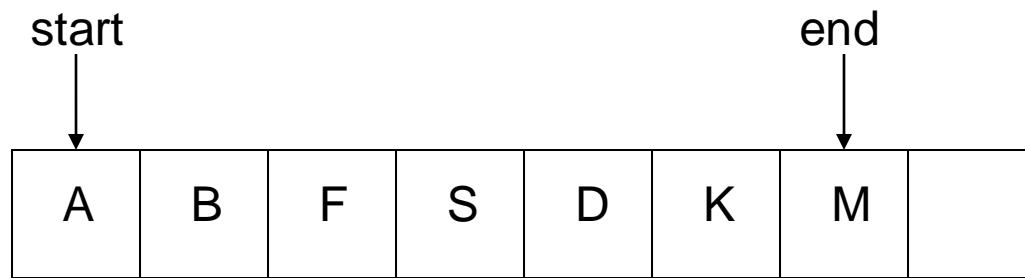
# Implementing a Queue

- At least two common methods for implementing a queue
  - array
  - linked list
- Which method to use depends on the application
  - advantages? disadvantages?

# Regular Linear Array

- In a standard linear array
  - 0 is the first element
  - `array.length - 1` is the last element
- All objects removed would come from element 0
- All objects added would go one past the last currently occupied slot
- To implement this, when an object is removed, all elements must be shifted down by one spot

# Regular Linear Array



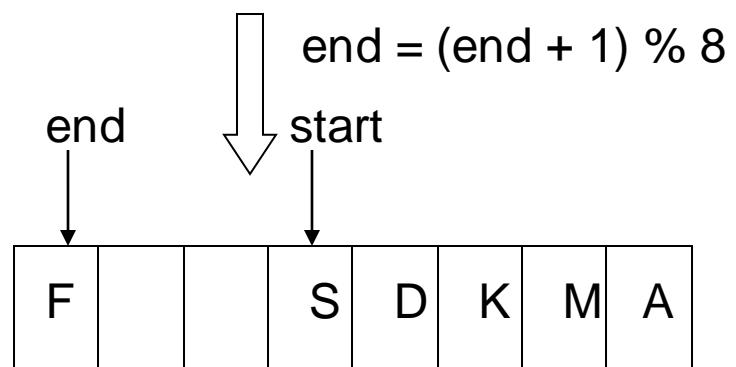
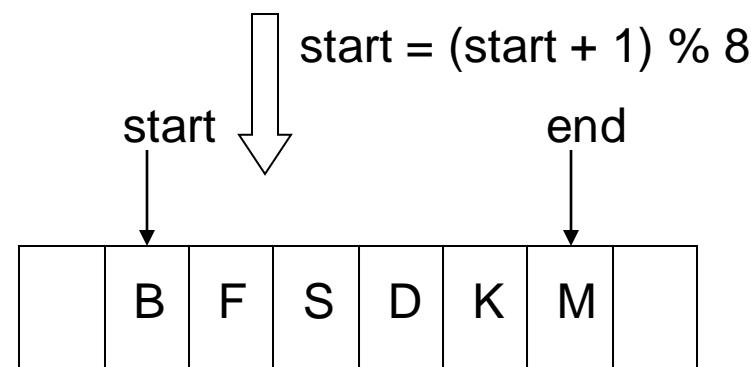
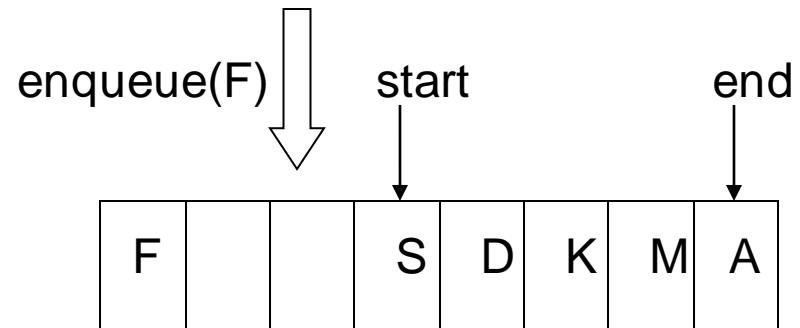
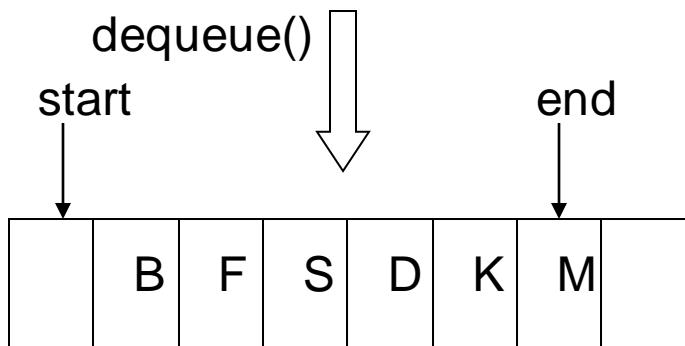
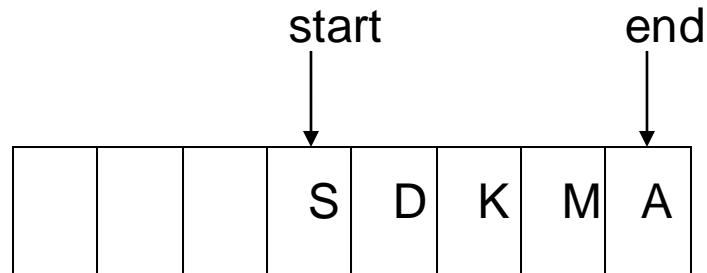
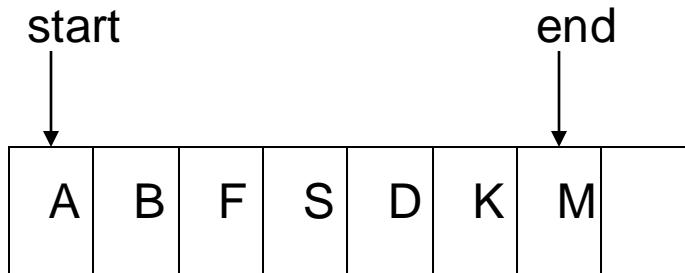
# Regular Linear Array

- Very expensive structure to use for a queue
  - shifting all the data down by one is very time consuming
- Would prefer to let the data “wrap-around”
  - the start would not always be zero
    - it would be the first occupied cell
  - the last item may actually appear in the array before the first item
  - this is called a *circular array*

# Circular Array

- Need to keep track of the index that holds the first item
- Need to keep track of the index that holds the last item
- The “wrap-around” is accomplished through the use of the *mod* operator (%)
  - $\text{index} = (\text{end} + 1) \% \text{array.length}$ 
    - assume  $\text{array.length} = 5$  and  $\text{end} = 4$
    - then:  $\text{index} = (4 + 1) \% 5 = 0$

# Circular Array



# Circular List

- If the list is empty, *start* and *end* would refer to the same spot
- If the list is full, *start* and *end* would refer to the same spot
- How do you tell the difference between an empty and a full list?
  - if the list is empty, make *start* and *end* refer to -1
  - then if *start* and *end* both refer to an element greater than -1, the list is full

# Implementing Queues: Array

- Advantage:
  - best performance
- Disadvantage:
  - fixed size
- Basic implementation
  - initially empty circular array
  - two fields: *start* and *end*
    - where the next data goes in, and the next data comes out
  - if array is full, *enqueue()* returns false
    - otherwise the data is added to the queue
  - if array is empty, *dequeue()* returns null
    - otherwise removes the start and returns it

# Queue Class (array based)

```
class QueueArray {  
    private Object[ ] queue;  
    private int start, end;  
    public QueueArray(int size) {  
        queue = new Object[size];  
        start = end = -1;  
    }  
    public boolean enqueue(Object data);  
    public Object dequeue();  
    public void clear();  
    public boolean isEmpty();  
    public boolean isFull();  
}
```

# **enqueue()** Method (array based)

```
public boolean enqueue(Object data) {  
    if(((end + 1) % queue.length) == start)  
        return false; // queue is full  
  
    // move the end of the queue and add the element  
    end = (end + 1) % queue.length;  
    queue[end] = data;  
    if(start == -1) { start = 0; }  
    return true;  
}
```

# *dequeue()* Method (array based)

```
public Object dequeue() {  
    if(start == -1)  
        return null; // empty list  
  
    // get the object, update the start, and return the object  
    Object tmp = queue[start];  
    if(start == end)  
        start = end = -1;  
    else  
        start = (start + 1) % queue.length;  
    return tmp;  
}
```

# Notes on *enqueue()* and *dequeue()*

- Just implementing a circular list
  - if start and end equal -1, the list is empty
  - if start and end are the same and not equal to -1, there is only one item in the list
  - if the end of the list is one spot “behind” the start of the list, the list is full
    - `if(((end + 1) % queue.length) == start) { ... }`
  - always remove from the start and add to the end
    - make sure to move *end* before adding
    - make sure to move *start* after removing

# Remaining Methods (array based)

```
public void clear() {  
    start = end = -1;  
}
```

```
public boolean isEmpty() {  
    return start == -1;  
}
```

```
public boolean isFull() {  
    return ((end + 1) % queue.length) == start;  
}
```

# Implementing a Stack: Linked List

- Advantages:
  - can grow to an infinite size
  - lists are well suited for implementing a queue
    - makes things very easy
- Disadvantage
  - potentially slower than an array based queue
  - can grow to an infinite size
- Basic implementation
  - add new node to the tail of the list
  - remove a node from the head of the list

# Queue Class (list based)

```
class QueueList {  
    private LinkedList queue;  
    public QueueList() {  
        queue = new LinkedList();  
    }  
    public boolean enqueue(Object data) { list.addTail(data); }  
    public Object dequeue() { return list.deleteHead(); }  
    public void clear() { list.clear(); }  
    public boolean isEmpty() { return list.isEmpty(); }  
}
```

# Additional Notes

- It should appear obvious that linked lists are very well suited for queues
  - *addTail()* and *deleteHead()* are basically the *enqueue()* and *dequeue()* methods, respectively
- Our original list implementation did not have a *clear()* method
  - all it has to do is set the *head* and *tail* to null
- Again, no need for the *isFull()* method
  - list can grow to an infinite size

# Priority Queue

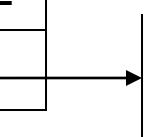
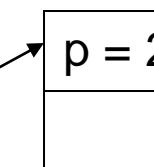
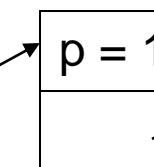
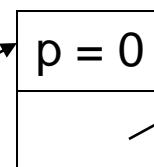
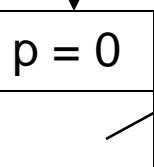
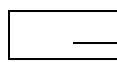
- Sometimes it is not enough just do FIFO ordering
  - may want to give some items a higher priority than other items
    - these should be serviced before lower priority even if they arrived later
- Two major ways to implement a priority queue
  - insert items in a sorted order
    - always remove the head
  - insert in unordered order and search list on remove
    - always add to the tail
  - either way, time is  $O(n)$ 
    - either adding data takes time and removing is quick, or
    - adding data is quick and removing takes time

# Inserting in Order

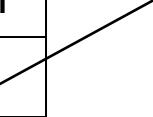
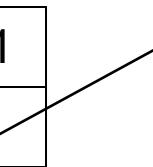
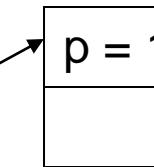
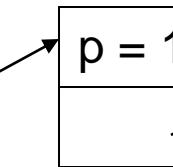
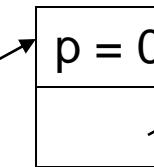
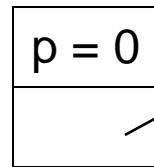
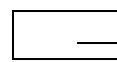
- Use the very first linked list class shown
  - only need to use the `add()` and `removeHead()` methods
    - `add()` method puts things in the list in order
  - the `compareTo()` method (implemented by your data class) should return a value based on priority
    - usually consider lower number a higher priority
  - Performance
    - $O(n)$  to add
    - $O(1)$  to remove

# Inserting in Order

head

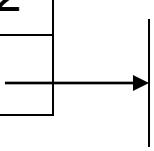
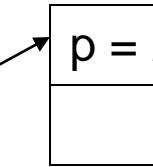
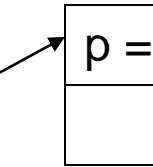
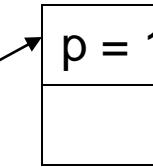
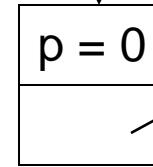
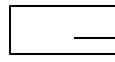


head



insert(new Data(1))

head



deleteHead()

# Queue Applications

- As with stacks, queues are very common
  - networking
    - routers queue packets before sending them out
  - operating systems
    - disk scheduling, pipes, sockets, etc.
  - system modeling and simulation
    - queueing theory
- Any of these queues can be done as a priority queue
  - consider a disk scheduler
    - higher priority is given to a job closer to the current position of the disk head
    - next request done is that closest to the current position

# Disk Scheduler

- Requests for disk sectors arrive randomly
- Disk requests are completed at a much slower rate than disk requests arrive
  - need to place waiting jobs in a queue
- All requests should be placed in a priority queue
  - jobs closest to the current position get placed closer to the front of the queue
- When the current job finishes, the next job is removed from the head of the queue

# Code for a Priority Queue Class

```
class QueuePriority {  
    private LinkedList queue;  
    public Queue() { queue = new LinkedList(); }  
    public void enqueue(Object data) { queue.add(data); }  
    public Object dequeue() { return queue.deleteHead(); }  
    public void clear() { queue.clear(); }  
    public boolean isEmpty() { return queue.isEmpty(); }  
}
```

# Code for a Disk Request Class

```
class DiskRequest implements Comparable{
    private int cylinder;
    private int head;
    private int sector;
    public DiskRequest(int cylinder, int head, int sector) {
        this.cylinder = cylinder;
        this. head = head;
        this.sector = sector;
    }
    public int get Cylinder() { return cylinder; }
    public int get Head() { return head; }
    public int get Sector() { return sector; }
    public int compareTo(Object obj) {
        DiskRequest req = (DiskRequest)obj;
        return cylinder - req.get Cylinder();
    }
    public String toString() {
        String msg = new String("Cylinder(" + cylinder + ")\\tHead(" +
                               head + ")\\tSector(" + sector + ")");
        return msg;
    }
}
```

```
public static void main(String[] args) {  
    QueuePriority diskQueue = new QueuePriority();  
    int option = getOption();  
    DiskRequest req;  
    while(option != 3) {  
        switch(option) {  
            case 1:  
                req = getRequest();  
                diskQueue.enqueue(req);  
                break;  
            case 2:  
                if(diskQueue.isEmpty())  
                    System.out.println("Disk queue is empty.");  
                else {  
                    req = (DiskRequest)diskQueue.dequeue();  
                    System.out.println("Removed request: " + req.toString());  
                }  
                break;  
            case 3:  
                break;  
            default:  
                System.out.println("Error: invalid entry.");  
        }  
        option = getOption();  
    }  
}
```

# Stacks

This lesson is borrowed from the following:

Reference

CS 367 – Introduction to Data Structures

<http://pages.cs.wisc.edu/~mattmcc/cs367/notes/Stacks.ppt>

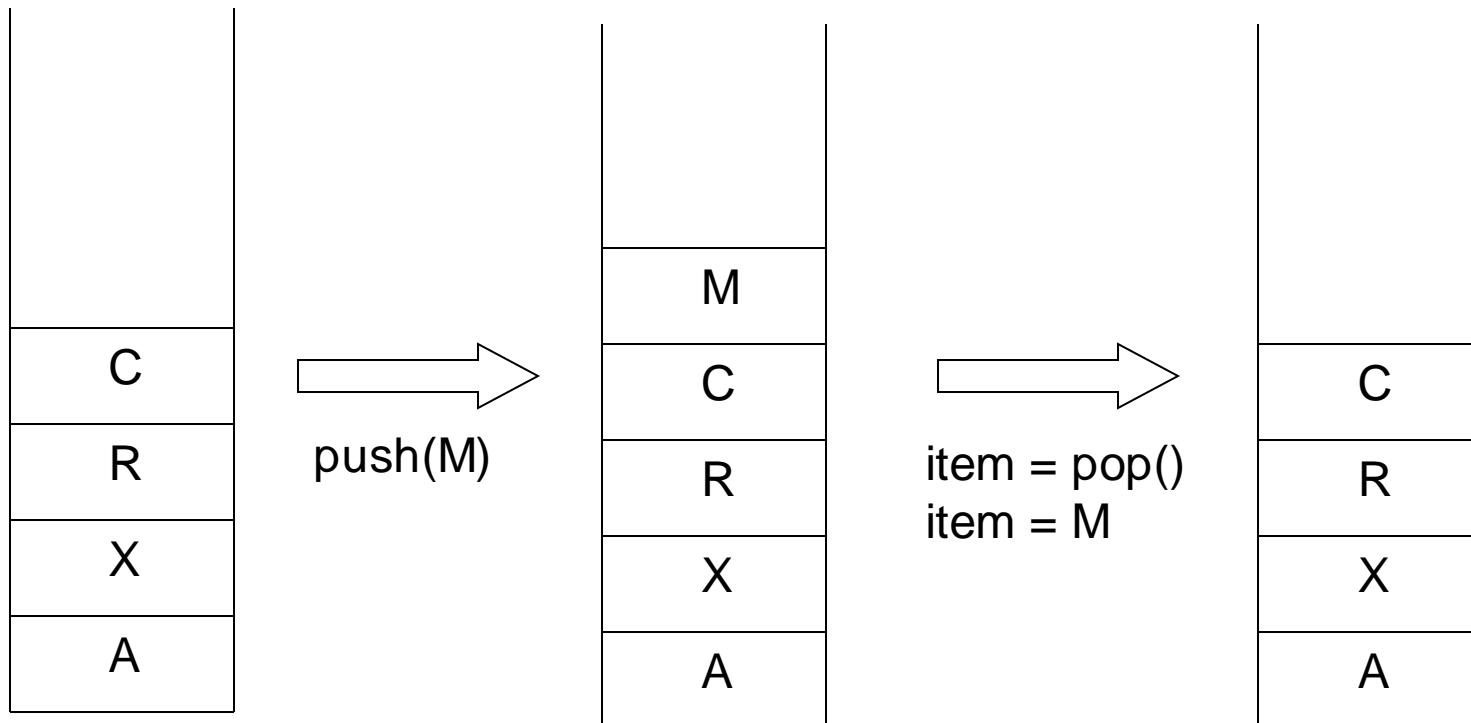
# Stack

- A stack is a data structure that stores data in such a way that the last piece of data stored, is the first one retrieved
  - also called last-in, first-out
- Only access to the stack is the top element
  - consider trays in a cafeteria
    - to get the bottom tray out, you must first remove all of the elements above

# Stack

- *Push*
  - the operation to place a new item at the top of the stack
- *Pop*
  - the operation to remove the next item from the top of the stack

# Stack



# Implementing a Stack

- At least three different ways to implement a stack
  - array
  - Vector (**not covered in this lesson**)
  - linked list
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

# Implementing Stacks: Array

- Advantages
  - best performance
- Disadvantage
  - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
    - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
    - otherwise removes the next item in the stack

# Stack Class (array based)

```
class StackArray {  
    private Object[ ] stack;  
    private int nextIn;  
    public StackArray(int size) {  
        stack = new Object[size];  
        nextIn = 0;  
    }  
    public boolean push(Object data);  
    public Object pop();  
    public void clear();  
    public boolean isEmpty();  
    public boolean isFull();  
}
```

# *push()* Method (array based)

```
public boolean push(Object data) {  
    if(nextIn == stack.length) { return false; } // stack is full  
  
    // add the element and then increment nextIn  
    stack[nextIn] = data;  
    nextIn++;  
    return true;  
}
```

# *pop()* Method (array based)

```
public Object pop() {  
    if(nextIn == 0) { return null; } // stack is empty  
  
    // decrement nextIn and return the data  
    nextIn--;  
    Object data = stack[nextIn];  
    return data;  
}
```

# Notes on *push()* and *pop()*

- Other ways to do this even if using arrays
  - may want to keep a *size* variable that tracks how many items in the list
  - may want to keep a *maxSize* variable that stores the maximum number of elements the stack can hold (size of the array)
    - you would have to do this in a language like C++
  - could add things in the opposite direction
    - keep track of *nextOut* and decrement it on every push; increment it on every pop

# Remaining Methods (array based)

```
public void clear() {  
    nextIn = 0;  
}
```

```
public boolean isEmpty() {  
    return nextIn == 0;  
}
```

```
public boolean isFull() {  
    return nextIn == stack.length;  
}
```

# Additional Notes

- Notice that the array is considered empty if *nextIn* equals zero
  - doesn't matter if there is more data stored in the array – it will never be retrieved
    - *pop()* method will automatically return
- For a truly robust implementation
  - should set array elements equal to null if they are not being used
    - why? how?

# Implementing a Stack: Linked List

- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - the common case is the slowest of all the implementations
  - can grow to an infinite size
- Basic implementation
  - list is initially empty
  - *push()* method adds a new item to the head of the list
  - *pop()* method removes the head of the list

# Stack Class (list based)

```
class StackList {  
    private LinkedList list;  
    public StackList() { list = new LinkedList(); }  
    public void push(Object data) { list.addHead(data); }  
    public Object pop() { return list.deleteHead(); }  
    public void clear() { list.clear(); }  
    public boolean isEmpty() { return list.isEmpty(); }  
}
```

# Additional Notes

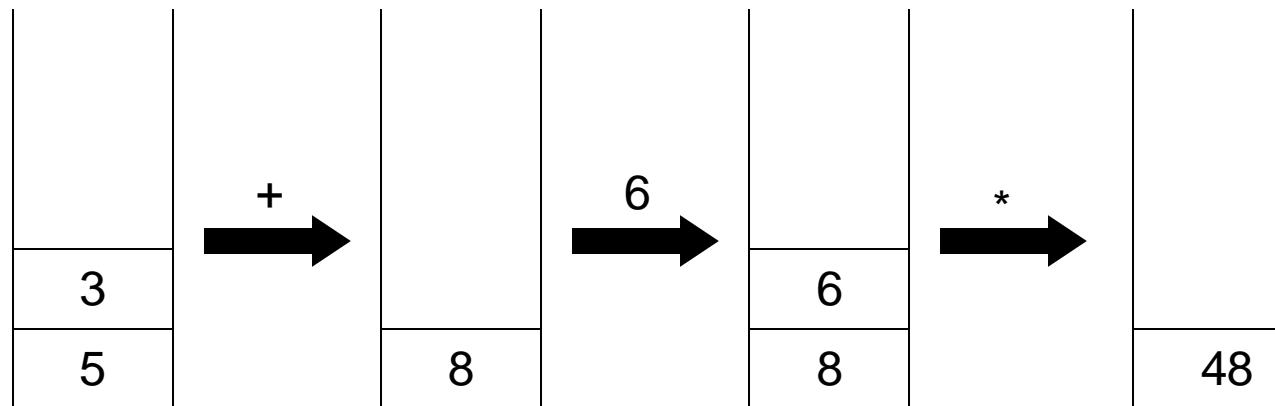
- It should appear obvious that linked lists are very well suited for stacks
  - *addHead()* and *deleteHead()* are basically the *push()* and *pop()* methods
- Our original list implementation did not have a *clear()* method
  - it's very simple to do
  - how would you do it?
- Again, no need for the *isFull()* method
  - list can grow to an infinite size

# Stack Applications

- Stacks are a very common data structure
  - compilers
    - parsing data between delimiters (brackets)
  - operating systems
    - program stack
  - virtual machines
    - manipulating numbers
      - pop 2 numbers off stack, do work (such as add)
      - push result back on stack and repeat
  - artificial intelligence
    - finding a path

# Reverse Polish Notation

- Way of inputting numbers to a calculator
  - $(5 + 3) * 6$  becomes  $5\ 3\ +\ 6\ *$
  - $5 + 3 * 6$  becomes  $5\ 3\ 6\ *\ +$
- We can use a stack to implement this
  - consider  $5\ 3\ +\ 6\ *$

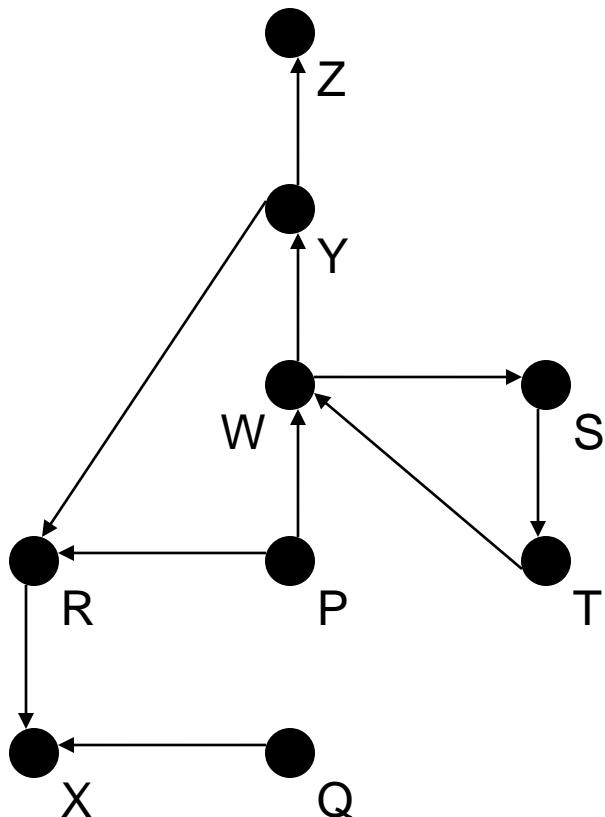


– try doing  $5\ 3\ 6\ *\ +$

```
public int rpn(String equation) {  
    StackList stack = new StackList();  
    StringTokenizer tok = new StringTokenizer(equation);  
    while(tok.hasMoreTokens()) {  
        String element = tok.nextToken();  
        if(isOperator(element)) {  
            char op = element.charAt(0);  
            if(op == '=') {  
                int result = ((Integer)stack.pop()).intValue();  
                if(!stack.isEmpty() || tok.hasMoreTokens()) { return Integer.MAX_VALUE; } // error  
                else { return result; }  
            }  
            else {  
                Integer op1 = (Integer)stack.pop()  
                Integer op2 = (Integer)stack.pop();  
                if((op1 == null) || (op2 == null)) { return Integer.MAX_VALUE; }  
                stack.push(doOperation(op, op1, op2));  
            }  
        }  
        else {  
            Integer operand = new Integer(Integer.parseInt(element));  
            stack.push(operand);  
        }  
    }  
    return Integer.MAX_VALUE;  
}
```

# Finding a Path

- Consider the following graph of flights



---

## Key

● : city (represented as C)

$C_1 \rightarrow C_2$  : flight from city  $C_1$  to city  $C_2$

---

## Example

W → S      flight goes from W to S

# Finding a Path

- If it exists, we can find a path from any city  $C_1$  to another city  $C_2$  using a stack
  - place the starting city on the bottom of the stack
    - mark it as visited
    - pick any arbitrary arrow out of the city
      - city cannot be marked as visited
    - place that city on the stack
      - also mark it as visited
    - if that's the destination, we're done
    - otherwise, pick an arrow out of the city currently at
      - next city must not have been visited before
      - if there are no legitimate arrows out, pop it off the stack and go back to the previous city
    - repeat this process until the destination is found or all the cities have been visited

# Example

- Want to go from P to Y
  - push P on the stack and mark it as visited
  - pick R as the next city to visit (random select)
    - push it on the stack and mark it as visited
  - pick X as the next city to visit (only choice)
    - push it on the stack and mark it as visited
  - no available arrows out of X – pop it
  - no more available arrows from R – pop it
  - pick W as next city to visit (only choice left)
    - push it on the stack and mark it as visited
  - pick Y as next city to visit (random select)
    - this is the destination – all done

# Psuedo-Code for the Example

```
public boolean findPath(City origin, City destination) {  
    StackArray stack = new Stack(numCities);  
    clearAllCityMarks();  
    stack.push(origin);  
    origin.mark();  
    while(!stack.isEmpty()) {  
        City next = pickCity();  
        if(next == destination) { return true; }  
        if(next != null) { stack.push(next); }  
        else { stack.pop(); } // no valid arrows out of city  
    }  
    return false;  
}
```

# Lecture 11.1

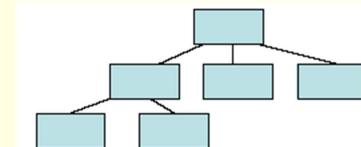
## Tree data structure

CS112, semester 2, 2007

1

## Trees

- Trees are structures that represent data in a hierarchical manner
- The top node is called the root and it may have one or several leaves or children



CS112, semester 2, 2007

2

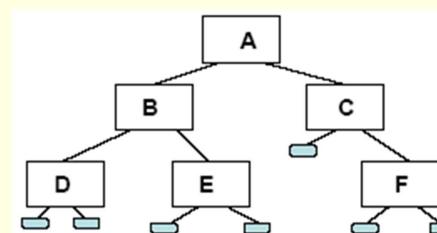
## Binary Trees

- Binary trees have at most two children (left and right)
- A binary tree is either:
  - A leaf with no branches; or
  - consist of a root and two children, left and right, each of which are themselves binary trees.
- This is a recursive definition!

CS112, semester 2, 2007

3

## Binary Tree representation

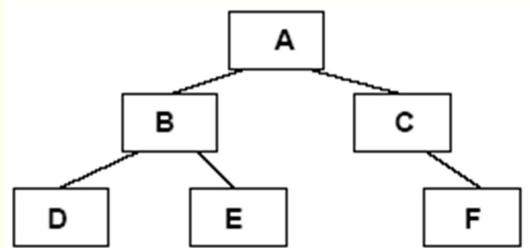


- A node without branches is a leaf (Here D, E and F are leaves) or
- A binary tree with a left and a right node, each of which is a binary tree.

CS112, semester 2, 2007

4

## Binary Tree representation



- A is a root
- B and C are left and right branches respectively
- D, E and F are leaves

CS112, semester 2, 2007

5

## Binary class template

```
template <class dataType>
class BinaryTreeNode {
public:
    BinaryTreeNode();
    bool isLeaf();
    dataType getData();
    void setData( const dataType & d );
    BinaryTreeNode * getLeft();
    BinaryTreeNode * getRight();
    void setLeft ( BinaryTreeNode * T1 );
    void setRight ( BinaryTreeNode *T1 );
private:
    dataType treeNodeData;
    BinaryTreeNode * leftTreeNode;
    BinaryTreeNode * rightTreeNode;
};
```

CS112, semester 2, 2007

7

## Node implementation

- Each node can be implemented with the following code:

```
struct TreeNode{
    int data;
    TreeNode* leftTreeNode;
    TreeNode* rightTreeNode;
};
```

OR

CS112, semester 2, 2007

6

## Constructor

```
template <class dataType> //constructor
BinaryTreeNode<dataType>::BinaryTreeNode
()
{
    leftTreeNode = 0;
    rightTreeNode = 0;
}
```

CS112, semester 2, 2007

8

## Binary tree implementation

```
template <class dataType>
bool BinaryTreeNode<dataType>::isLeaf()
{
    return ((this->leftTreeNode == NULL) && (this-
        >rightTreeNode == NULL));
}
template <class dataType>
dataType BinaryTreeNode<dataType>::getData( )
{
    return treeNodeData;
}
```

CS112, semester 2, 2007

9

## Binary tree implementation (cont.)

```
template <class dataType>
void BinaryTreeNode <dataType>::setData ( const
    dataType & d )
{
    treeNodeData = d;
}
```

CS112, semester 2, 2007

10

## Binary tree implementation (cont.)

```
template <class dataType>
BinaryTreeNode <dataType> *
BinaryTreeNode<dataType>:: getLeft( )
{
    return leftTreeNode;
}
template <class dataType>
BinaryTreeNode <dataType> *
BinaryTreeNode<dataType>:: getRight( )
{
    return rightTreeNode;
}
```

CS112, semester 2, 2007

11

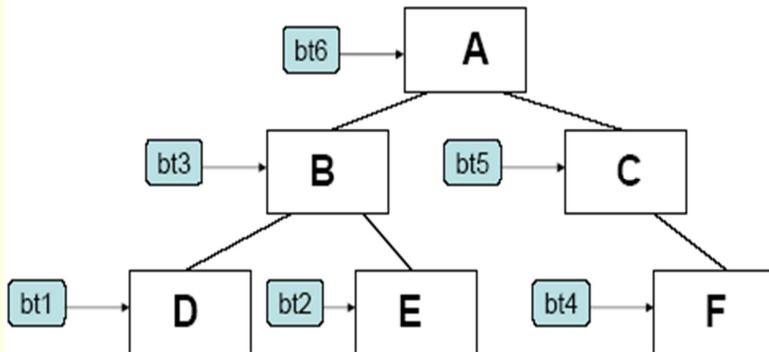
## Binary tree implementation (cont.)

```
template <class dataType>
void BinaryTreeNode <dataType> :: setLeft (
    BinaryTreeNode * T1 )
{
    leftTreeNode = T1;
}
template <class dataType>
void BinaryTreeNode<dataType> :: setRight (
    BinaryTreeNode * T1 )
{
    rightTreeNode = T1;
}
```

CS112, semester 2, 2007

12

## Example



CS112, semester 2, 2007

13

## Example

```
int main()
{
    typedef BinaryTreeNode <char> charTree;
    typedef charTree * charTreePtr;

    //Create left subtree ( rooted at B )
    //Create B's left subtree
    charTreePtr bt1 = new charTree;
    bt1->insert ( 'D' );

    //Create B's right subtree
    charTreePtr bt2 = new charTree;
    bt2->insert ( 'E' );
```

CS112, semester 2, 2007

14

## Example (cont.)

```
//Create node containing B, and link
//up to subtrees
charTreePtr bt3 = new charTree;
bt3->insert ( 'B' );
bt3->makeLeft ( bt1 );
bt3->makeRight ( bt2 );
/** done creating left subtree
```

CS112, semester 2, 2007

15

## Example (cont.)

```
//Create right subtree
//Create C's right subtree
charTreePtr bt4 = new charTree;
bt4->insert ( 'F' );

//Create node containing C and link
//up its right subtree;
charTreePtr bt5 = new charTree;
bt5->insert ( 'C' );
bt5->makeRight ( bt4 );
/** done creating right subtree
```

CS112, semester 2, 2007

16

## Example (cont.)

```
//Create the root of the tree and link together  
charTreePtr bt6 = new charTree;  
bt6->insert('A');  
bt6->makeLeft( bt3 );  
bt6->makeRight( bt5 );  
  
//print out the root  
cout<< "Root contains: " << bt6 ->getData() << endl;  
  
//print out the left subtree  
cout <<"Left subtree root: " <<bt6 ->left( ) ->getData( )  
    <<endl;
```

CS112, semester 2, 2007

17

## Example (cont.)

- //print out the right subtree
- cout <<"Right subtree root: "  
 << bt6->right( )->getData( ) << endl;
  
- //print out left most child in tree
- cout <<"Left most child is: "<<  
 bt6->left( )->left( )->getData( ) << endl;
  
- //print out right most child in tree
- cout <<"Left most child is: "<<  
 bt6->right( )->right( )->getData( ) << endl;

CS112, semester 2, 2007

18

## Lecture 11.2

### Traversing Tree

CS112, semester 2, 2007

1

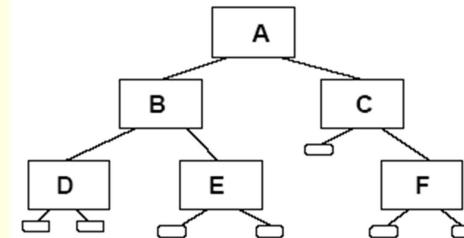
### Tree Traversal

- Just like in linked lists we traverse the list by visiting every node in the list, we can find a way to visit every node in the tree.
- Unlike traversing the list, the best way to visit each item in a tree in an orderly fashion is not so obvious
- Where should you start?
  - At the root, maybe?
- Where should you proceed?

CS112, semester 2, 2007

3

### Binary Tree representation



- Each node is either an empty tree or
- A binary tree with a left and a right node, each of which is a binary tree.

CS112, semester 2, 2007

2

### Traversal schemes

- These schemes take advantage of the recursive nature of the tree.
- The basic idea is:
  - visit the root directly
  - visit the children recursively
  - If we find an empty branch (i.e. tree pointer pointing to NULL), no action is required
- so this serves as the base case for the recursion.

CS112, semester 2, 2007

4

## Preorder Traversal

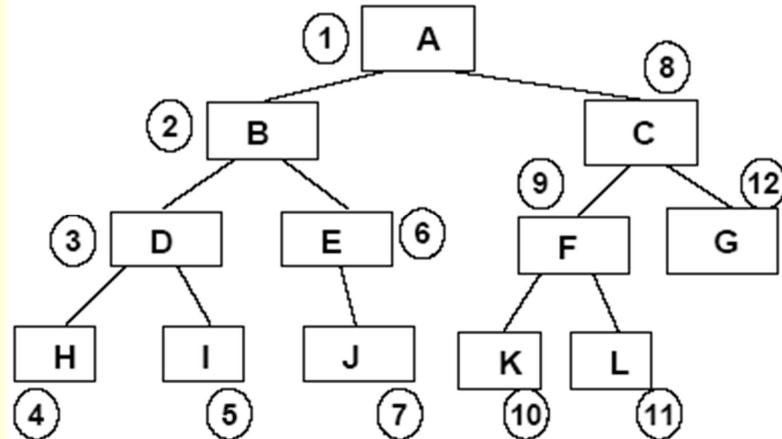
If the tree is not NULL

1. visit the root
2. preOrderTraverse (left child)
3. preOrderTraverse (right child)

CS112, semester 2, 2007

5

## Preorder Traversal



CS112, semester 2, 2007

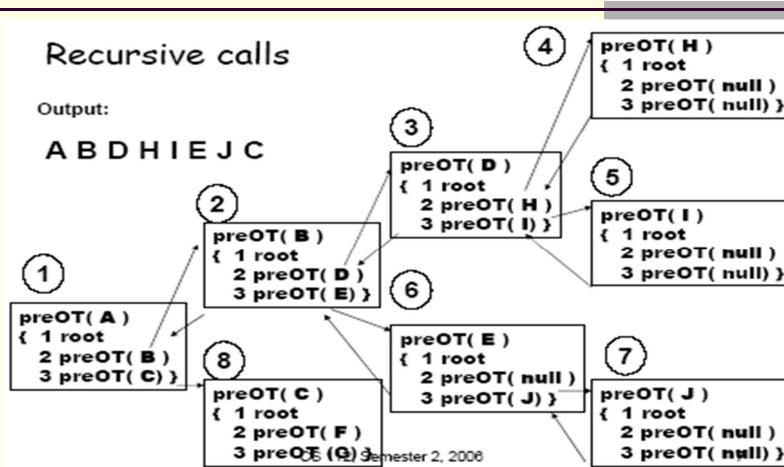
6

## Preorder Traversal

### Recursive calls

Output:

A B D H I E J C



CS112, semester 2, 2006

7

## Preorder Traverse code

```
template <class dataType>
void preOrderTraverse (BinaryTreeNode <dataType>* bt)
{
    if ( ! (bt == NULL) )
    {
        //visit tree
        cout << bt -> getData ( ) << "\t";
        //traverse left child
        preOrderTraverse ( bt -> left ( ) );
        //traverse right child
        preOrderTraverse ( bt -> right ( ) );
    }
}
```

CS112, semester 2, 2007

8

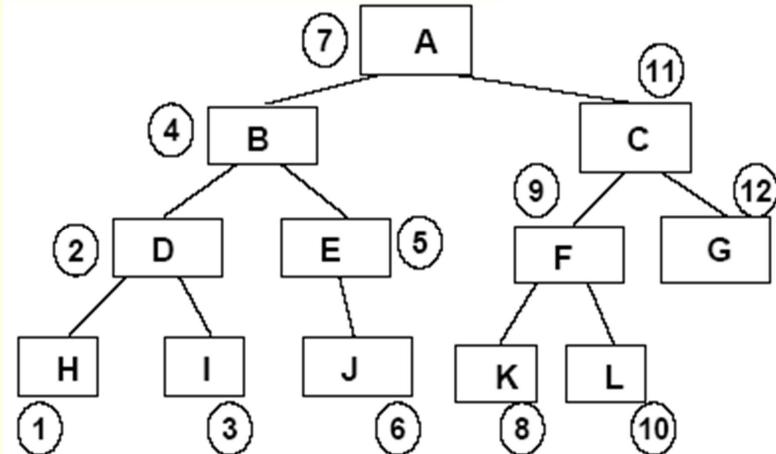
## Inorder Traversal

- If the tree is not NULL
  - 1. inOrderTraverse (left child)
  - 2. visit the root
  - 3. inOrderTraverse (right child)

CS112, semester 2, 2007

9

## Inorder Traversal



CS112, semester 2, 2007

10

## Code for InorderTraverse

```
template <class dataType>
void inOrderTraverse (BinaryTreeNode <dataType> * bt)
{
    if ( ! (bt == NULL) )
    {
        //traverse left child
        inOrderTraverse ( bt -> left ( ) );
        //visit tree
        cout << bt->getData ( ) << "\t";
        //traverse right child
        inOrderTraverse ( bt -> right ( ) );
    }
}
```

CS112, semester 2, 2007

11

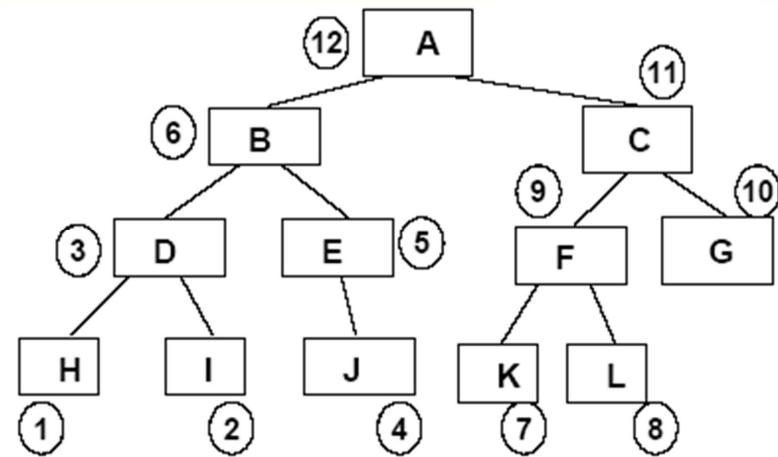
## Postorder Traversal

- If the tree is not NULL
  - 1. postOrderTraverse (left child )
  - 2. postOrderTraverse (right child)
  - 3. visit the root

CS112, semester 2, 2007

12

## Postorder Traversal



CS112, semester 2, 2007

13

## Postorder traversal code

```
template <class dataType>
void postOrderTraverse(BinaryTreeNode <dataType> * bt)
{
    if ( ! (bt == NULL) )
    {
        //traverse left child
        postOrderTraverse ( bt->left ( ) );
        //traverse right child
        postOrderTraverse ( bt -> right ( ) );
        //visit tree
        cout << bt -> getData ( ) << "\t";
    }
}
```

CS112, semester 2, 2007

14

# Lecture 11.3

## Binary Tree and Reverse Polish Notation

CS112, semester 2, 2007

1

CS112, semester 2, 2007

2

## Infix Notation

- This notation is not very clear every time:
- Example:  
What is the result of  
 $3 - 4 - 5 * 3 = ?$   
evaluating from left to right : -18  
evaluating according to precedence rules: -16

CS112, semester 2, 2007

3

## Infix Notation

- An arithmetic calculator evaluates an expression such as  $5+7 * 3$  to determine that it is equal to 26.
- This kind of expression is known as ***infix*** because the *operator* (+) is written between two *operands* (5,7)

## Reverse Polish Notation

- We can use a different notation to make an expression more clear without using parenthesis.
- Reverse Polish Notation (RPN)** is also known as **postfix** notation

CS112, semester 2, 2007

4

## Reverse Polish Notation

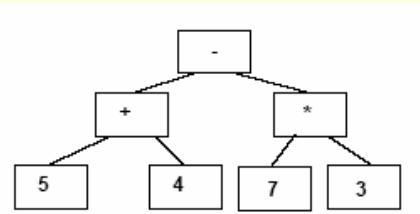
- Instead of having an operator **between** two operands, you will have an operator **after** two operands
- Instead of:  $3 - 4 - 5 * 3$
- You have:  $34 - 53 * -$

CS112, semester 2, 2007

5

## Implementation with a binary Tree

- For example:  
If your given expression =  $54 + 73 * -$   
You can build a tree to represent the expression



CS112, semester 2, 2007

7

## Evaluation rules for RPN

- Evaluate expressions from left to right
- At each occurrence of an operator, apply it to the two operands to the immediate left, and replace the sequence of two operands and one operator by the resulting value.

The expression :  $34 - 53 * -$   
Evaluates to  $(3-4)(5*3)-$   
 $-1 15 -$   
 $(-1 -15) = -16$

CS112, semester 2, 2007

6

- Write a function to evaluate this expression
- Hint: Which traversal scheme would help you?

CS112, semester 2, 2007

8

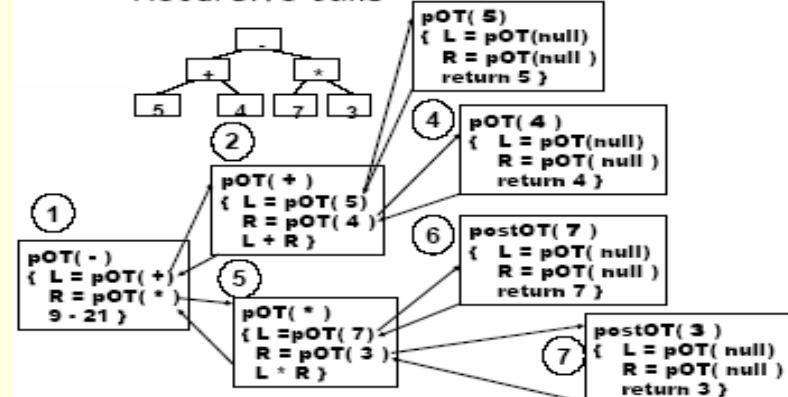
# PostOrderEvaluate

```
double postOrderEvaluate ( BinaryTree <dataType> * bt )
{
    double left, right;
    if ( bt != NULL )
    {
        //traverse left child
        left = postOrderEvaluate ( bt->left () );
        //traverse right child
        right = postOrderEvaluate ( bt->right() );
        //visit tree
        if ( strcmp(bt -> getData(), "+") == 0 )
        {
            cout << endl << left << "+" << right;
            return (left + right);
        }
        else if ( strcmp(bt -> getData(), "-") == 0 )
        {
            cout << endl << left << "-" << right;
            return (left - right);
        }
        else if ( strcmp(bt -> getData(),"*") == 0 )
        {
            cout << endl << left << "*" << right;
            return (left * right);
        }
        else
            return atof(bt -> getData());
    }
    return 0;
}
```

CS112, semester 2, 2007

9

## Recursive calls



CS112, semester 2, 2007

10

# Binary Trees – Part I

This lesson is borrowed from the following:

Reference

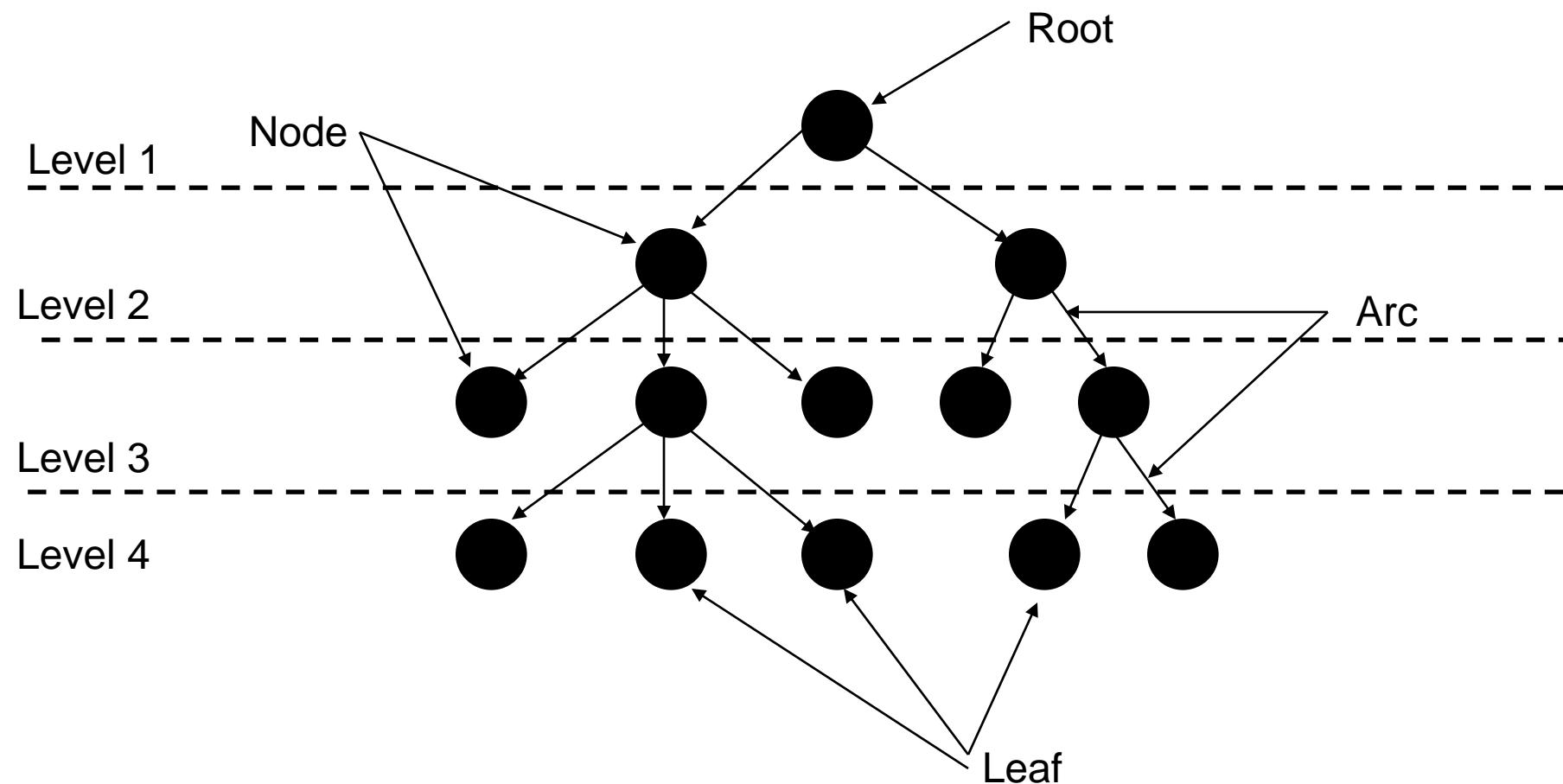
CS 367 – Introduction to Data Structures

<http://pages.cs.wisc.edu/~mattmcc/cs367/notes/Trees-I.ppt>

# Trees

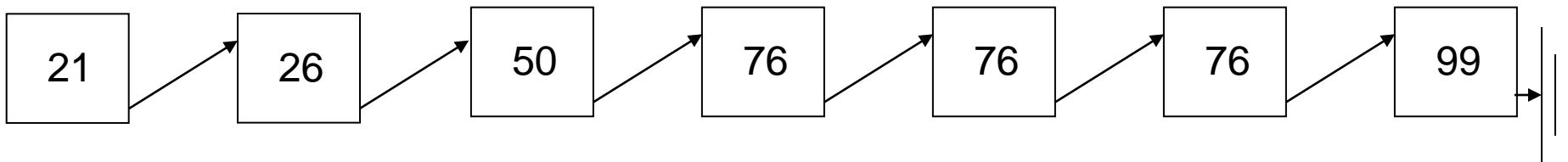
- Nodes
  - element that contains data
- Root
  - node with children and no parent
- Leaves
  - node with a parent and no children
- Arcs
  - connection between a parent and a child
- Depth
  - number of levels in a tree
- A node can have 1 to n children – no limit
- Each node can have only one parent

# Tree of Depth 4



# Trees vs. Linked Lists

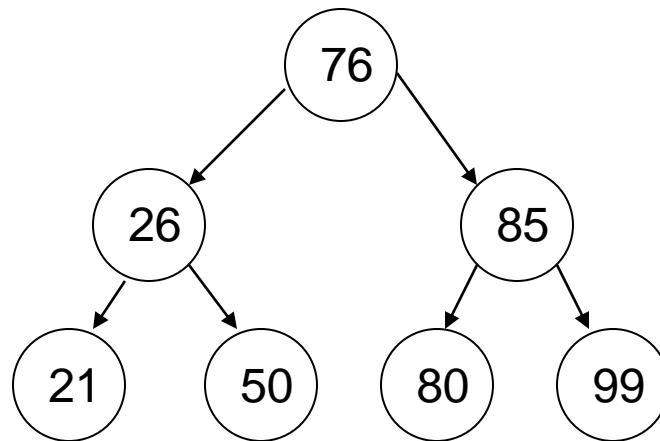
- Imagine the following linked list



- How many nodes must be touched to retrieve the number 99?
  - answer: 7
  - must touch every item that appears in the list before the one wanted can be retrieved
  - random insert, delete, or retrieve is  $O(n)$

# Trees vs. Linked Lists

- Now consider the following tree



- How many nodes must now be touched to retrieve the number 99?
  - answer: 3
  - random insert, delete, or retrieve is  $O(\log n)$

# Trees vs. Linked Lists

- Similarities
  - both can grow to an unlimited size
  - both require the access of a previous node before the next one
- Difference
  - access to a previous node can give access to multiple next nodes
    - if we're smart (and we will be) we can use this fact to drastically reduce search times

# Trees vs. Arrays

- Consider the following array

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 21 | 26 | 50 | 76 | 76 | 76 | 99 |
|----|----|----|----|----|----|----|

- How many nodes must be touched to retrieve the number 99?
  - answer: 3
    - remember the binary search of a sorted array?
  - searching a sorted array is  $O(\log n)$
  - how about inserting or deleting from an array
    - $O(n)$

# Trees vs. Arrays

- Similarities
  - searching the list takes the same time
- Differences
  - inserting or deleting from an array is more time consuming
  - an array is fixed in size

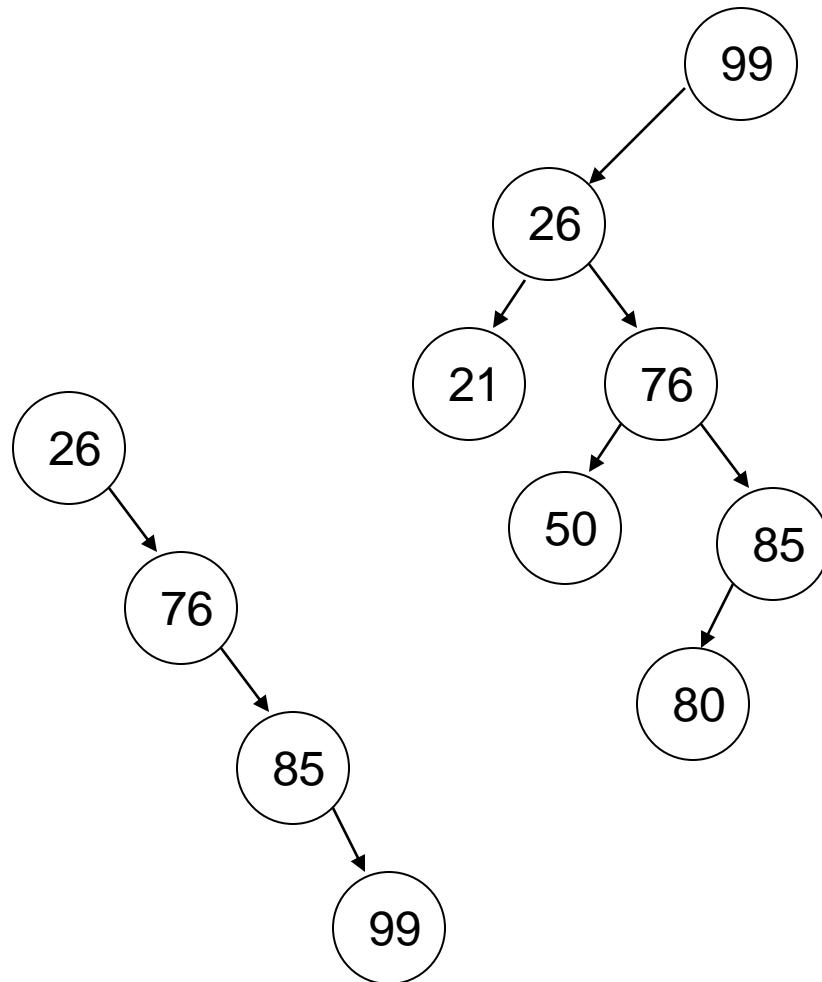
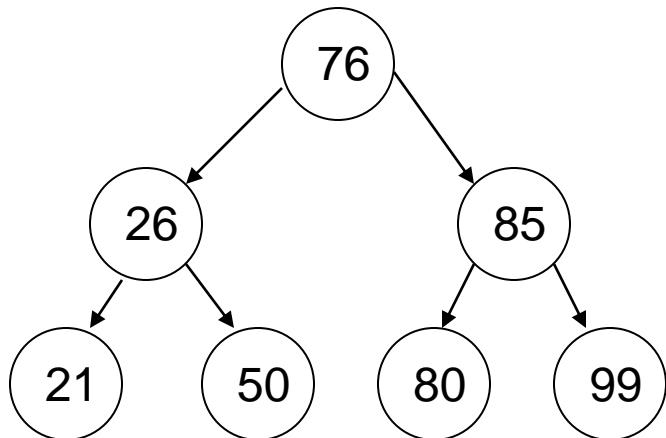
# Tree Operations

- Sorting
  - way to guarantee the placement of one node with respect to other nodes
- Searching
  - finding a node based on a *key*
- Inserting
  - adding a node in a sorted order to the tree
- Deleting
  - removing a node from the tree in such a way that the tree is still sorted

# Binary Tree

- One specific type of tree is called a *binary tree*
  - each node has at most 2 children
    - right child and left child
  - sorting the tree is based on the following criteria
    - every item rooted at node N's right child is greater than N
    - every item rooted at node N's left child is less than N

# Binary Trees



# Implementing Binary Trees

- Each node in the tree must contain a reference to the data, key, right child, and left child

```
class TreeNode {  
    public Object key;  
    public Object data;  
    public TreeNode right;  
    public TreeNode left;  
    public TreeNode(Object k, Object data) { key=k; data=d; }  
}
```

- Tree class only needs reference to root of tree
  - as well as methods for operating on the tree

# Implementing Binary Trees

```
class BinaryTreeRec {  
    private TreeNode root;  
    public BinaryTree() { root = null; }  
    public Object search(Object key) { search(key, root); }  
    private Object search(Object key, TreeNode node);  
    public void insert(Object key, Object data) { insert(key, data, root); }  
    private void insert(Object key, Object data, TreeNode node);  
    public Object delete(Object key) { delete(key, root, null); }  
    private Object delete(Object key, TreeNode cur, TreeNode prev);  
}
```

# Searching a Binary Tree

- Set reference P equal to the root node
  - if P is equal to the key
    - found it
  - if P is less than key
    - set P equal to the right child node and repeat
  - if P is greater than key
    - set P equal to the left child node and repeat

# Recursive Binary Tree Search

```
private Object search(Object key, TreeNode node) {  
    if(node == null) { return null; }  
  
    int result = node.key.compareTo(key);  
    if(result == 0)  
        return node.data; // found it  
    else if(result < 0)  
        return search(key, node.right); // key in right subtree  
    else  
        return search(key, node.left); // key in left subtree  
}
```

# Inserting into Binary Tree

- Set reference P equal to the root node
  - if P is equal to null
    - insert the node at position P
  - if P is less than node to insert
    - set P equal to the right child node and repeat
  - if P is greater than node to insert
    - set P equal to the left child node and repeat

# Recursive Binary Tree Insert

```
private void insert(Object key, Object data, TreeNode node) {  
    if(node == null) {  
        root = new TreeNode(key, data);  
        return;  
    }  
  
    int result = node.key.compareTo(key);  
    if(result < 0) {  
        if(node.right == null)  
            node.right = new TreeNode(key, data);  
        else  
            insert(key, data, node.right);  
    }  
    else {  
        if(node.left == null)  
            node.left = new TreeNode(key, data);  
        else  
            insert(key, data, node.left);  
    }  
}
```

# Binary Tree Insert

- One important note from this insert
  - the key cannot already exists
    - if it does, an error should be returned (our code did not do this)
  - all keys in a tree must be unique
    - have to have some way to differentiate different nodes

# Binary Tree Delete

- Two possible methods
  - delete by merging
    - discussed in the book
    - problem is that it can lead to a very unbalanced tree
  - delete by copying
    - this is the method we will use
    - can still lead to an unbalanced tree, but not nearly as severe

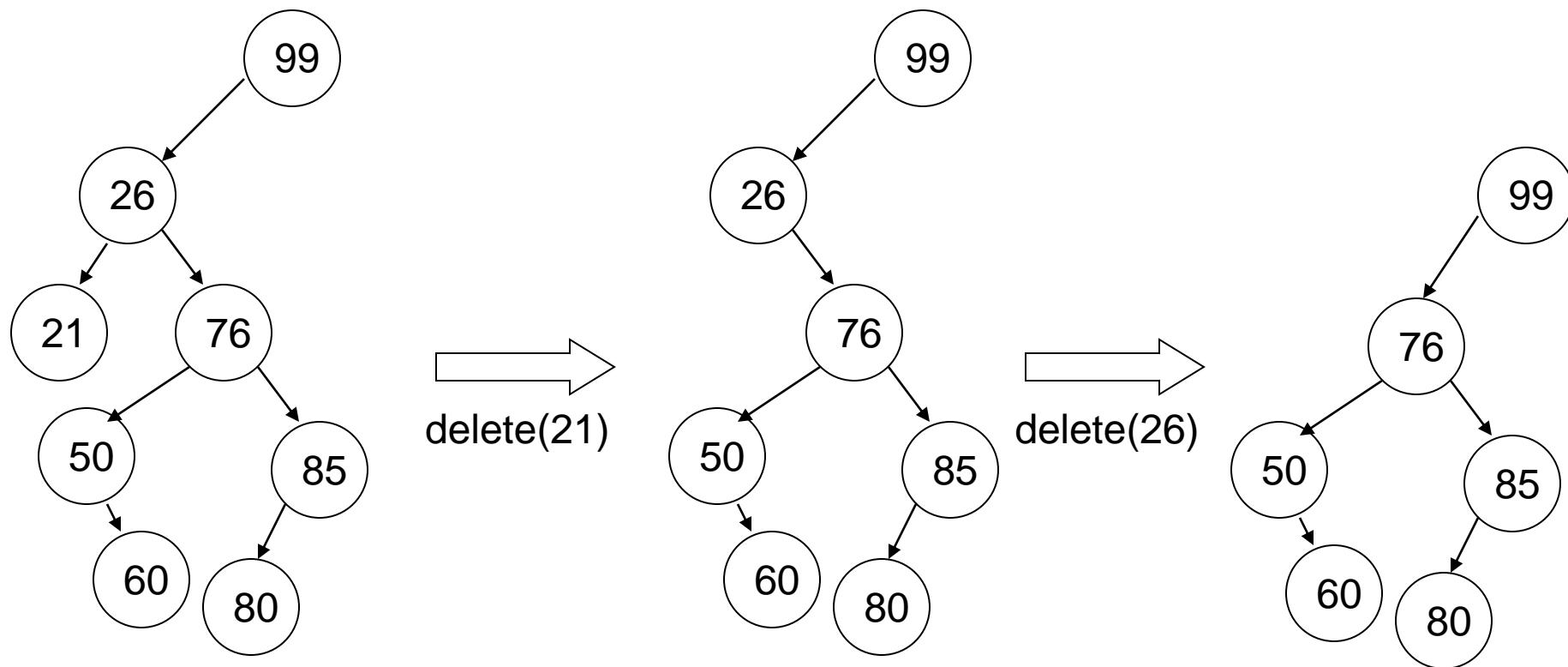
# Binary Tree Delete

- Set reference P equal to the root node
  - search the tree to find the desired node
    - if it doesn't exist, return null
  - once node is found,
    - replace it
    - this is going to require some more explanation

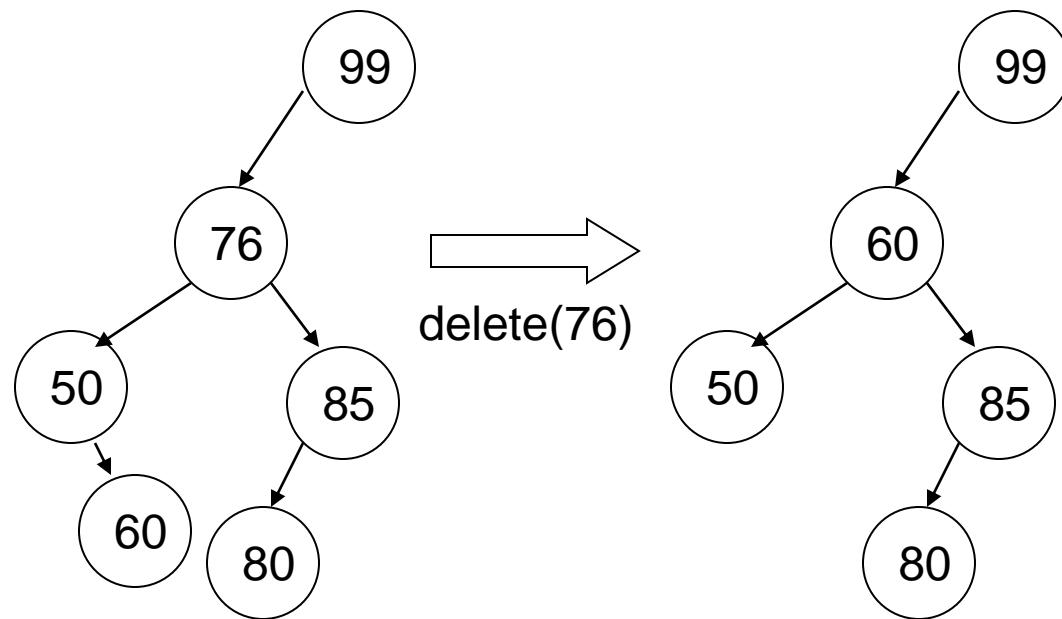
# Binary Tree Delete

- Three possible cases for node to delete
  - it is a leaf
    - simple, make it's parent point to null
  - it has only a single child
    - simple, make it's parent point to the child
  - it has two children
    - need to find one of it's descendants to replace it
      - we will pick the largest node in the left subtree
      - could also pick the smallest node in the right subtree
    - this is a fairly complex operation

# Simple Cases



# Complex Case



# Complex Case

- Notice that we must first find the largest value in the left subtree
  - keep moving to the next right child until the right.next pointer is equal to null
    - this is the largest node in a subtree
  - then make this child's parent point to this child's left pointer
  - move this child into the same spot as the deleted node
    - requires the manipulation of a few pointers

# Removing a Node

```
public void remove(TreeNode node, TreeNode prev) {  
    TreeNode tmp, p;  
    if(node.right == null)  
        tmp = node.left;  
    else if(node.left == null)  
        tmp = node.right;  
    else {  
        tmp = node.left;  p = node;  
        while(tmp.right != null) {  
            p = tmp;  
            tmp = tmp.right;  
        }  
        if(p == node) { p.left = tmp.left; }  
        else { p.right = tmp.left; }  
        tmp.right = node.right;  
        tmp.left = node.left;  
    }  
    if(prev == null) { root = tmp; }  
    else if(prev.left == node) { prev.left = tmp; }  
    else { prev.right = tmp; }  
}
```

# Recursive Binary Tree Delete

```
private Object delete(Object key, TreeNode cur, TreeNode prev) {  
    if(cur == null)  
        return null; // key not in the tree  
  
    int result = cur.key.compareTo(key);  
    if(result == 0) {  
        remove(cur, prev);  
        return cur.data;  
    }  
    else if(result < 0)  
        return delete(key, cur.right, cur);  
    else  
        return delete(key, cur.left, cur);  
}
```

# Iterative Solution

- Most operations shown so far can easily be converted into an iterative solution

```
class BinaryTreelter {  
    private TreeNode root;  
    public BinaryTree() { root = null; }  
    public Object search(Object key);  
    public void insert(Object key, Object data);  
    public Object delete(Object key);  
}
```

# Iterative Binary Search

```
public Object search(Object key) {  
    TreeNode node = root;  
    while(node != null) {  
        int result = node.key.compareTo(key);  
        if(result == 0)  
            return node.data;  
        else if(result < 0)  
            node = node.right;  
        else  
            node = node.left;  
    }  
    return null;  
}
```

# Iterative Binary Tree Insert

```
public void insert(Object key, Object data) {  
    TreeNode cur = root;  
    TreeNode prev = null;  
    while(cur != null) {  
        if(cur.key.compareTo(key) < 0) { prev = cur;  cur = cur.right; }  
        else { prev = cur;  cur = cur.left; }  
    }  
  
    if(prev == null) { root = new TreeNode(key, data); }  
    else if(prev.key.compareTo(key) < 0)  
        prev.right = new TreeNode(key, data);  
    else  
        prev.left = new TreeNode(key, data);  
}
```

# Iterative Binary Tree Delete

```
public Object delete(Object key) {  
    TreeNode cur = root;  
    TreeNode prev = null;  
    while((cur != null) && (cur.key.compareTo(key) != 0)) {  
        prev = cur;  
        if(cur.key.compareTo(key) < 0) { cur = cur.right; }  
        else { cur = cur.left; }  
    }  
    if(cur != null) {  
        replace(cur, prev);  
        return cur.data;  
    }  
    return null;  
}
```

# Trees – Part 2

This lesson is borrowed from the following:

Reference

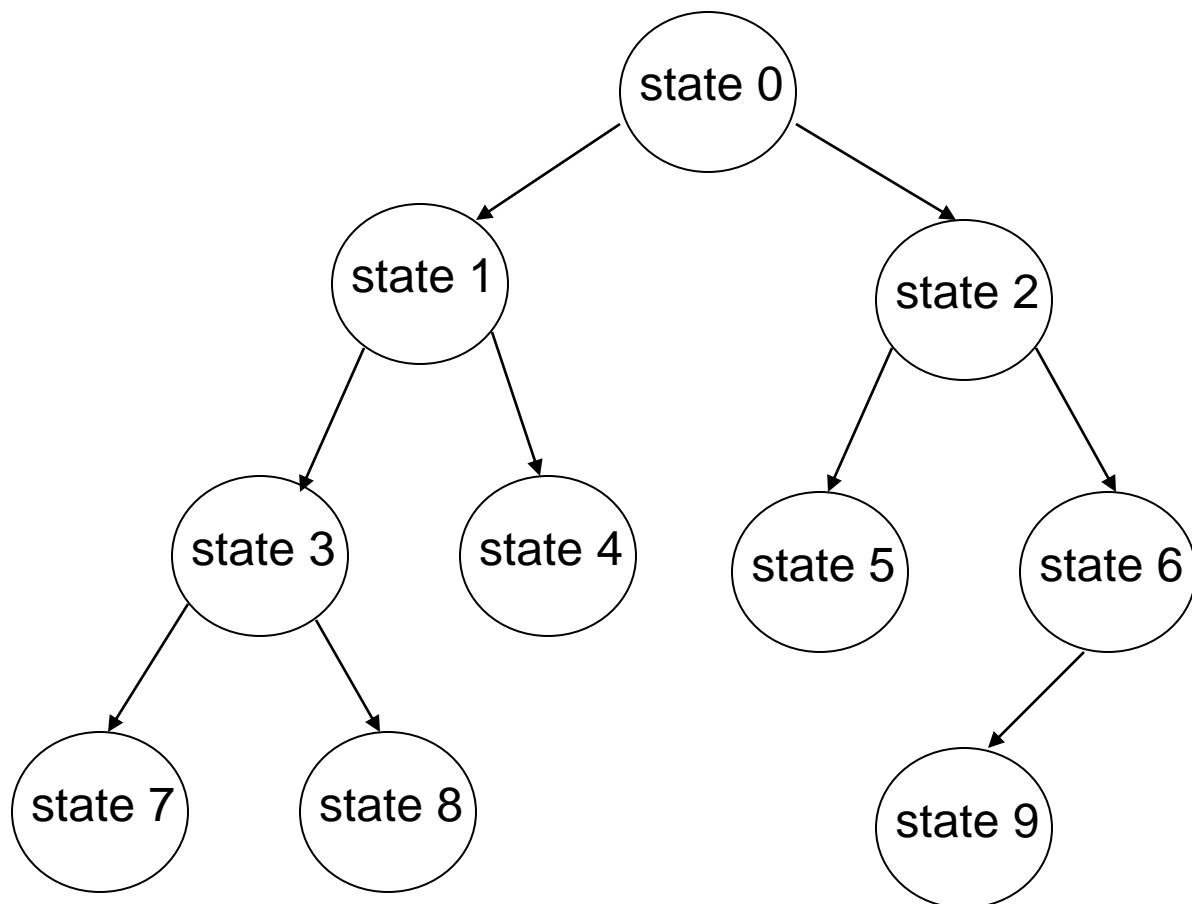
CS 367 – Introduction to Data Structures

*<http://pages.cs.wisc.edu/~mattmcc/cs367/notes/Trees-II.ppt>*

# Tree Traversal

- Sometimes necessary to scan entire tree
  - imagine a system that has no keys, just states
    - AI algorithms are a great example
  - to find the best state, must search all nodes
- Two ways to search an entire tree
  - breadth first
    - search all nodes at one level, and then go to next level
  - depth first
    - go all the way down one branch and then the next and so on

# Breadth First Search



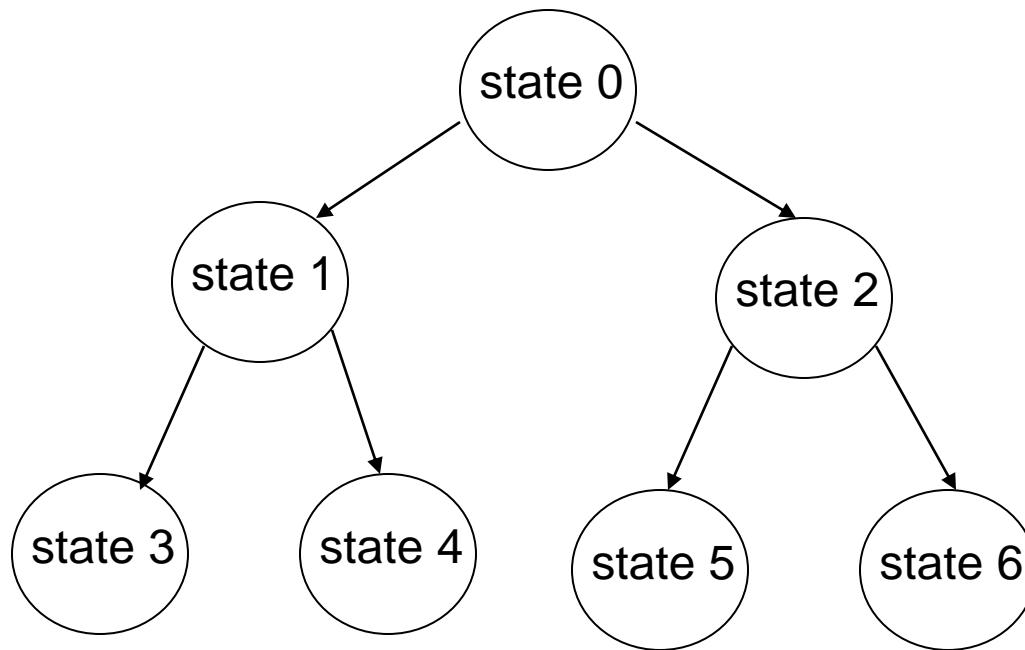
## Search Order

state 0  
state 1  
state 2  
state 3  
state 4  
state 5  
state 6  
state 7  
state 8  
state 9

# Breadth First Search

- Best way to implement a breadth first search is with a queue
  - visit a node
  - enqueue all of the nodes children
  - dequeue the next item from the queue
  - repeat until the queue is empty

# Breadth First Search

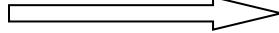


Initial Queue

|    |  |  |  |
|----|--|--|--|
| s0 |  |  |  |
|----|--|--|--|

pop s0

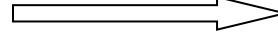
push children



|    |    |  |  |
|----|----|--|--|
| s1 | s2 |  |  |
|----|----|--|--|

pop s1

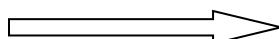
push children



|    |    |    |  |
|----|----|----|--|
| s2 | s3 | s4 |  |
|----|----|----|--|

pop s2

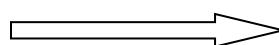
push children



|    |    |    |    |
|----|----|----|----|
| s3 | s4 | s5 | s6 |
|----|----|----|----|

pop s3

no children



|    |    |    |  |
|----|----|----|--|
| s4 | s5 | s6 |  |
|----|----|----|--|

etc.

...

# Breadth First Search

```
public void printTree-Breadth() {  
    if(root == null) {  
        System.out.println("Empty tree.");  
        return;  
    }  
    Queue queue = new QueueList();  
    queue.enqueue(root);  
    while(!queue.isEmpty()) {  
        TreeNode tmp = queue.dequeue();  
        System.out.println(tmp.data.toString());  
        if(tmp.left != null) { queue.enqueue(tmp.left); }  
        if(tmp.right != null) { queue.enqueue(tmp.right); }  
    }  
}
```

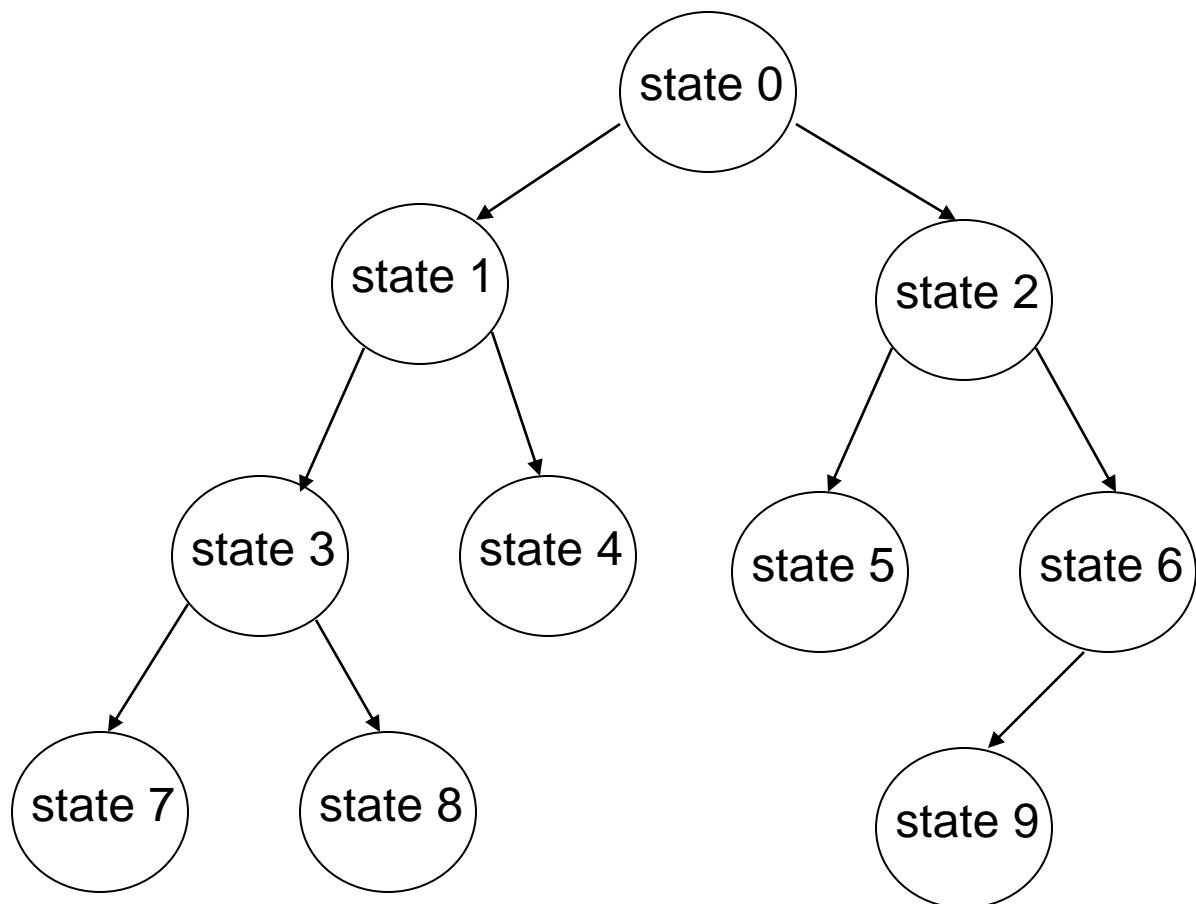
# Breadth First Search

- Advantage
  - guaranteed to find the wanted state
    - if it exists
- Disadvantage
  - excessive memory requirements
  - $M_{needed} = 2^{level - 1}$

# Depth First Search

- Three possible orderings for depth first
  - preorder
    - visit node, then its left child, then its right child
  - inorder
    - visit left child, then the node, then right child
  - postorder
    - visit left child, then right child, then the node
- All depth first searches are easy to implement with recursion

# Depth First - Preorder



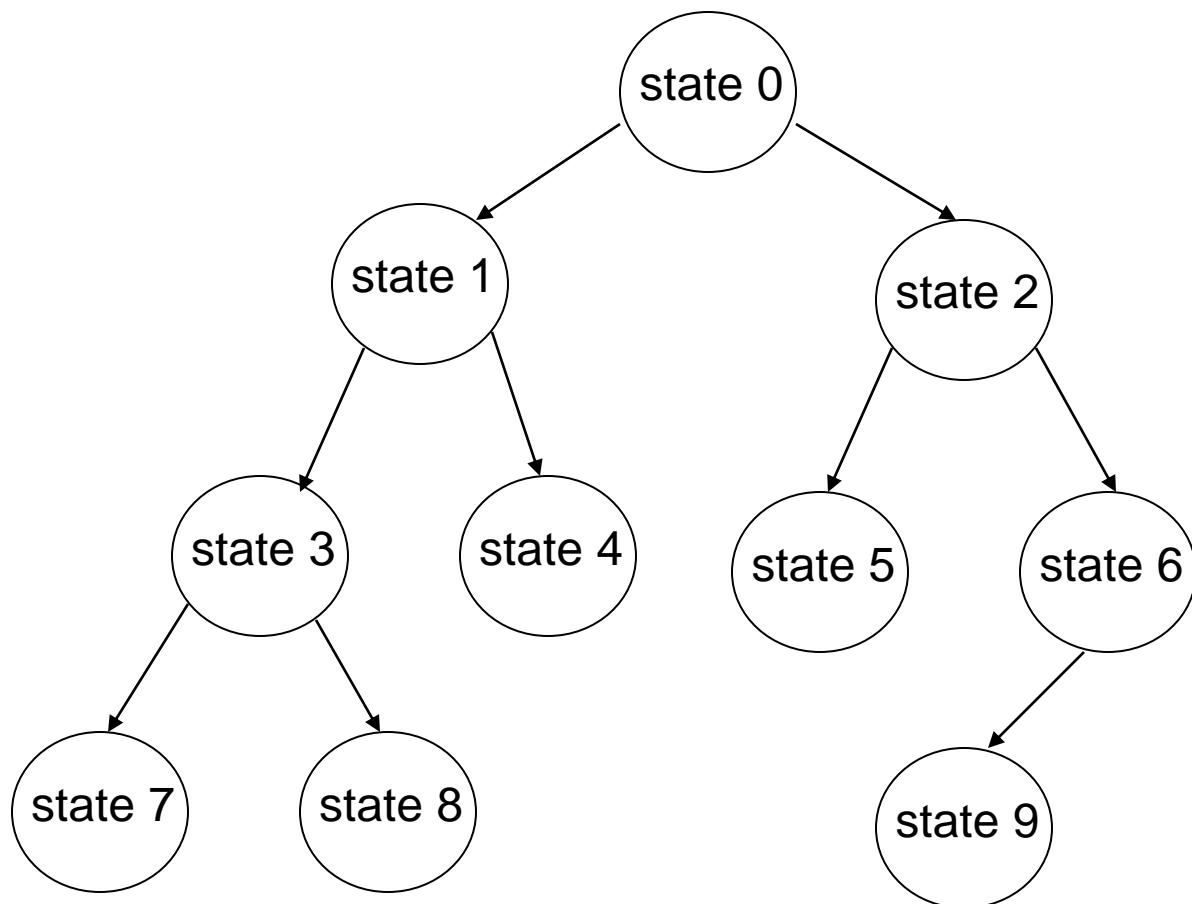
## Search Order

state 0  
state 1  
state 3  
state 7  
state 8  
state 4  
state 2  
state 5  
state 6  
state 7

# Depth First - Preorder

```
public void printTree-Preorder() {  
    if(root == null) { System.out.println("Empty tree"); }  
    else { printTree-Preorder(root); }  
}  
  
private void printTree-Preorder(Node node) {  
    if(node == null)  
        return;  
    System.out.println(node.data.toString());  
    printTree-Preorder(node.left);  
    printTree-Preorder(node.right);  
}
```

# Depth First - Inorder



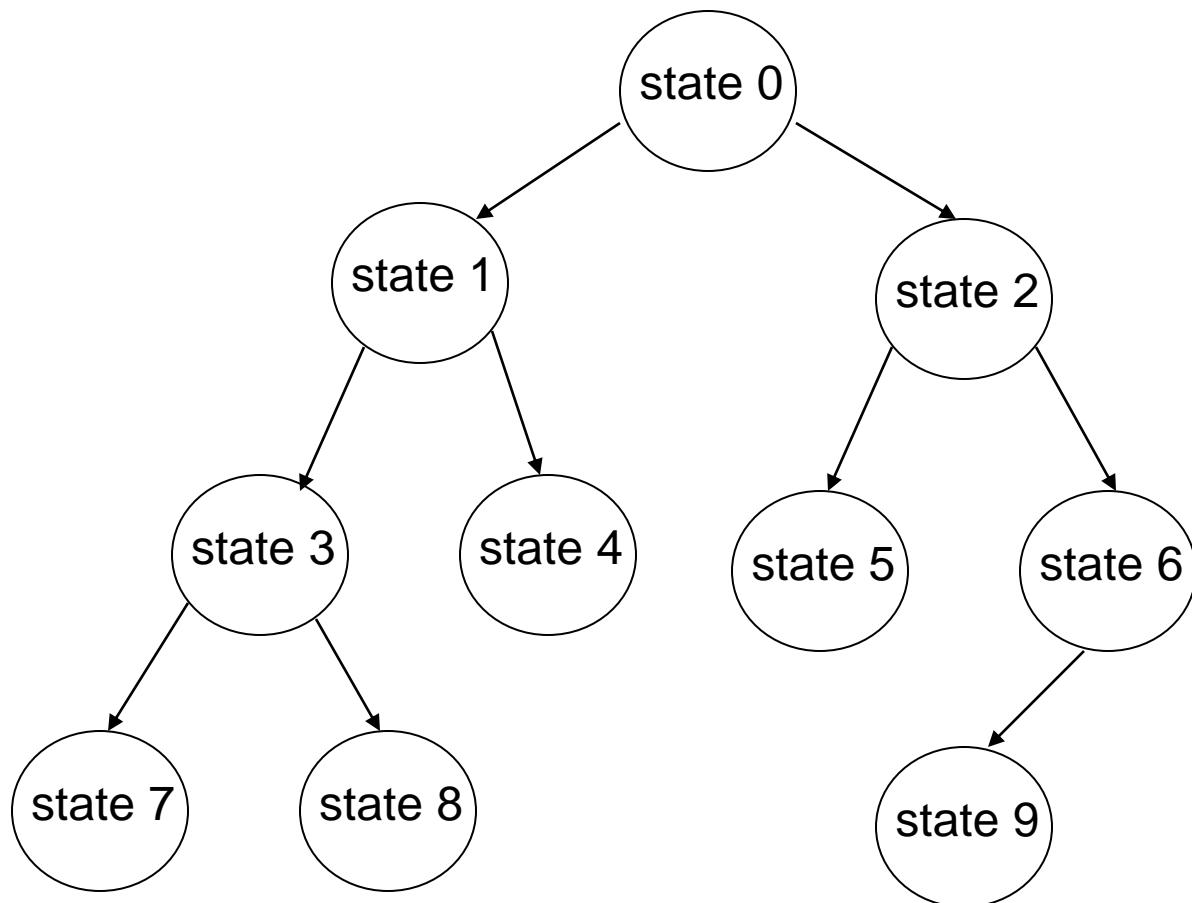
## Search Order

state 7  
state 3  
state 8  
state 1  
state 4  
state 0  
state 5  
state 2  
state 9  
state 6

# Depth First - Inorder

```
public void printTree-Inorder() {  
    if(root == null) { System.out.println("Empty tree"); }  
    else { printTree-Inorder(root); }  
}  
  
private void printTree-Inorder(Node node) {  
    if(node == null)  
        return;  
    printTree-Inorder(node.left);  
    System.out.println(node.data.toString());  
    printTree-Inorder(node.right);  
}
```

# Depth First – Postorder



## Search Order

state 7  
state 8  
state 3  
state 4  
state 1  
state 5  
state 9  
state 6  
state 2  
state 0

# Depth First - Postorder

```
public void printTree-Postorder() {  
    if(root == null) { System.out.println("Empty tree"); }  
    else { printTree-Postorder(root); }  
}
```

```
private void printTree-Postorder(Node node) {  
    if(node == null)  
        return;  
    printTree-Postorder(node.left);  
    printTree-Postorder(node.right);  
    System.out.println(node.data.toString());  
}
```

# Depth First Search

- Advantages
  - requires much less memory than breadth first
    - $M_{\text{needed}} = \text{level}$
- Disadvantage
  - may never find the solution
    - some search spaces have an infinite number of states (or very nearly infinite)
    - this means a single “branch” is infinite
    - we’ll never search other branches

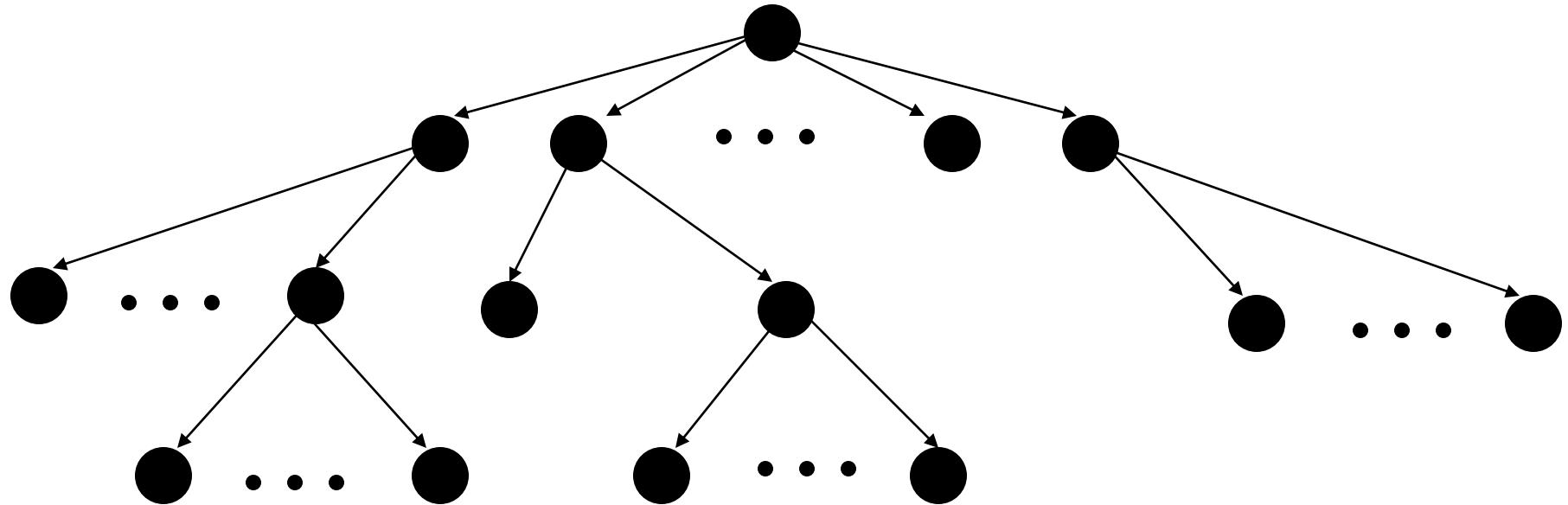
# Application of Tree Traversal

- We've already shown how tree traversal can be used to print out all of the nodes
  - this is good for debugging, but not needed for much else
- Consider a computer chess game
  - each node represents a state of the game
    - where each piece is on the board
  - for the computer to decide it's next move, it would like to pick a node that will lead to the best state

# Chess Game

- We'll say a node contains the following
  - the position of each piece on the board
  - a value indicating how favorable the current positions are
    - high value means it's good
      - check mating opponent would be the highest value
    - a low value means it's bad
      - being in check mate will be the lowest value
  - each node will have from 0 to 16 children
    - why?

# Chess Game



There are millions of more states but they won't all fit on the slide.



# Chess Game

- Consider the first possible move
  - can move any one of 8 pawns or 2 knights
    - that means we have 10 successor states
  - opponent will then have 10 possible moves
  - obviously, the number of states available at just the second level is immense
    - the overall search space is virtually infinite
- Two possible solutions
  - use a breadth first search and only go to a certain level
    - possibly monumental memory requirements
  - use a depth first search but only go so far along each branch
    - limits the memory requirements

# Lecture 12.1

## Handling Exceptions

CS112, semester 2, 2007

1

### What to do when an exception happens?

- Not handle the exception. Allow the program to die or core dump.
- Issue a warning. Print out some type of error message, probably at the spot in the code where the exception occurred and then exit.
- Handle the exception gracefully and continue executing.

CS112, semester 2, 2007

3

### What is an Exception?

- Exceptions are errors or anomalies that occur during the **execution** of a program.
- They can be the result of unavailability of system resources, such as memory, file space, channels or from data dependent conditions such as a divide by zero, or numerical overflow.
- Exceptions tend to be rare events but are predictable.

CS112, semester 2, 2007

2

### What to do when an exception happens? (cont)

- Do nothing?
  - Not acceptable if you want to remain employed or pass your courses.
- Print an error message?
  - Still not ideal.
  - Most real world programs need to be more robust than this.
- Exceptions need to be handled and corrected. Execution must continue.

CS112, semester 2, 2007

4

## How to go about this?

- Suppose an exception occurs in when dividing by 0.
  - Should the function or method that attempted the division be the one to handle it?
  - Can it?
  - Probably some other, higher level, section of code will have the information necessary to decide how to handle the exception.
  - Maybe different programs using the same classes and methods will handle exceptions differently.

CS112, semester 2, 2007

5

## Handling Exceptions

- The code that raises exceptions needs to be developed separately from code that handles them.
- If we pass exceptions up to calling routines, it is necessary to have a way to bundle information and for the exception to have some methods to assist in its handling.

CS112, semester 2, 2007

6

## Exception Basics

- The section of code that causes or detects a run-time abnormality (divide by zero, out of memory) will "throw" an exception.
- The exception is the object that is thrown. It may be a simple object such as an int, or a class object, including programmer defined class objects.

CS112, semester 2, 2007

7

## Exception Basics (cont)

- The exception is "caught" by another section of code.
- The exception object, itself, is used to convey information from the section of code that throws the object to the section of code that catches the object.

CS112, semester 2, 2007

8

## Exception Basics (cont2)

- Separation of exception creation and exception handling is very significant.
- Higher level sections of code can better handle exceptions in many cases.
  - If an exception in a library routine. That routine cannot know how to respond in a way that is appropriate for your program.
    - Appropriate response might be to terminate the program
    - Appropriate response might be a warning message
    - maybe the exception can be caught and disregarded.

CS112, semester 2, 2007

9

## Example 1

```
int main()
{
    int x = 5;
    int y = 0;
    int result;
    int exceptionCode = 25;
    try {
        if (y == 0) { throw exceptionCode; }
        result = x/y;
    }
    catch (int e) {
        if (e == 25) { cout << "Divide by zero" << endl; }
        else {cout << "Exception of unknown type" ; }
    }
    cout << "Goodbye" << endl; return 0;
}
```

CS112, semester 2, 2007

11

## Exceptions Syntax

- Sections of code that can “throw” exceptions are surrounded in “try blocks”.
- Exceptions thrown from within try blocks are “caught” by a “catch clause”.

CS112, semester 2, 2007

10

## Example 2

```
int main()
{
    int x = 5;
    int y = 0;
    int result;
    char * err_msg = "Division by Zero";
    try {
        if (y == 0) { throw err_msg}
        result = x/y;
    }
    catch (char *e) {
        cout << e << endl;
    }
    cout << "Goodbye" << endl; return 0;
}
```

CS112, semester 2, 2007

12

## Exception Handling with classes

- In C++ an exception is usually an object
- This allows more information to be bundled into the exception
- Also allows the exception to contain methods to assist the sections of code that handle or process the exception.

CS112, semester 2, 2007

13

## Example 1

```
class DivideByZero {  
public:  
    DivideByZero (int n, int d) { num = n;  
                           denom = d ;  
                           message ="Divide by zero"; }  
    int getNumerator() {return num;}  
    int getDenominator() {return denom;}  
    string getMessage() {return message;}  
private:  
    int num;  
    int denom;  
    string message;  
};
```

CS112, semester 2, 2007

14

## Example 1 (cont)

```
int main()  
{  
    int x = 5;  
    int y = 0;  
    int result;  
    try {  
        if (y == 0) { throw DivideByZero ( x, y ) ; }  
        result = x/y;  
    }  
    catch ( DivideByZero e ) {  
        cout << e.getMessage() << endl;  
        cout << "Numerator: " << e.getNumerator() ;  
        cout << "Denominator: " << e.getDenominator();}  
        cout << "Goodbye" << endl;  
    return 0;}  
}
```

CS112, semester 2, 2007

15

# Lecture 14.1

## Revision I

# Arrays

---

- What would the following segment of C++ code print?

```
#define MAXI 50
#define MAXJ 75
int i , j ;
float values[ MAXI ][ MAXJ ];
for (i = 0; i < MAXI; i++) {
    for (j = 0; j < MAXJ; j++) {
        values[ i ][ j ] = i+j; } }
cout<<values[i-1][j-1]<<endl;
```

# Answer

---

■ 123

# Pointer

---

- What would the following segment of C++ code print?

```
#define MAXI 50
#define MAXJ 75
int i , j ;
float *ptr;
float values[ MAXI ][ MAXJ ];
for (i = 0; i < MAXI; i++) {
    for (j = 0; j < MAXJ; j++) {
        values[ i ][ j ] = i+j; } }
ptr = &values[i-1][j-1];
cout<<*ptr<<endl;
```

# Answer

---

■ 123

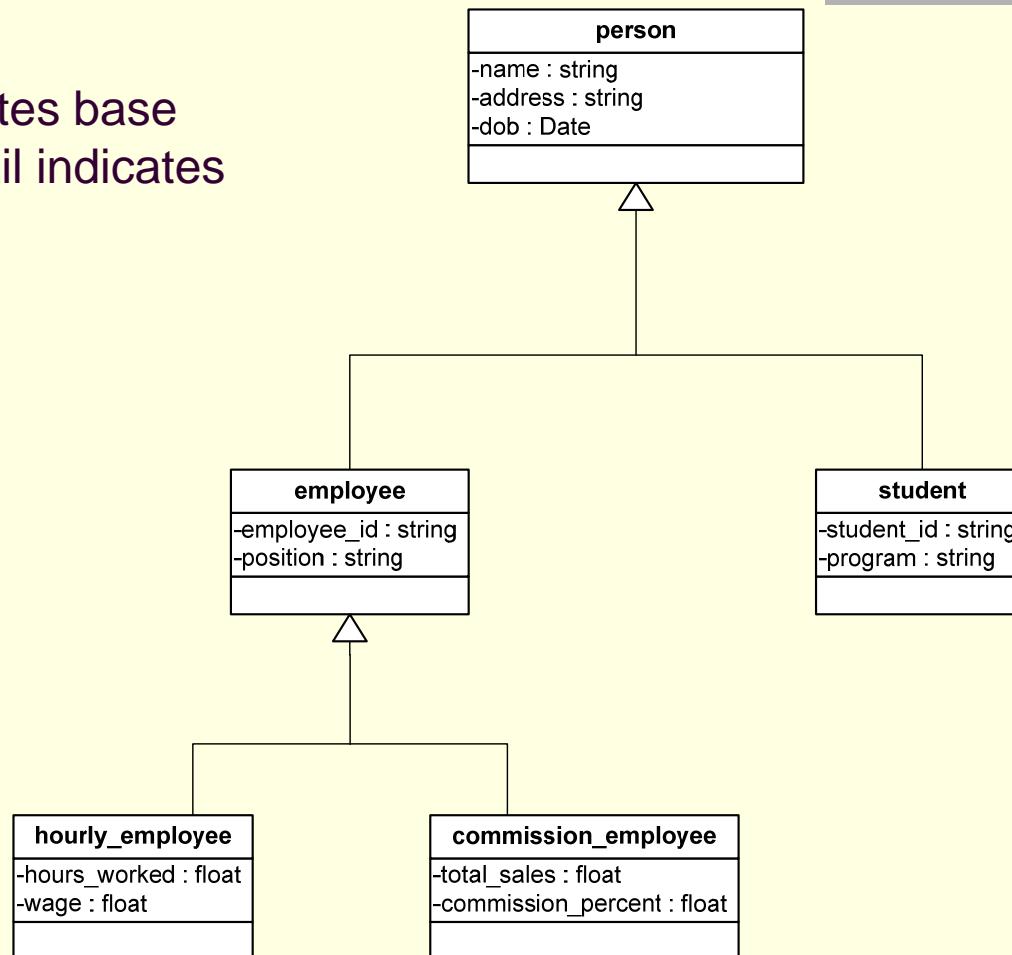
# Classes

---

- What is the difference between public and protected members?
- A hierarchical relationship can be made using inheritance relationships.  
An example is shown in the next slide where hourly\_employee class and commission employee class inherit from employee class. employee and student class inherit from person class.

# Inheritance representation

Arrow head indicates base class and arrow tail indicates derived class.



# Question

---

- Using the example shown in previous slide write a code in C++ for all the classes related by inheritance relationships.

# Answer: person class

---

```
class person{
    public:
        person();
        ...
    private:
        string name;
        string address;
        string dob;
};
```

# employee and student class

---

```
class employee: public person{
    public:
        employee();
        ...
    private:
        string employee_id
        string position;
};

class student: public person{
    public:
        student();
        ...
    private:
        string student_id;
        string program;
};
```

# hourly\_employee and commission\_employee classes

---

```
Class hourly_employee: public employee{  
    public:  
        hourly_employee();  
        ...
```

```
    private:  
        float hours_worked;  
        float wage;  
};
```

```
Class commission_employee: public employee{  
    public:  
        commission_employee();  
        ...
```

```
    private:  
        float total_sales;  
        float commission_percent;  
};
```

# Growth rate (time complexity)

---

- Find the Big O of the following expression:

$$f(n) = 6n^2 + 20n + 100$$

# Answer

---

Ans:  $O(n^2)$

# Lecture 14.2

## Revision II

# Linked list

---

Convert the following struct to templated struct so that any type of data can be stored in a node and a linked list can be made by linking the given nodes.

```
struct node{  
    int data;  
    node * next;  
    node * prev;  
    node * top;  
    node * bottom;  
};
```

# Answer

---

```
template <class T>
struct node{
    T data;
    node <T>* next;
    node <T>* prev;
    node <T>* top;
    node <T>* bottom;
};
```

# Stack

---

- Using the following definition of stack class, write a method to empty the whole stack.

```
class stack{
    public:
        ...
        int top(); //return top element
        bool pop(); //returns true if element removed
        bool push(int); //inserts data on the top of a stack
        bool isEmpty(); //returns true if stack is empty
}
```

# Answer

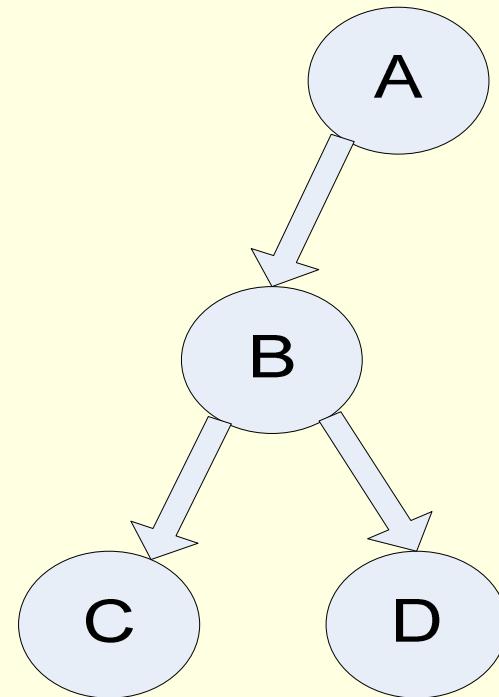
---

```
Void stack::empty(){
    While(!isEmpty()){
        this->pop();
    }
}
```

# Binary Tree

---

- Write the pseudocode for preorder traverse and then print the following binary tree using preorder traverse.



# Answer:

---

- ABCD