Data Structure

Dr. Sawsan M. Mahmoud
Computer & Software Engineering
College Engineering
Al- Mustansiriyah University
2014-2015

Contact

- **Name:** Sawsan M. Mandalawi
- **Email:** sawsan.mandalawi@yahoo.com
- **Room:** Room 6 in the Department of Computer & Software Engineering
- **Lecture Time:** Tuesday 11:30 - 02:30
  Thursday 08:30 - 11:30
- **Notes:** Other office hours are available by an appointment. The contents of this syllabus is not to be changed during the current semester.

What is this Module all about?

- This course introduces the basic data structures used in computer software
    - Understand the data structures
    - Analyze the algorithms that use them
    - Know when to apply them

  Also, practice design and analysis of data structures and practice using these data structures by writing programs.
- On successful completion of this course, you will understand:
    - What the tools are for storing and processing common data types
    - Which tools are appropriate

  So that you will be able to make good design choices as a developer, project manager, or system customer

Module Content

- Introduction to data structure and C++ programming language
- Declaration, Comments, Numbers, Variables, I/O data (cout, cin).
- Expressions and Assignments, Increment and Decrement Operator, Combined Operator.
- Selection and Switch, If statement Nested If, If else, Switch
- Repetition statement, While, Do While For Nested For statement.
- One dimensional array.
- Two dimensional array.
- Stack (Insertion, Deletion).
- Evaluation and Translation of Expression.
- Queue, Circular Queue (Insertion, Deletion).
- Structure
- Linked List, Linear Structure (Insertion).

Sources of Information

- Book
  - James F. Korsh,Leonard J.Garrett (2008), **Data Structures, Algorithms, and Program style using(c), Textbook.**
- Suggested references
  - Clifford A. Shaffer, Virginia Tech (2011), **Data Structures and Algorithm Analysis.**
  - Michael T. Goodrich, Roberto Tamassia. David M. Mount (2007), **Data Structures and Algorithms in C++ .**

Assessment Software Techniques

- **Assessment of Learning:** The student will be evaluated based on homework, exams, and class participation.
- **Examination Information:** The exams will cover material that has been discussed in class or covered in the book.
- **Assignments:** Homework must be submitted at the beginning of class on the date which it is due. Late homework will be accepted only when special circumstances are approved by the instructor.
- **Terms examination: 30%**
  - Term 1
  - Half Term
  - Term 2
- **Final Exam: 70%**

Lecture one
Introduction to Data Structure and C++ Programming Language

The Task of Programming

- **Programming:** writing instructions that enable a computer to carry out tasks
- **Programs** are frequently called applications
- Learning a computer programming language requires learning both vocabulary and syntax
- The rules of any language make up its syntax
- Types of errors:
  - Syntax errors
  - Logical errors
    - Semantic errors

The Task of Programming

- **Machine language:** language that computers can understand; it consists of 1s and 0s
- **Interpreter:** program that translates programming language instructions one line at a time
- **Compiler:** translates entire program at one time
- **Run** a program by issuing a command to execute the program statements
- **Test** a program by using sample data to determine whether the program results are correct

Data Structures: What?

- Data Structures can be defined as: An organization of information, usually in memory, for better algorithm efficiency, such as stack, queue, linked list, dictionary, and tree.
- Need to organize program data according to problem being solved
- Abstract Data Type (ADT) - A data object and a set of operations for manipulating it
  - List ADT with operations **insert** and **delete**
  - Stack ADT with operations **push** and **pop**

Data Structures: Why?

- Program design depends crucially on how data is structured for use by the program
  - Implementation of some operations may become easier or harder
  - Speed of program may dramatically decrease or increase
  - Memory used may increase or decrease
  - Debugging may be become easier or harder

Selection of Data Structures

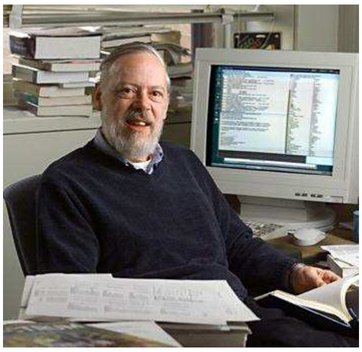
- The volume of data
- Speed and method of use of the data
- The dynamic nature of the data, such as changed and adjusted periodically
- The required storage capacitance
- Time required to retrieve any information from the data structure
- Method of programming

Data and Information

- Data is a collection of facts, figures and statistics related to an object. Data can be processed to create useful information.
- Example: Students fill an admission form when they get admission in college.
- The manipulated and processed form of data is called information. Data is used as input for processing and information is output of this processing.
- Example: Data collected from census is used to generate different type of information. Governments can use the information in important decision

What is C?

- Programming languages provide the appropriate formulas to define and use primitive data types.
- C is a rather old programming language (1972, Richie, Bell Labs)
- Originally designed as a systems software platform (OS and the like)
- Procedural, block oriented language (no object-oriented programming)



Dennis Ritchie
Inventor of the
C Programming Language

Why learn C++?

- Small, extensible language. Progenitor of many languages.
- Many applications and much support due to its age and general use
- Many tools written to support C development.
- Close to the hardware, programmer manages memory
- Common embedded systems language
- Can be fast, efficient (most cited reason)

Lecture Two
Declaration, Comments, Numbers, Variables, I/O data (cout, cin).

hello world

```
#include <iostream.h>

/*
hi mom
*/

  int main () {
  cout<<“hello world\n”;
}
```

#include statement

hello world

```
#include <iostream.h>

/*
hi mom
*/

int main () {
  cout<<"hello world\n";
}
```

→ **#include statement**

comments

hello world

```
#include <iostream.h>

/*
hi mom
*/

  int main () {
  cout<<"hello world\n";
}
```

→ **#include statement**

comments

Main program

#include statement

- Lines that begin with # sign,
- A preprocessor directive is an instruction to the preprocessor. Named file inclusion is concerned with adding the content of a header file to a source program file. Standard header files. For example,

  #include <iostream.h>
- #include causes a header file to be copied into the code.
- programmer-defined header file surrounded by double quotation marks. #include “header1.h” to advantage in partitioning large programs into several files.

Monday, December 01, 2014 Data Structure 5

Comments

- Standard C++ comments are bracketed between the symbols /* and the symbols */
- Anything in between (spacing, tabs, newlines, punctuation, code, etc.) is ignored

```
/*
hi mom
*/
```

“hi mom” is a comment

Monday, December 01, 2014 Data Structure 6

the main function

- Every C program that can be run must have a main function.
- When the system starts the executable, it runs that function by default

```
int main () {

        /* your code goes here */

}
```


The { } symbols

- A set of statements grouped together called a block.
- A “block” is indicated here by the beginning and end of the { } symbols

;

- The ; character is used to indicate the end “of a statement”
- The end of a statement is not necessarily the end of a line. Statement is logical, line is layout

hello world

```
#include <iostream.h>

/*
hi mom
*/

int main () {
  cout<<“hello world\n”;
}
```

No semicolon here (#include line)

Don’t care whether there are semicolons or not (comment)

Main program

Each statement should be terminated with a semicolon unless it defines a block of statements {…}

cout

- cout is the function that takes a string as input and prints it as indicated
- Strings are listed between " ".
- Remember that to get a newline, you must explicitly include \n
  - Example:

```
cout<<"hello world\n";
```


Escape characters

- Characters that are hard to express:
  - \n newline
  - \t tab
  - \' print a single quote
  - \\ print a backslash
  - many others

#include: < > versus “ “

```
#include <iostream.h>
#include "simpleCalc.h”
```

- < > means get the built-in variable/function definitions from “the standard place” (usually include)
- “ “ means get them from the current directory, i.e, your own variable/function definitions

simplecalc.h (Header file)

```
double
 INSTALLATION_FEE = 230.00, // installation fee
 COST_PER_FOOT = 3.25; // pipe cost per foot
```

Single-line comment: //

- in C++ you can use // as a comment
- Ignore everything from // to the end of the line

```
double
  INSTALLATION_FEE = 230.00;    // installation fee
```

This is the comment part

Why do we need variable declaration? Answer: memory

- Here is how C++ deals with memory
  - Imagine the system memory as a nice, flat stretch of beach
  - You want a variable, you need to dig a hole in the sand and dump the value in
  - How big a hole?

Declare variable before use

- Declare a variable to be of type integer, the compiler allocates a memory location for that variable. The size of this memory location depends on the type of the compiler.
- When you declare a variable, you are telling the compiler the size of value the variable may hold (its type)
- You cannot change the type of value a variable can hold once declared.

Must declare before use

- Every variable must be declared before it can be used (its type must be indicated)
- Syntax:

  <variable_type> <variable_name> [=<initial_value>];
- Example:

  int length, width = 5, height = 10;

Common types, “regular” C

- int : an integer, usually 4 bytes
- float: float, usually 4 bytes
- double : float, usually 8 bytes
- char : single char, value in single quotes

Basic Types

Type (16 bit)	Smallest Value	Largest Value
short int	-32,768(-2^{15})	32,767(2^{15}-1)
unsigned short int	0	65,535(2^{16}-1)
Int	-32,768	32,767
unsigned int	0	65,535
long int	-2,147,483,648(-2^{31})	2,147,483,648(2^{31}-1)
unsigned long int	0	4,294,967,295

Basic Types

Type (32 bit)s	Smallest Value	Largest Value
short int	-32,768(-2^{15})	32,767(2^{15}-1)
unsigned short int	0	65,535(2^{16}-1)
Int	-2,147,483,648(-2^{31})	2,147,483,648(2^{31}-1)
unsigned int	0	4,294,967,295
long int	-2,147,483,648(-2^{31})	2,147,483,648(2^{31}-1)
unsigned long int	0	4,294,967,295

Rules for Variable Names

1. Must begin with a letter or an underscore.; any combination of letters, digits and underscore: A to Z , a to z , 0 to 9 , and the underscore "_"
2. Only the first 32 characters as significant.
3. There can be no embedded blanks.
4. Reserved words cannot be used as identifiers.
5. Identifiers are case sensitive.
   - E.g., int x; int weight,height;

Constants

- Constant is an identifier whose value is fixed and does not change during the execution of a program in which it appears. const double CITY_TAX_RATE = 0.0175;
- In C++ the declaration of a named constant begins with the keyword const.
- During execution, the processor replaces every occurrence of the named constant .

Constants

- fixed values CITY_TAX_RATE = 0.0175 is an example of a constant.
- Integer Constants: commas are not allowed in integer constants.
- Floating-Point Constants: either in conventional or scientific notation. For example, 20.35; 0.2035E+2
- Character Constants and Escape Sequences: a character enclosed in single quotation marks. Precede the single quotation mark by a backslash,

```
cout<<"%c", '\";
```

- Escape sequence: causes a new line during printing. \n

Strings

- A string is a sequence of characters that is treated as a single data item. A string variable is a variable that stores a string constant.
- characters surrounded by double quotation marks.
- format specified for output converts the internal representation of data to readable characters.(%f) for example, City tax is 450.000000 dollars.
- backslash character can be used as a continuation character:

```
cout<< "THIS PROGRAM COMPUTES \ CITY INCOME TAX";
```


Strings

- how to declare string variables.
  - Begin the declaration with the keyword char,
    Char report_header [41]
  - To initialize a string variable at complie time,
    char report_header [41] = "Annual Report"

Statements

- A statement is a specification of an action to be taken by the computer as the program executes.
- Compound Statements: is a list of statements enclosed in braces, { }

C++ Keywords

- Keywords are predefined reserved identifiers that have special meanings.
- They cannot be used as identifiers in your program.
- C reserved words must be typed fully in lowercase.
- The reserved words of C++ may be conveniently placed into several groups.
- In the first group we put those that were also present in the C programming language and have been carried over into C++. There are 32 of these, and here they are:

C++ Keywords

auto const double float int
short struct unsigned break continue
else for long signed switch
void case default enum goto
register sizeof typedef volatile char
do extern if return static
union while

C++ Keywords

- There are another 30 reserved words that were not in C, are therefore new to C++, and here they are:

  asm dynamic_cast namespace
  reinterpret_cast try bool explicit
  new static_ cast typeid catch
  false operator template typename
  class friend private this using
  const_cast inline public throw virtual
  delete mutable protected true wchar_t

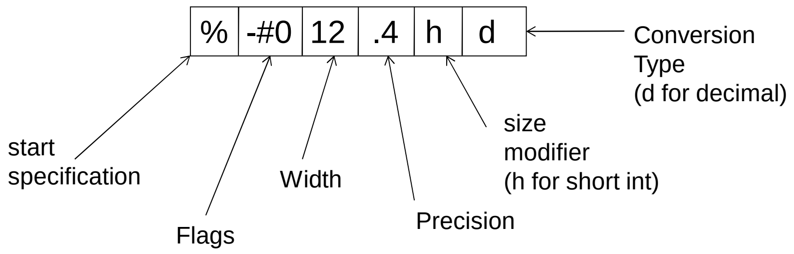
cout, more detail

Many descriptors

- %s string
- %d decimal
- %e floating point exponent
- %f floating point decimal
- %u unsigned integer
- and others

Full Format string

- The format string contains a set of format descriptors that describe how an object is to be printed



Examples

- cout<<"%f\n",M_PI;

3.141593

- cout<<"%.4f\n",M_PI;

3.1416 (4 decimal points of precision, with rounding)

- cout<<"%10.2f\n",M_PI;

  3.14 (10 spaces in total including the number and the decimal point)

- cout<<"%10.2f is PI\n",M_PI;

3.14   is PI

Input

cin

```
cout<<"Please enter the yards of pipe used: ";
cin>>"%f",&yardsOfPipe;
```

- cin is an input routine
  - useful for reading in string input and doing conversion to the correct type, all at once
  - syntax is "kind of like" cout
  - beware the use of the & operator!!!

Basic form

- To understand input, it is probably better to start with an example.

```
Cin>>“%d, %f”, &myInt, &myFloat;
```

- is waiting for input of the exact form

25, 3.14159

Arithmetic Expression

Types determine results

- For integers: +,-,*,/ all yield integers. Thus division can lead to truncation (2/3 has value 0). % gives the remainder
- For floats: +,-,*,/ all work as advertised. No remainder.

Mixed computation

- As with most languages, C++ expects to work with like types. 1 + 1, 3.14. + 4.56
- When mixing, usually errors except where C++ can "help"
- It will promote a value to a more "detailed" type when required
- 1 + 3.14 yields a float (1 promoted to 1.0)

coercion, cast

- Explicit type conversion:
  - (double) 3
  - convert int 3 to double 3.0. Note the parens!
  - (int) 3.14
  - convert 3.14 to int. No rounding!
  - Makes a new value, does not affect the old one!

Example

```
#include <stdio.h>

int main(){
  // const means the variable value cannot be changed from this initial setting
  const int A = 3, B = 4, C = 7;
  const double X = 6.5, Y = 3.5;

  printf("\n*** Integer computations ***\n\n");

  printf("%d + %d equals %d\n",A,B,A+B);
  printf("%d - %d equals %d\n",A,B,A-B);
  printf("%d * %d equals %d\n",A,B,A*B);
  printf("%d / %d equals %d with remainder %d\n",A,B,A/B,A%B);
  printf("\n");

  printf("\n*** Real computations ***\n\n");
  printf("%f + %f equals %f\n",X,Y,X+Y);
  printf("%f - %f equals %f\n",X,Y,X-Y);
  printf("%f * %f equals %f\n",X,Y,X*Y);
  printf("%f / %f equals %f\n",X,Y,X/Y);
```

Example

```
printf("\n*** Mixed-type computations ***\n\n");
printf("%f + %d equals %f\n",X,A,X+A);
printf("%f - %d equals %f\n",X,A,X-A);
printf("%f * %d equals %f\n",X,A,X*A);
printf("%f / %d equals %f\n",X,A,X/A);

printf("\n*** Compound computations ***\n\n");
printf("%d + %d / %d equals %d\n",C,B,A,C+B/A);
printf("(%d + %d) / %d equals %d\n",C,B,A,(C+B)/A);
printf("\n");
printf("%d / %d * %d equals %d\n",C,B,A,C/B*A);
printf("%d / (%d * %d) equals %d\n",C,B,A,C/(B*A));

printf("\n*** Type conversions ***\n\n");
printf("Value of A: %d\n",A);
printf("Value of (double)A: %f\n",(double)A);
printf("Value of X: %f\n",X);
printf("Value of (int)X: %d\n",(int)X);

return 0;
}
```


Assignment

- the “=“ means assignment, not equalty.
- Assignment means:
  - do everything on the lhs of the =, get a value
  - dump the value into the memory indicated by the variable on the rhs
  - variable now associated with a value
- declare first, assign second!

example

```
int val1, val2;
val1 = 7 * 2 + 5;    // 19 now in val1
val2 = val1 + 5;     // 24 in val2
val2 = val2 + 10;    // calc lhs (34),
                     // reassign to val2
```


rules

- rhs must yield a value to be assigned
- lhs must be a legal name
- type of the value and the variable must either match or there be a way for a conversion to take place automatically

Example

```
 // Addition program
#include <iostream.h>

int main()
{

    int integer1, integer2, sum; /*declaration */
    cout<<"Enter first integer\n";  /* prompt */
    sin>>"%d", &integer1;           /* read an
    cout<<"Enter second integer\n"; /* prompt */
    cin>>"%d", &integer2;           /* read an
    sum = integer1 + integer2;          /*
    cout<<"Sum is %d\n", sum ;       /* print sum

    return 0;  /* indicate that program ended
}
```


Example

```
#include <stdio.h>
 int main (void)
{
     double number;
     cout<<"Enter a number : ";
     cin<<"%lf" , &number;
     Inverse = 1.0 / number ;
     cout<<"Inverse of %f is %f" << number, inverse;
}
```

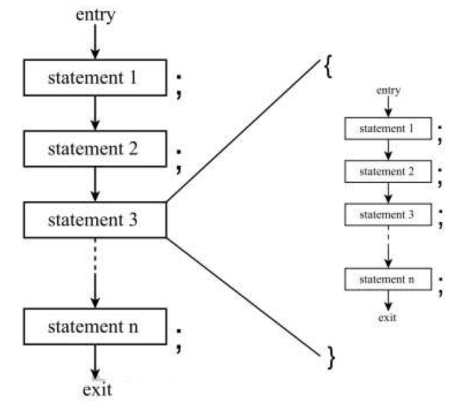
Lecture Three
Flow Control and Booleans

Logical Expressions

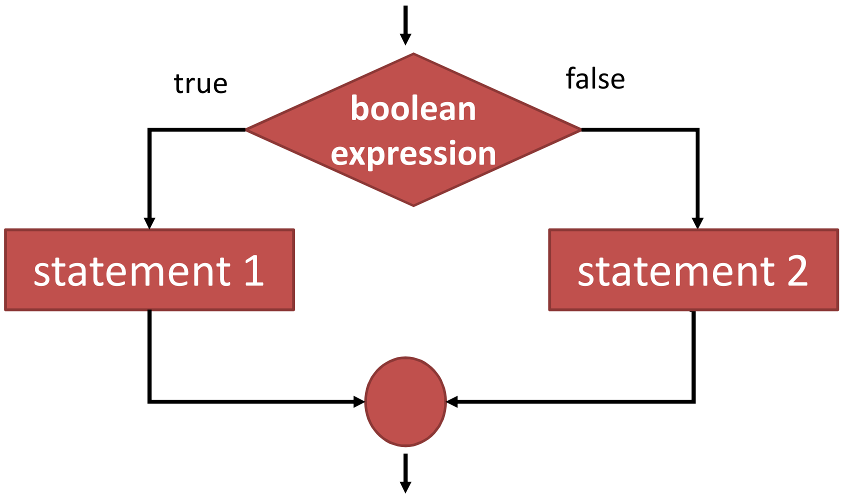
- Relational and logical operators – result is boolean-valued
  - == equal to `counter == 0`
  - != not equal to `counter != 0`
  - > greater than `counter > 0`
  - < less than `counter < 0`
  - >= greater than or equal to `counter >= 0`
  - <= less than or equal to `counter <= 0`
  - && logical and `0 < i && i < 10`
  - | | logical or `i <= 0 || i >= 10`
  - ! logical not `! done`

Flow of Control Statements

- Sequential Statements: all statements are executed once, one after another



Selective Execution



Structure of an if statement

```
if(expression1)
   statement1;
else if(expression2)        /* Optional */
   statement2;
else                        /* Optional */
   statement3;
```

The expressions are *boolean expressions* that resolve to a true or a false.

If expression1 is true, execute statement1.

Otherwise, test to see if expression2 is true. If so, execute statement2.

Otherwise, execute statement3.

if statements

```
if(age > 39)
   cout<<"You are so
old!\n";
```

The if statement

Fundamental means of *flow control*

How we will make decisions

Boolean expressions

The actual determination of the decision

```
age > 39
c == 0
l <= 0
(age >= 18) && (age
< 65)
```

Important: The test for equality is ==, not =. This is the most common error in a C program.

Example if statements

```
if(age < 18)
   cout<<"Too young to vote!\n";
```

```
if(area == 0)
   cout<< "The plot is empty\n";
else
   cout<< "The plot has an area of %.1f\n", area;
```

```
if(val < 0)
   cout<< "Negative input is not allowed\n";
else if(val == 0) ;
   cout<< "A value of zero is not allowed\n";
else
   cout<< "The reciprocal is %.2f\n", 1.0 / val;
```

Note the indentation

Example if statements

Note the indentation

```
if (n % 2)
  cout<<"n is odd\n";
else
  cout<<"n is even\n";
```

```
if (strcmp(s1, s2))
  cout<<"s1&s2 differ\n";
else
  cout<<"s1&s2 are
  equal\n";
```

Blocks

Single Statement

```
cout<<"This is statement\n";
```

Block

```
{
    "All items in a curly brace\n";
    cout<< "as if there are one statement";
    cout<< "They are executed sequentially";
}
```


Where is this useful?

```
if(value > 0)
{
    result = 1.0 / value;
    cout<<"Result = %f\n", result;
}
```

If the expression is true, all of the statements in the block are executed

Where is this useful?

```
if(value > 0)
{
     result = 1.0 / value;
     cout<<"Result = %f\n", result;
}

if(value > 0)
     result = 1.0 / value;
     cout<<"Result = %f\n", result;
```

Will these two sections of code work differently?

Where is this useful?

Yes!

```
if(value > 0)
{
     result = 1.0 / value;
     cout<<"Result = %f\n", result;
}

if(value > 0)
     result = 1.0 / value;
     cout<<"Result = %f\n", result;
```

Will always execute!

Nested Blocks

```
    if(bobsAge != suesAge)  /* != means "not equal" */
    {
        cout<<"Bob and Sue are different ages\n”;
        if(bobsAge > suesAge)
        {
            cout<<"In fact, Bob is older than Sue\n”;
            if((bobsAge - 20) > suesAge)
            {
                cout<<"Wow, Bob is more than 20 years
older\n”;
            }
        }
    }
```

What does this do?

Saturday, December 20, 2014 Data Structure 13

Importance of indentation

```
    if(bobsAge != suesAge)  /* != means "not equal"
*/
    {
    cout<<"Bob and Sue are different ages\n”;
    if(bobsAge > suesAge)
    {
    cout<<"In fact, Bob is older than Sue\n”;
    if((bobsAge - 20) > suesAge)
    {
    cout<<“Wow, Bob is more than 20 years older\n”;
    }
    }
    }
```

See how much harder this is to read?

Saturday, December 20, 2014 Data Structure 14

Boolean Expressions

- An expression whose value is true or false
- In C:
  - integer value of 0 is “false”
  - nonzero integer value is “true”
- Example of Boolean expressions:
  - age < 40
  - graduation_year == 2010

Relational operator


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

int main()
{
    const bool trueVar = true, falseVar = false;
    const int int3 = 3, int8 = 8;

    cout<<"No 'boolean' output type\n”;
    cout<<"bool trueVar: %d\n",trueVar;
    cout<<"bool falseVar: %d\n\n",falseVar;
    cout<<"int int3: %d\n",int3);
    cout<<"int int8: %d\n",int8);
}
```

Library that defines: bool, true, false

What does the output look like?

// Example3 (continued…)

```
cout<<"\nint3 comparators\n”;
cout<<"int3 == int8: %d\n",(int3 == int8);
cout<<"int3 != int8: %d\n",(int3!=int8);
cout<<"int3 < 3:  %d\n",(int3 <  3);
cout<<"int3 <= 3: %d\n",(int3 <= 3);
cout<<"int3 > 3:  %d\n",(int3 >  3);
cout<<"int3 >= 3: %d\n",(int3 >= 3);
```

Comparing values of two integer constants

What does the output look like?

More Examples

- char myChar = ‘A’;
  - The value of myChar==‘Q’ is false (0)
- Be careful when using floating point equality comparisons, especially with zero, e.g. myFloat==0

Suppose?

- What if I want to know if a value is in a range?
- Test for: 100 ≤ L ≤ 1000?

You can't do…

This code is WRONG and will fail.

```
if(100 <= L <= 1000)
{
    cout<<"Value is in range…\n");
}
```

Why this fails…

C ++ Treats this code this way

```
if((100 <= L) <= 1000)
{
     cout<<"Value is in range…\n");
}
```

Suppose L is 5000. Then 100 <= L is true, so (100 <= L) evaluates to true, which, in C, is a 1. Then it tests 1 <= 1000, which also returns true, even though you expected a false.

Compound Expressions

- Want to check whether -3 <= B <= -1
  - Since B = -2, answer should be True (1)
- But in C++, the expression is evaluated as
  - ((-3 <= B) <= -1)   (<= is left associative)
  - (-3 <= B) is true (1)
  - (1 <= -1) is false (0)
  - Therefore, answer is 0!

Compound Expressions

- Solution (not in C): (-3<=B) and (B<=-1)

- In C: (-3<=B) && (B<=-1)
- Logical Operators
  - And: &&
  - Or: ||
  - Not: !

Compound Expressions

```
#include <iostream.h>

int main()
{
   const int A=2, B = -2;

   cout<<"Value of A is %d\n", A;
   cout<<"0 <= A <= 5?: Answer=%d\n", (0<=A) && (A<=5);

   cout<<"Value of B is %d\n", B;
   cout<<"-3 <= B <= -1?: Answer=%d\n", (-3<=B) && (B<=-1);
}
```

Compound Expressions

```
#include <iostream.h>

int main()
{
   const int A=2, B = -2;

   cout<<"Value of A is %d\n", A);
   cout<<"0 <= A <= 5?: Answer=%d\n", (0<=A) && (A<=5);

   cout<<"Value of B is %d\n", B);
   cout<<"-3 <= B <= -1?: Answer=%d\n", (-3<=B) && (B<=-1);
}
```

```
>./a.out
Value of A is 2
0 <= A <= 5?: Answer=1
Value of B is -2
-3 <= B <= -1?: Answer=1
```

Correct Answer!!!

Compound Expressions

```
#include <stdio.h>

int main()
{
   const int A=2, B = -2;

   cout<<"Value of A is %d\n", A;
   cout<<"0 <= A <= 5?: Answer=%d\n", (0<=A) && (A<=5);

   cout<<"Value of B is %d\n", B;
   cout<<"-3 <= B <= -1?: Answer=%d\n", (-3<=B) && (B<=-1);
}
```

```
>./a.out
Value of A is 2
0 <= A <= 5?: Answer=1
Value of B is -2
-3 <= B <= -1?: Answer=1
```

Correct Answer!!!

Truth Tables

p	q	Not !p	And p && q	Or p || q
True	True			
True	False			
False	True			
False	False			

Truth Tables

p	q	Not !p	And p && q	Or p || q
True	True	False		
True	False	False		
False	True	True		
False	False	True		

Truth Tables

p	q	Not !p	And p && q	Or p || q
True	True		True	
True	False		False	
False	True		False	
False	False		False	

Truth Tables

p	q	Not !p	And p && q	Or p || q
True	True			True
True	False			True
False	True			True
False	False			False

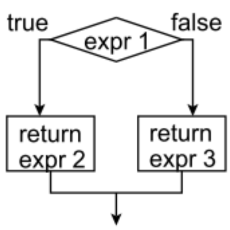
Truth Tables

Our comparison operators:
< <= == != >= >

p	q	Not !p	And p && q	Or p || q
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

Conditional Expressions

- Based on the Conditional Operator ?:
- `(expr 1)?(expr 2 :expr 3)`
  - If expr 1 is true, expr 2 is the value of the overall expression
  - If expr 1 is false, expr 3 is the value of the overall expression
  - Parentheses are not syntactically required
  - Typically used because ? has a high Precedence
- `max = (x > y) ? x : y;`
- `min = (x < y) ? x : y;`
- `index = (index+1 == size) ? 0 : ++index;`



Precedence & Associatively

Can you guess what's the answer?

```
int A=4, B=2;
Cout<<"Answer is %d\n",A+B>5&&(A=0)<1>1A+B-2;
```

Relational operators have precedence and associatively (just like arithmetic operators)

Use ( ) when in doubt


```
A = 4, B = 2;
A + B > 5 && (A = 0) < 1 > A + B - 2
```

(((A + B) > 5) && (((A=0) < 1) > ((A + B) – 2)))

((6 > 5) && (((A=0) < 1) > ((A + B) – 2)))

(1 && ((0 < 1) > ((A + B) – 2)))

(1 && (1 > (2 – 2)))

(1 && (1 > 0))

(1 && 1)

Answer: 1

Precedence: +/-
> <
&&

Associativity

“=“ is right associative

Example: X=Y=5

right associative: X = (Y=5)

expression Y=5 returns value 5: X = 5

You should refer to the C++ operator precedence and associative table

Or just use parentheses whenever you’re unsure about precedence and associativity

Operator	Description	Associativity
() [] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (change *type*) Dereference Address Determine size in bytes	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

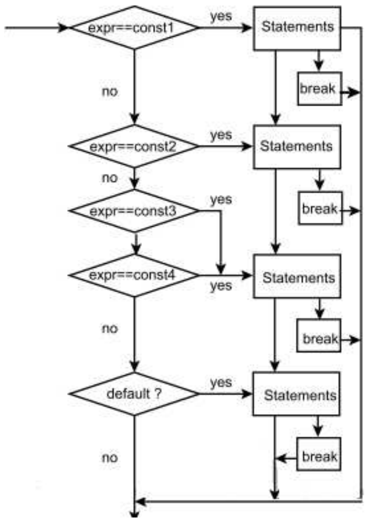
Switch Statement

- A less general substitute for the multi branch if. It is used for selecting among discrete values (int), i.e. not continuous values.

```
switch (int_expression)
{
    case_list:
            statement_list;
    case_list:
            statement_list;
    default:
            statement_list;
}
```


switch Example

```
/* counts characters, words, and
   lines in a file */
switch(c)
{ case '\n' : lines++;        /*
   fall through */
case '  '  :
case '\t' : inword = 0;
break;
default     : if (! inword) /*
   start word */
{ inword = 1;
words++;
}
break;
}
```



Behavior

- The int_expression is evaluated. If the value is in a case_list, execution begins at that statement_list and continues through subsequent statement_lists until: break, return, or end of switch.


```
#include <stdio.h>

void main()
{
   int gender;
   cout<<"Enter your gender (male=1, female=2): ";
   cin>>"%d",&gender;

   switch(gender)
   {
       case 1:
          cout<<"You are a male\n";
          break;
       case 2:
          cout<<"you are a female\n";
          break;
       default:
          cout<<"Not a valid input\n";
          break;
   }
}
```

Lecture Four
Loops and Repetition

Loops and Repetition

- Loops in programs allow us to repeat blocks of code.
- Useful for:
  - Trying again for correct input
  - Counting
  - Repetitive activities
  - Programs that never end

Three Types of Loops/Repetition in C

- while
  - top-tested loop (pretest)
- for
  - counting loop
  - forever-sentinel
- do
  - bottom-tested loop (posttest)

The while loop

- Top-tested loop (pretest)

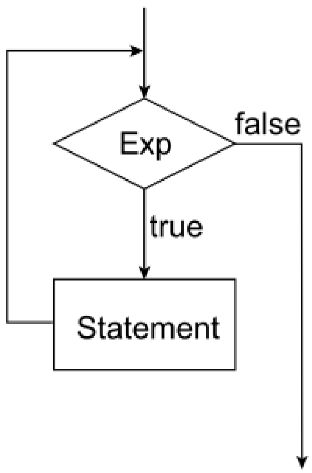
```
while (condition)
   statement;
```

- Note that, as in IF selection, only one statement is executed. You need a block to repeat more than one statement (using { })

While Loop

- Test at the top
  - May not be executed
- Loops while expression is true

```
while (expression)
statement;
int n = 100;
while (n > 0)
Cout<<"%d\n", n--;
```



Similar to the if statement

- Check the boolean condition
- If true, execute the statement/block
- Repeat the above until the boolean is false

Example

```
bool valid = true;              // Until we know otherwise

cout<<"Enter l: “;
cin>>"%lf", &l;

/* Test to see if the user entered an invalid value */
if(l <= 0)
{
   cout<<“you entered an invalid number!\n”;
   valid = false;
}
else
   cout<<"Okay, right\n”;
```

Remember this? What if we input invalid values?

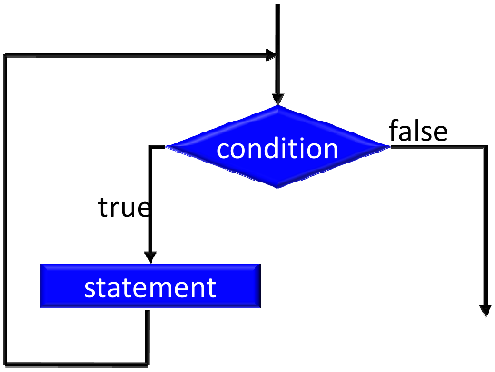
Example with while loop

```
bool valid = false;  /* Until we know otherwise */

while(!valid)       /* Loop until value is valid */
{
    cout<<"Enter l: “;
    cin>>"%lf", &l;

    /* Test to see if the user entered an invalid value */
    if(l < 0)
    {
        cout<<“you entered a negative number!\n”;
    }
    else if(l == 0)
    {
        cout<<“you entered zero.\n”;
    }
    else
    {
        cout<<"Okay, I guess that's reasonable\n”;
        valid = true;
    }
}
```

What does this do different?

while

```
while (condition)
    statement;

while (condition)
{
    statement1;
    statement2;
}
```




```
while(!valid)       /* Loop until value is valid */
{
   cout<<"Enter the inductance in millihenrys: “;
   cin>>"%lf", &l;

   if(l > 0)
   {
      valid = true;
   }
}
```

```
int i = 10;
while(i > 0)
{
    cout<<"i=%d\n", i;
    i = i - 1;
}
```


Forever loops and never loops

- Because the conditional can be “always true” or “always false”, you can get a loop that runs forever or never runs at all.

```
int count=0;
while(count !=0)
    cout<<“Hi Mom”;

while (count=1)
    count = 0;
```

What is wrong with these statements?

How to count using while

- First, outside the loop, initialize the counter variable
- Test for the counter's value in the boolean
- Do the body of the loop
- Last thing in the body should change the value of the counter!

```
i = 1;
while(i <= 10)
{
    cout<<"i=%d\n", i;
    i = i + 1;
}
```

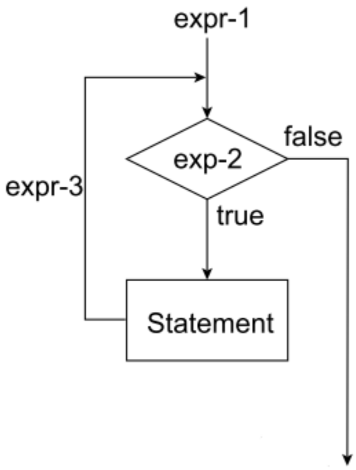

The for loop

- The while loop is pretty general. Anything that can be done using repetition can be done with a while loop
- Because counting is so common, there is a specialized construct called a for loop.
- A for loop makes it easy to set up a counting loop

For Loop

- Test at top
  - May not execute
- Any expression may be omitted
- Expression 1 is the initialize
  - Executed only once
- Expression 2 is the loop test
  - Loops while expression 2 is true
  - Tested after expr 1
  - Tested after expr 3
- Expression 3 is the update

```
for (expr-1; expr-2; expr-3)
    statement
```

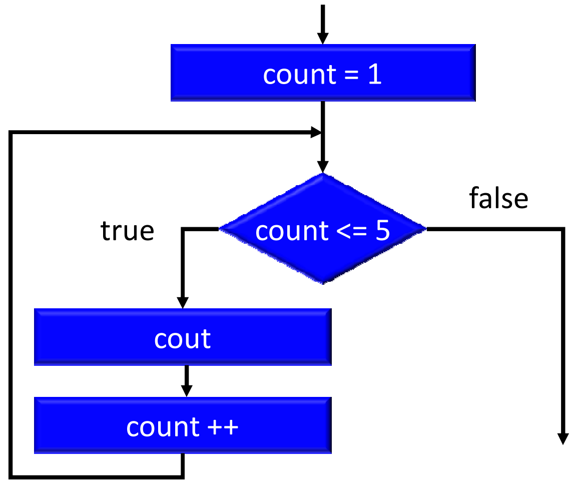


Three parts

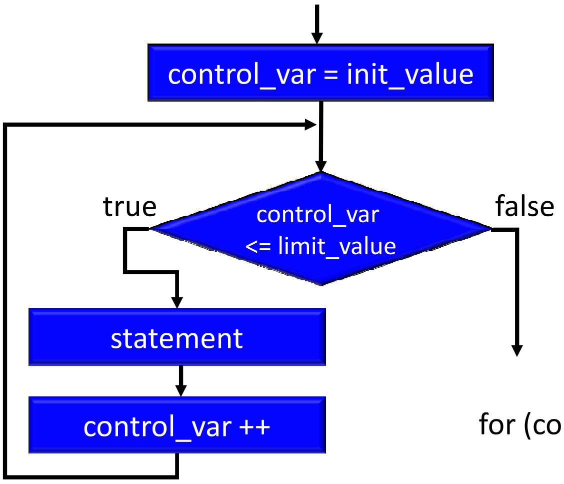
- Three parts to a for loop (just like the while):
  - Set the initial value for the counter
  - Set the condition for the counter
  - Set how the counter changes each time through the loop

```
for(count=1;count<=5; count++)
    statement;
```

for(count=1; count<=5; count++)
cout<<“count=%d\n”, count;

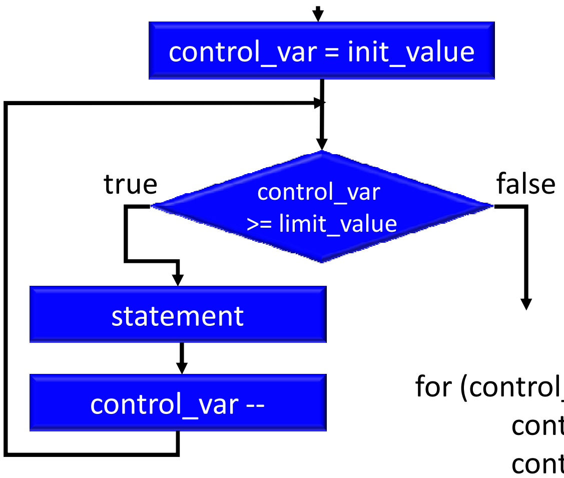


Ascending for <=,++



for (control_var=init_value;
control_var <=limit_value;
control_var++) statement;

Descending for >=,--




```
for (control_var=init_value;
        control_var >=limit_value;
        control_var--) statement;
```


Comments

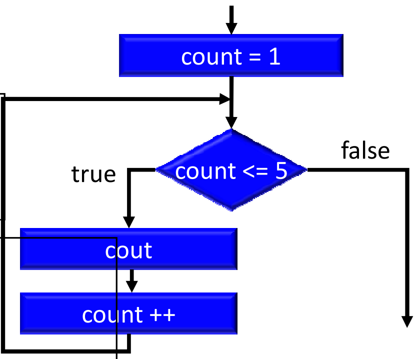
- It is dangerous to alter control_var or limit_var within the body of the loop.
- The components of the for statement can be a arbitrary statements, e.g. the loop condition may be a function call.

for(count=1; count<=5; count++) cout<<“count=%d\n”, count;

```
for(i=1;  i<=10;  i++)
{
  cout<<"%d\n", i;
}
```

```
for(t = 1.7; t < 3.5;  t = t + 0.1)
{
   cout<<"%f\n", t;
}
```

```
for(i=1;  i<5;  i++)
{
  for(j=1;  j<4;  j++)
  {
    cout<<"%d * %d = %d\n", i, j, i * j;
  }
}
```



Top-tested Equivalence

- The following loop

  for(x=init; x<=limit; x++)

  statement_list

- is equivalent to

  x=init;

  while (x<=limit){

  statement_list;

  x++;

  }

Some Magic Statements

s += 12; /* Equivalent to s = s + 12; */
s -= 13; /* Equivalent to s = s – 13; */

These work fine for integers or floating point

break;

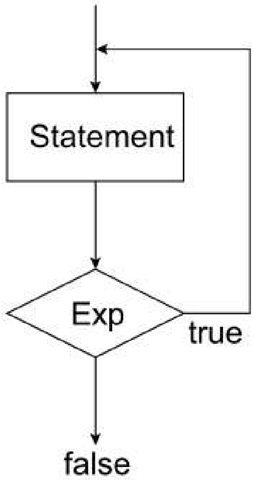
- The break statement exits the containing loop immediately!

```
while(true)       /* Loop until value is valid */
{
    cout<<"Enter l: ";
    cin>>"%lf", &l;

    /* Test to see if the user entered an invalid value */
    if(l <= 0)
    {
        cout<<"you entered an invalid number!\n";
    }
    else
    {
        cout<<"Okay, right\n";
        break;
    }
}
```

do-while Loop

- Test at the bottom
  - Executed at least once
- Loops while expression is true
  - Opposite of Pascal's repeat-until
- Useful when the test expression is effected by a statement in the loop body

```
do
{
statements;
} while (expression);
```



The do/while loop

Often just called a "do loop".

- do/while
  - – bottom-tested loop (posttest)

```
do
{
    angle += 2 * M_PI / 20;
    sinVal = sin(angle);
    cout<<“sin(%f) = %f\n”, angle, sinVal;
} while(sinVal < 0.5);
```

Bottom-tested Loop: do

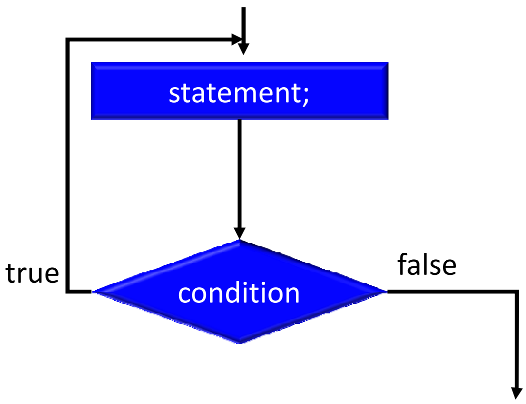
- Bottom-tested (posttest)
- One trip through loop is guaranteed, i.e. statement is executed at least once

```
do
    statement
while (loop_condition);
```

```
do
{
    statement1;
    statement2;
}
while (loop_condition);
```

Usually!

do { statement; } while(condition)



do/while Examples

```
angle = M_PI / 2;
do
{
   angle -= 0.01;
   cosVal = cos(angle);
   cout<<"cos(%f)=%f\n", angle, cosVal;
} while(cosVal < 0.5);
```

```
i = 0;
do
{
    i++;
    cout<<"%d\n", i;
} while(i < 10);
```

```
do
   {
       cout<<"Enter a value > 0: “;
       cin>>"%lf", &val;
   } while(val <= 0);
```


Bottom-tested Equivalence

- Bottom-tested do loop (posttest)

```
do
{
        statement;
}
while (condition);
```

- Similar to bottom-tested forever loop

```
for (;;)
{
        statement_list;
        if (!condition) break;
}
```

The "one off" error

- It is easy to get a for loop to be "one off" of the number you want. Be careful of the combination of init_value and < vs. <=
- Counting from 0, with <, is a good combination and good for invariants as well.

```
for(i=1;  i<10;  i++)
{
}

for(i=1;  i<=10;  i++)
{
}

for(i=0;  i<10;  i++)
{
}
```


The "one off" error

- It is easy to get a for loop to be "one off" of the number you want. Be careful of the combination of init_value and < vs. <=
- Counting from 0, with <, is a good combination and good for invariants as well.

```
for(i=1;  i<10;  i++)
{
}    9 values:  1 to 9

for(i=1;  i<=10;  i++)
{
}    10 values:  1 to 10

for(i=0;  i<10;  i++)
{    10 values:  0 to 9
}
```

while, for, do/while

```
for(t = 1.7; t < 3.5;  t = t + 0.1)
{
   cout<<"%f\n", t;
}
```

```
for(i=1;  i<5;  i++)
{
  for(j=1;  j<4;  j++)
  {
    cout<<"%d * %d = %d\n", i, j, i * j;
  }
}
```

```
i = 0;
do
{
    i++;
    cout<<"%d\n", i;
} while(i < 10);
```

```
int i = 10;
while(i > 0)
{
    cout<<"i=%d\n", i;
    i = i - 1;
}
```

```
while(!valid)       /* Loop until value is valid */
{
   cout<<"Enter the inductance in millihenrys: “;
   cin>>"%lf", &l;

   if(l > 0)
   {
      valid = true;
   }
}
```

```
angle = M_PI / 2;
do
{
   angle -= 0.01;
   cosVal = cos(angle);
   cout<<"cos(%f)=%f\n", angle, cosVal;
} while(cosVal < 0.5);
```


The null-statement

- Uncommon but sometimes useful with for-loops
- Not so useful with other control statements

```
if (expression) ;
for (ex1, ex2, ex3) ;
for (ex1, ex2, ex3);
```

Null Statement (pointing to the `;` in the first two lines)

More clear (beside the third line)

The goto Statement

- Unconditional Jump
- Must be used carefully
  - Use in rare, limited situations
    - Interrupting nested loops
    - Creating error handling code
    - Implementing state machines
- May not jump into (i.e., goto) functions, loops, ifs, or switches

```
for (i = 0; i <
100; i++)
{ for (j = 0; j <
100; j++)
{
...
if (error)
goto done;
}
}
done: ...;
```

Saturday, December 20, 2014 Software Techniques 33

More library functions

- exit(status)
  - Terminates program
  - Returns exit status (0 is okay, non-zero is error)
- getch() and getche()
  - Returns a character as soon as the key is pressed (do not need to press the enter key)
  - conio.h header file

Saturday, December 20, 2014 Software Techniques 34

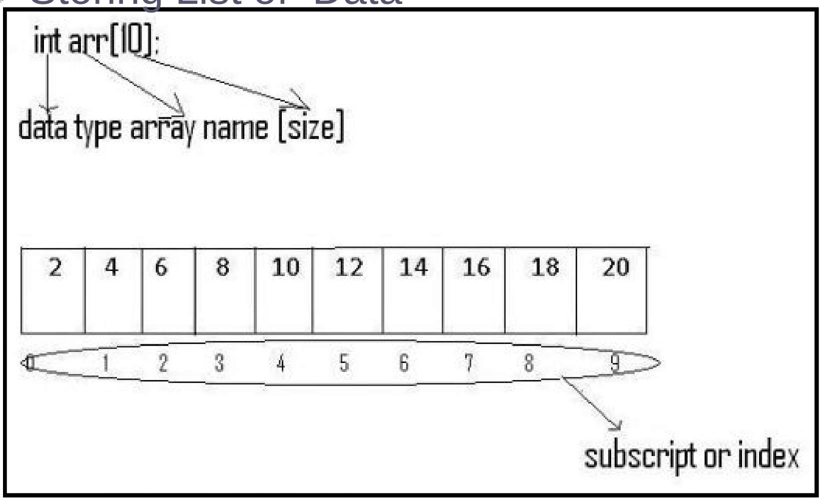
Program

```
// counts characters and words typed in
#include <iostream>
int main()
{
int chcount=0;
int wdcount=1;            // space between two words
char ch;
while( (ch=getche()) != '\r' )  // loop until Enter typed
    {
        if( ch==' ' )          // if it's a space
        wdcount++;             // count a word
        else                   // otherwise,
        chcount++;             // count a character
    }                   // display results
cout << "\nWords=" << wdcount << endl << "Letters=" << chcount << endl;
return 0;
}
```

Lecture Five
Arrays

Intro to Arrays

- Storing List of Data



Why Arrays

- Suppose we want to store the grade for each student in a class

/* Need a variable for each? */
int bob, mary, tom, …;

Easier to have a variable that stores the grades for all students

An array is a "Chunk of memory"

- An array is a contiguous piece of memory that can contain multiple values
- The values within the contiguous chunk can be addressed individually

Address in memory

0xeffffa00	0xeffffa04	0xeffffa08	0xeffffa0c	0xeffffa10	0xeffffa14	0xeffffa18	0xeffffa1c	0xeffffa20
74	59	95	85	71	45	99	82	76

grades

Array: “Chunk of memory”

Physical address	0xeffffa00	0xeffffa04	0xeffffa08	0xeffffa0c	0xeffffa10	0xeffffa14	0xeffffa18	0xeffffa1c	0xeffffa20
grades	74	59	95	85	71	45	99	82	76
index	0	1	2	3	4	5	6	7	8

- Use an index to access individual elements of the array: grades[0] is 74, grades[1] is 59, grades[2] is 95, and so on.

Array Declaration

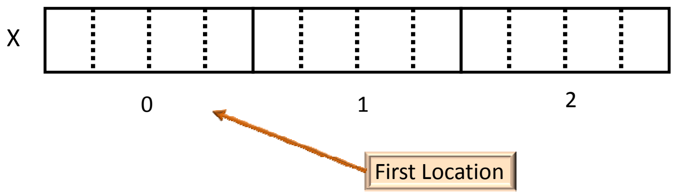
- Syntax for declaring array variable:
  type array_name[capacity];

  - type can be any type (int, float, char, …)
  - array_name is an identifier
  - capacity is the number of values it can store (indexing starts at 0)

Example

int x[3];   // an array of 3 integers
double y[7]; // an array of 7 doubles

Storage, e.g. 4-bytes per int



Notice: The first location is location 0 (zero)!

Operations with Arrays

- Assignment
  - x[0] = 6;   /* Assign 6 to element x[0] */
  - y[2] = 3.1;   /* Assign 3.1 to element y[2] */
- Access
  - m = x[2];
  - p = y[0];
- Input/Output:
  - the elements are handled as their types, e.g.

Cin<<&x[2]<< &y[3];
Cout<<x[0]<< y[2];   /* output 6 and 3.1 */

Arithmetic Operations

```
int main()
{       double x[5];        <- Variable Declaration for the array

         x[0] = 1;
         x[1] = 2;
         x[2] = x[0] + x[1];          /* X[2] = 3 */
         x[3] = x[2] / 3;             /* X[3] = 1 */
         x[4] = x[3] * x[2];          /* X[4] = 3 */
}
```


for loops

- "for" loops are ideal for processing elements in the array.

```
int main()
{
   int i;
    double values[4] = {3.14, 1.0, 2.61, 5.3};
   double sumValues = 0.0;

   for (i=0; i<4; i++)
    {
       sumValues = sumValues + values[i];
    }
   cout<<"Sum = %lf\n"<<sumValues;
}
```

for loops

- "for" loops are ideal for processing elements in the array.

```
int main()
{
    int i;
     double values[4] = {3.14, 1.0, 2.61, 5.3};
    double sumValues = 0.0;

    for (i=0; i<=4; i++)
     {
         sumValues = sumValues + values[i];
     }
    cout<<"Sum = %lf\n"<< sumValues;
}
```

ERROR! Out of bound

Initialization

- Syntax: int X[4] = {2, 4, 7, 9};
- Behavior: initialize elements starting with leftmost, i.e. element 0. Remaining elements are initialized to zero.

X	2	4	7	9
	0	1	2	3

- Initialize all to 0: int X[4]={0};

Example

```
int main()
{
   double grades[5] = {90, 87, 65, 92, 100};
   double sum;
   int i;

   cout<<"The first grade is: %.1f\n", grades[0];

   sum = 0;
   for(i=0;  i<5;  i++)
   {
      sum += grades[i];
   }
   cout<<"The average grade is: %.1f\n", sum / 5;

   grades[2] = 70;     /* Replaces 65 */
   grades[3] = grades[4];  /* Replaces 92 with 100 */
}
```


Constants for capacity

- Good programming practice:
  use #define for constants in your program
- For example:

```
 #define MaxLimit  25
int grades[MaxLimit];
for(int i; i<MaxLimit; i++){ };
```

- If size needs to be changed, only the capacity “MaxLimit” needs to be changed.

2-D Arrays

int cave[ArraySize][ArraySize];

Column

Row	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

2D Arrays

Column

Row	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

myMatrix[0][1] → 2

myMatrix[2][3] → 12

myMatrix[row][col]

int myMatrix[3][4] = { {1,2,3,4},{5,6,7,8},{9,10,11,12} };

Physically, in one block of memory

int myMatrix[2][4] = { {1,2,3,4},{5,6,7,8} };

ffe2de0c	ffe2de10	ffe2de14	ffe2de18	ffe2de1c	ffe2de20	ffe2de24	ffe2de28
1	2	3	4	5	6	7	8

row 1 (1–4) row 2 (5–8)

Array elements are stored in row major order. Row 1 first, followed by row2, row3, and so on

2D Array Name and Addresses

int myMatrix[2][4] = { {1,2,3,4},{5,6,7,8} };

ffe2de0c	ffe2de10	ffe2de14	ffe2de18	ffe2de1c	ffe2de20	ffe2de24	ffe2de28
1	2	3	4	5	6	7	8

myMatrix: pointer to the first element of the 2D array
myMatrix[0]: pointer to the first row of the 2D array
myMatrix[1]: pointer to the second row of the 2D array

Accessing 2D Array Elements

int myMatrix[2][4] = { {1,2,3,4} , {5,6,7,8} };

1	2	3	4	5	6	7	8

Indexing: myMatrix[i][j] is same as

```
(myMatrix[i] + j)
( (myMatrix + i))[j]
(( (myMatrix + i)) + j)
(&myMatrix[0][0] + 4*i + j)
```


Declaration

```
#define ROWS 3
#define COLS 5

   int table[ROWS][COLS];

   void display (table);
```

```
void display( int x[ROWS][COLS] )
{
    for (int i=0; i < ROWS; i++)
    {
        for (int j=0; j < COLS; j++ )
        {
            cout<<" x[%d][%d]: %d", i, j, x[i][j];
        }
        cout<<"\n";
    }
    cout<<"\n";
}
```

2D Arrays often require nested loops – two variables

Table A = { {13, 22, 9, 23},
{17, 5, 24, 31, 55},
{4, 19, 29, 41, 61} };

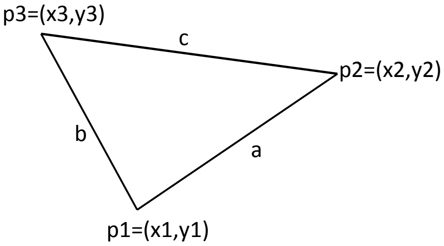
13	22	9	23	?
17	5	24	31	55
4	19	29	41	61

Table B = {1, 2, 3, 4,
5, 6, 7, 8, 9,
10, 11, 12, 13, 14 };

1	2	3	4	5
6	7	8	9	10
11	12	13	14	?

Lecture Six
Functions

Triangle Area Computation



$$a = |p2 - p1|$$
$$b = |p3 - p1|$$
$$c = |p3 - p2|$$

$$a = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

$$area = \sqrt{p(p-a)(p-b)(p-c)}$$
$$p = \frac{a+b+c}{2}$$

How would you write this program?

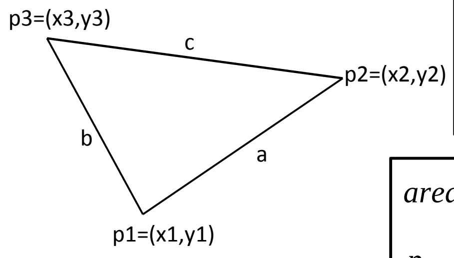
Variables

```
int main()
{
    double x1=0, y1=0;
    double x2=17, y2=10.3;
    double x3=-5.2, y3=5.1;

    double a, b, c;     /* Triangle side lengths */
    double p;           /* For Heron's formula */
    double area;
```

$$a = |p2 - p1|$$
$$b = |p3 - p1|$$
$$c = |p3 - p2|$$
$$a = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

Lengths of Edges

```
a = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
b = sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3));
c = sqrt((x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3));
```



$$a = |p2 - p1|$$
$$b = |p3 - p1|$$
$$c = |p3 - p2|$$
$$a = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

$$area = \sqrt{p(p-a)(p-b)(p-c)}$$
$$p = \frac{a+b+c}{2}$$

Area

```
p = (a + b + c) / 2;
area = sqrt(p * (p - a) * (p - b) * (p - c));

cout<<"%f\n", area;
```

$$area = \sqrt{p(p-a)(p-b)(p-c)}$$
$$p = \frac{a+b+c}{2}$$

Whole Program

What if I made a mistake on the edge length equation?

```
int main()
{
    double x1=0, y1=0;
    double x2=17, y2=10.3;
    double x3=-5.2, y3=5.1;

    double a, b, c;     /* Triangle side lengths */
    double p;           /* For Heron's formula */
    double area;

    a = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
    b = sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3));
    c = sqrt((x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3));

    p = (a + b + c) / 2;
    area = sqrt(p * (p - a) * (p - b) * (p - c));

    cout<<"%f\n", area;
}
```

Functions

- Functions are subprograms that perform some operation and return one value
- They "encapsulate" some particular operation, so it can be re-used by others (for example, the abs() or sqrt() function)

Characteristics

- Reusable code
  - code in sqrt() is reused often
- Encapsulated code
  - implementation of sqrt() is hidden
- Can be stored in libraries
  - sqrt() is a built-in function found in the math library

Writing Your Own Functions

- Consider a function that converts temperatures in Celsius to temperatures in Fahrenheit.
  - Mathematical Formula:

    F = C * 1.8 + 32.0

  - We want to write a C++ function called CtoF

Convert Function in C

```
double CtoF ( double paramCel )
{
    return paramCel*1.8 + 32.0;
}
```

- This function takes an input parameter called paramCel (temp in degree Celsius) and returns a value that corresponds to the temp in degree Fahrenheit

How to use a function?

```
#include <iostream.h>
double CtoF( double );
/*****************************************************************
    ********
* Purpose: to convert temperature from Celsius to Fahrenheit
******************************************************************
    *******/
int main()
{
    double c, f;
    cout<<"Enter the degree (in Celsius): ";
    cin>>&c;
    f = CtoF(c);
    cout<<"Temperature (in Fahrenheit) is %lf\n", f;
}
double CtoF ( double paramCel)
{
    return paramCel * 1.8 + 32.0;
}
```


Terminology

- Declaration: double CtoF(double);
- Invocation (Call): Fahr = CtoF(Cel);
- Definition:

```
double CtoF( double paramCel )
{
  return paramCel*1.8 + 32.0;
}
```

Function Declaration

return_type function_name (parameter_list)

- Also called function prototype:

double CtoF(double)

- Declarations describe the function:
  - the return type and function name
  - the type and number of parameters

Function Definition

```
return_type  function_name (parameter_list)
{
        ....
      function body
        ....
}
```

```
double  CtoF(double paramCel)
{
      return paramCel*1.8 + 32.0;
}
```

Function Invocation

```
int main()
{ …
  f = CtoF(c);
}
```

- Call copies argument c to parameter paramCel

- Control transfers to function “CtoF”

```
double CtoF ( double paramCel )
{
    return paramCel*1.8 + 32.0;
}
```

Thursday, January 15, 2015 Data Structure 15

Invocation (cont)

```
int main()
{ …
   f = CtoF(c);
}
```

- Expression in “CtoF” is evaluated

- Value of expression is returned to “main”

```
double CtoF ( double paramCel )
{
    return paramCel*1.8 + 32.0;
}
```

Thursday, January 15, 2015 Data Structure 16

Local Objects

- The parameter "paramCel" is a local object which is defined only while the function is executing. Any attempt to use "paramCel" outside the function is an error.
- The name of the parameter need not be the same as the name of the argument. Types must agree.

Can we do better than this?

```
int main()
{
   double x1=0, y1=0;
   double x2=17, y2=10.3;
   double x3=-5.2, y3=5.1;

   double a, b, c;    /* Triangle side lengths */
   double p;          /* For Heron's formula */
   double area;

   a = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
   b = sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3));
   c = sqrt((x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3));

   p = (a + b + c) / 2;
   area = sqrt(p * (p - a) * (p - b) * (p - c));

   cout<<"%f\n", area;
}
```

What should we name our function?

```
a = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
```

"Length" sounds like a good idea.

```
???    Length(   ???   )
{
}
```


What does our function need to know?

```
a = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
```

(x, y) for two different points:

```
???    Length(double x1, double 
y1,
                 double x2, double 
y2)
{
}
```

What does our function return?

```
a = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
```

A computed value which is of type *double*

```
double Length(double x1, double y1,
              double x2, double y2)
{
}
```


How does it compute it?

```
a = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
```

A computed value which is of type *double*

```
double Length(double x1, double y1,
              double x2, double y2)
{
    double len;
    len = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
    return(len);
}
```

Using This

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

/* Declaration */
double Length(double x1, double y1, double x2, double y2);

/*
 * Program to determine the area of a triangle
 */

int main()
{
    double x1=0, y1=0;
    double x2=17, y2=10.3;
    double x3=-5.2, y3=5.1;

    double a, b, c;     /* Triangle side lengths */
    double p;           /* For Heron's formula */
    double area;

    a = Length(x1, y1, x2, y2);
    b = Length(x1, y1, x3, y3);
    c = Length(x2, y2, x3, y3);

    p = (a + b + c) / 2;
    area = sqrt(p * (p - a) * (p - b) * (p - c));

    cout<<"%f\n", area;
}

/* Definition */
double Length(double x1, double y1, double x2, double y2)
{
    double len;
    len = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
    return(len);
}
```

Declaration

Invocations

Definition

Potential Errors

```
#include <iostream.h>
double convert( double );
int main()
{    double c, f;
     cout<<"Enter the degree (in Celsius): ";
      cin>>"%lf", &c;
     f= convert(c);
     cout<<"Temp (in Fahrenheit) for %lf Celsius is %lf", paramCel, f;
}
double CtoF( double paramCel)
{
     return c * 1.8 + 32.0;
}
```

Error! paramCel is not defined

Error! C is not defined

- Scope – Where a variable is known to exist.
- No variable is known outside of the curly braces that contain it, even if the same name is used!

Another Example

```
#include <iostream.h>

double GetTemperature();
double CelsiusToFahrenheit( double );
void DisplayResult( double, double );

int main()
{
  double
    TempC,  // Temperature in degrees Celsius
    TempF;  // Temperature in degrees Fahrenheit

  TempC = GetTemperature();
  TempF = CelsiusToFahrenheit(TempC);
  DisplayResult(TempC, TempF);

  return 0;
}
```

Declarations

Invocations

Function: GetTemperature

```
double GetTemperature()
{
  double Temp;

  cout<<"\nPlease enter a temperature in degrees
          Celsius: “;
  cin>>&Temp;
  return Temp;
}
```

Function: CelsiusToFahrenheit

```
double CelsiusToFahrenheit(double Temp)
{
  return (Temp * 1.8 + 32.0);
}
```


Function: DisplayResult

```
void DisplayResult(double CTemp, double FTemp)
{
  cout<<"Original: %5.2f C\n", CTemp;
  cout<<"Equivalent: %5.2f F\n", FTemp;

  return;
}
```

Declarations (Prototypes)

```
double GetTemp( );
double CelsiusToFahrenheit( double );
void Display( double, double );
```

- void means "nothing". If a function doesn't return a value, its return type is void

Abstraction

1. Get Temperature
2. Convert Temperature
3. Display Temperature

We are hiding details on *how* something is done in the function implementation.

```
int main()
{
  double
    TempC,          // Temperature in degrees Celsius
    TempF;          // Temperature in degrees Fahrenheit

  TempC = GetTemperature();
  TempF = CelsiusToFahrenheit(TempC);
  DisplayResult(TempC, TempF);

  return 0;
}
```

Another Way to Compute Factorial

Pseudo code for factorial(n)

```
if n == 0 then
   result = 1
else
   result = n * factorial(n – 1)
```

After all, 5! = 5 * 4 * 3 * 2 * 1 = 5 * 4!

Recursive Functions

Factorial function contains an invocation of itself.

We call this a: *recursive call*.

```
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * Factorial(n-1);
}
```

Recursive functions must have a *base case* (if n == 0): why?

This works much like proof by induction.

```
if n == 0 then
   result = 1
else
   result = n * factorial(n – 1)
```

Infinite Recursion

```
int Factorial(int n)
{
    return n * Factorial(n-1);
}
```

What if I omit the “base case”?

This leads to *infinite recursion*!

Factorial(3)=
3 * Factorial(2) =
3 * 2 * Factorial(1) =
3 * 2 * 1 * Factorial(0) =
3 * 2 * 1 * 0 * Factorial(-1) =
…

Psuedocode and Function

```
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * Factorial(n-1);
}
```

Base Case

if n == 0 then
  result = 1
else
  result = n * factorial(n – 1)

Declaration:
```
int Factorial(int n);
```

Invocation:
```
f = Factorial(7);
```

Definition:
```
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * Factorial(n-1);
}
```

Lecture Seven
Passing parameters of Functions

Pass By Value

Copy values from the call to the formal parameter

```
void  func(int i);
int  main( )
{
int  a = 5;
func(a);
}
void  func(int i)
{
i = i + 1;
}
```

Pointers and References

- Pointers and references are both addresses
- Pointers are simple, primitive data types
  - Addresses are exposed
  - Addresses may be operated on (i.e., they allow address arithmetic)
  - Must be explicitly de-referenced
- References wrap addresses
  - Addresses are NOT exposed
  - Addresses may NOT be operated on (address arithmetic disallowed)
  - Dereferencing is automatic

Address Operators

Memory content versus memory address

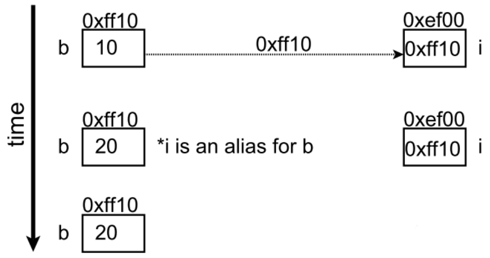
```
int i;
int *p = &i;
i = 10;
*p is now 10
```

i	undefined	0x0a000010	i is undefined
p	0x0a000010	0x0a000014	p is 0x0a000010 *p is undefined
i	10	0x0a000010	i is 10
p	0x0a000010	0x0a000014	p is 0x0a000010 *p is 10

Pass By Pointer

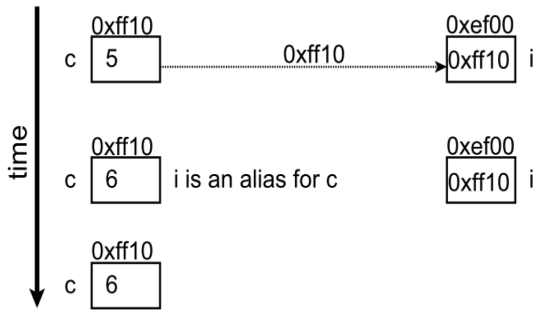
- Pass by address is used
  - When a function must change its argument
  - To increase efficiency when passing large data types

```
void  func(int* i);
void main( )
{
int b = 10;
func(&b);
}
void func(int* i)
{
*i = *i + 10;
}
```



Pass By Reference

```
void  func(int& i);
int  main( )
{
int  c = 5;
func(c);
}
void  func(int&  i)
{
i = i + 1;
}
```



Pass By Reference Example

- Pass the address of an object
  - Provides efficiency (especially for large objects)
  - Propagate parameter changes to the calling function
  - Creates an alias (i.e., two names for the same memory location)
  - Do not have the “safety” of a pass by value
- One place to modify syntax
  - Function definition (from value to reference variable)
- Example: Swapping variables

```
void    swap(int&  v1,  int&
   v2)
{
int   temp = v1;
v1 = v2;
v2 = temp;
}
```

```
int  main( )
{
int   a = 10, b = 20;
swap(a, b);
}
```


Swapping Two Variables

- The classic pass by address example

```
void   swap(int*   v1,
  int* v2)
{
int   temp = *v1;
*v1 = *v2;
*v2 = temp;
}
```

```
int  main( )
{
int   a = 10, b = 20;
swap(&a, &b);
}
```

Arrays as parameters of functions

```
int main()
{
    double values[4] = {3.14, 1.0, 2.61, 5.3};

    cout<<"Sum = %lf\n", SumValues( values, 4));
}
```

Suppose we want a function that sums up values of the array

Arrays as parameters of functions

```
double SumValues(double x[], int numElements)
{
    int i;
        double result = 0;
        for (i=0; i < numElements; i++)
                result = result + x[i];
        return result;
}
```

"[]" flags the parameter as an array.

- always passed by reference Array size is passed separately (as numElements)

```
Array before sorting
  Element     0 :  58.7000
  Element     1 :   8.0100
  Element     2 :  72.3700
  Element     3 :   4.6500
  Element     4 :  58.3000
  Element     5 :  92.1700
  Element     6 :  95.3100
  Element     7 :   4.3100
  Element     8 :  68.0200
  Element     9 :  72.5400

Array after sorting
  Element     0 :   4.3100
  Element     1 :   4.6500
  Element     2 :   8.0100
  Element     3 :  58.3000
  Element     4 :  58.7000
  Element     5 :  68.0200
  Element     6 :  72.3700
  Element     7 :  72.5400
  Element     8 :  92.1700
  Element     9 :  95.3100
```

Sample output : The array elements are randomly generated


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

void PrintArray( double [], int );
void SortArray( double [], int );
void Swap (double *, double *);
```

Functions are your friends! Make them work and then use them to do work!

```
#define NumElements 10

int main()
{
    int i;
    double values[NumElements]; /* The array of real numbers */

    srand(time(NULL));

    for (i=0; i < NumElements; i++)
    {
        values[i] = (double)(rand() % 10000) / 100.0;
    }

    cout<<"\nArray before sorting\n");
    PrintArray( values, NumElements );

    SortArray( values, NumElements );

    cout<<"\nArray after sorting\n");
    PrintArray( values, NumElements );

    return 0;
}
```

Tuesday, March 03, 2015 Data Structure 13

```
#define NumElements 10

int main()
{
    int i;
    double values[NumElements]; /* T

    srand(time(NULL));

    for (i=0; i < NumElements; i++)
    {
        values[i] = (double)(rand() % 10000) / 100.0;
    }

    cout<<"\nArray before sorting\n");
    PrintArray( values, NumElements );

    SortArray( values, NumElements );

    cout<<"\nArray after sorting\n");
    PrintArray( values, NumElements );

    return 0;
}
```

Array declaration

Declare an array of 10 doubles

The indices range from 0 to 9, i.e. `Value[0]` to `Value[9]`

Tuesday, March 03, 2015 Data Structure 14

```
#define NumElements 10

int main()
{
    int i;
    double values[NumElements]; /* The array of real numbers */

    srand(time(NULL));

    for (i=0; i < NumElements; i++)
    {
        values[i] = (double)(rand() % 10000) / 100.0;
    }
}
```

Initialize the array with random values

rand() returns a pseudo random number between 0 and RAND_MAX

rand()%10000 yields a four-digit integer remainder

/100.0 moves the decimal point left 2 places

So, Values is an array of randomly generated 2-decimal digit numbers between 0.00 and 99.99


```
cout<<"\nArray before sorting\n");
PrintArray( values, NumElements );
```

PrintArray prints the elements of the array in the order they are given to it

```
SortArray( values, NumElements );
```

SortArray sorts the elements into ascending order

```
cout<<"\nArray after sorting\n");
PrintArray( values, NumElements );
```

Parameter Passing

```
void PrintArray( double array[], int size )
{
}
```

array is an array of doubles
array is passed by reference, i.e. any changes to parameter *array* in the function would change the argument values
The array size is passed as "size"


```
void PrintArray( double array[], int size )
{
    int i;

    for (i=0; i<size; i++)
       cout<<"  Element %5d : %8.4lf\n",i, array[i]);
}
```

array[i] is a double so the output needs to be "%f"

The range of the "for" statement walks through the whole array from element 0 to element N-1.

Sorting Array

```
void SortArray( double array[], int size)
{
}
```

array is an array of doubles.

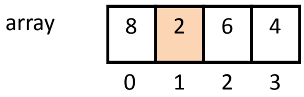
array is passed by reference, i.e. changes to parameter array change the argument values
There is no size restriction on array so the size is passed as "size".

Selection Sort

array

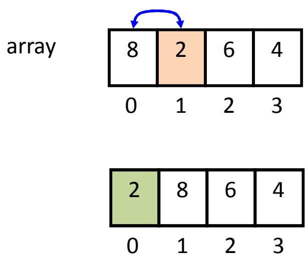
0	1	2	3
8	2	6	4

Selection Sort



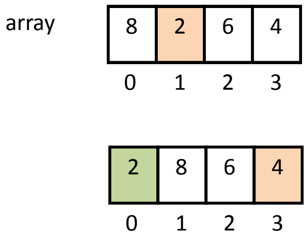
Search from array[0] to array[3]
to find the smallest number

Selection Sort



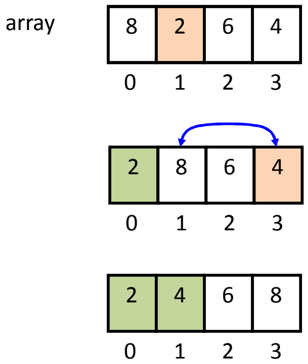
Search from array[0] to array[3]
to find the smallest number and
swap it with array[0]

Selection Sort



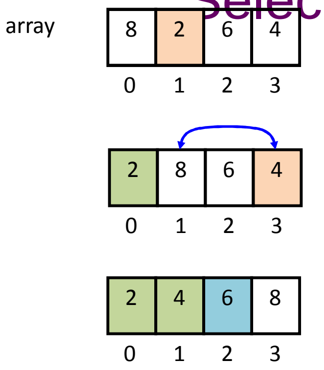
Search from array[1] to array[3]
to find the smallest number

Selection Sort



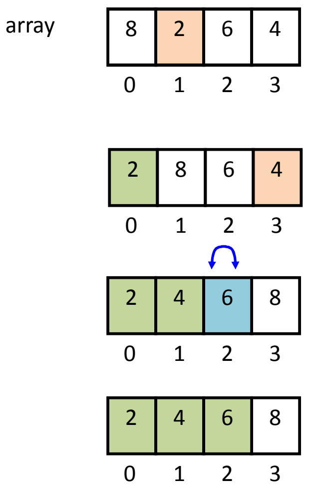
Search from array[1] to array[3]
to find the smallest number and
swap it with array[1]

Selection Sort



Search from array[2] to array[3]
to find the smallest number and
swap it with array[2]

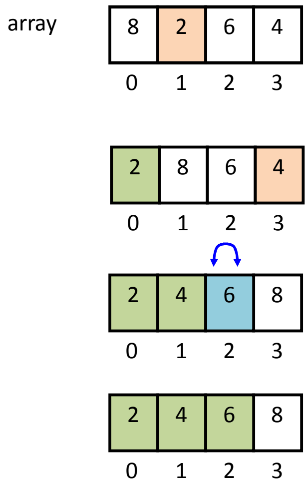
Selection Sort



Search from array[2] to array[3]
to find the smallest number and
swap it with array[2]

And we are done!

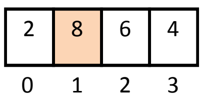
Selection Sort



How many iterations are there?
Answer: 3 (from i = 0 to i = 2)

More generally, if number of elements in the array is size, you need to iterate from
i = 0 to i = size - 2

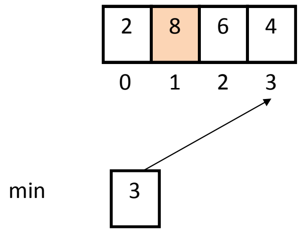
Selection Sort



At every iteration i, you need to search from array[i] to array[size – 1] to find the smallest element

How to do this?

Selection Sort

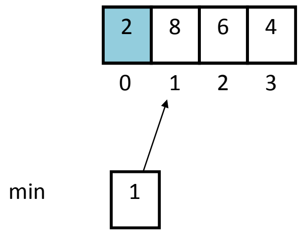


At every iteration i, you need to search from array[i] to array[size – 1] to find the smallest element

How to do this?

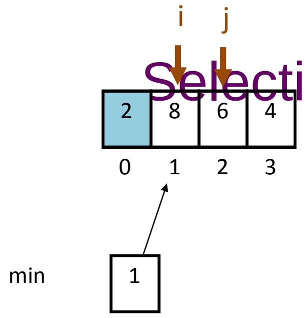
Use a variable called min to locate the *index* of the smallest element

Selection Sort



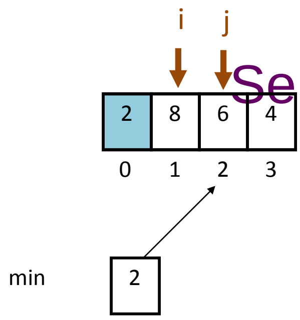
Assume current iteration i = 1
Initialize min = i

Selection Sort



Assume current iteration i = 1
Initialize min = I
Set j = i + 1
Compare array(min) to array(j)

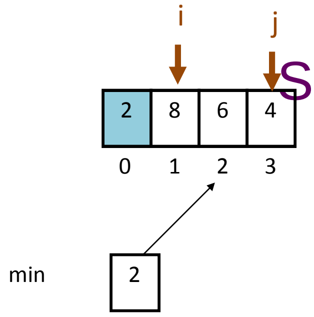
Selection Sort



Assume current iteration i = 1
Initialize min = i
Set j = i + 1
Compare array(min) to array(j)
If array(j) < array(min)
set min to j

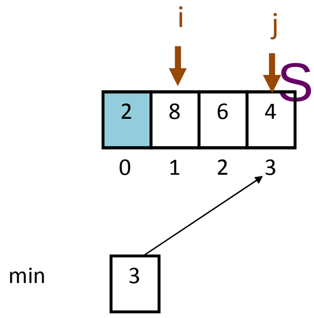
Because 6 < 8,
min is now set to 2

Selection Sort



Increment j
Compare array(min) to array(j)

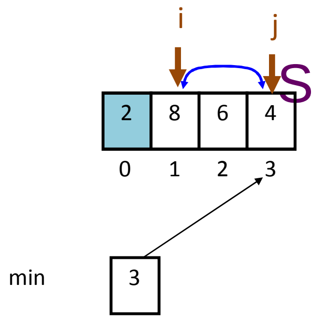
Selection Sort



Increment j
Compare array(min) to array(j)
If array(j) < array(min)
set min to j

Because 4 < 6,
min is now set to 3

Selection Sort



Swap array(i) with array(min)

SortArray

```
void SortArray( double array[], int size)
{
    int i, j, min;

    for (i=0; i < size-1; i++)
    {
        min = i;
        for (j=i+1; j<size; j++)
        {
            if (array[j] < array[min])
            {
                min = j;
            }
        }

        Swap(&array[i], &array[min]);
    }
}
```

Swap

```
void Swap (double *a, double *b)
                        {
   double temp = *a;
   *a = *b;
   *b = temp;
}
```


Swap

```
void Swap (double *a, double *b)
                        {
              double temp = *a;
                   *a = *b;
   *b = temp;
}
```

Note: We're passing two elements of the array; not passing the entire array

So, we **CANNOT** declare it as

void Swap(double a, double b)

void Swap(double a[], double b[])

passing 2d arrays

In passing a multi-dimensional array, the first array size does not have to be specified. The second (and any subsequent) dimensions must be given!

```
int myFun(int list[][10]);
```



```
#define ROWS 3
#define COLS 5

int addMatrix( int [ ][COLS] );

int main()
{
   int a[][COLS] =  { {13, 22,  9, 23, 12}, {17,  5, 24, 31, 55}, {4, 19, 29, 41, 61}  };
   printf("Sum = %d\n", addMatrix( a ) );
}

int addMatrix( int t[ ][COLS] )
{
   int i, j, sum = 0;
   for (i=0; i<ROWS; i++)
      for (j=0; j<COLS; j++)
         sum += t[i][j];
   return sum;
}
```

Recursive Functions

- Direct recursion: a function calls itself
- Indirect recursion: A calls B, B calls C, ..., Y calls Z, Z calls A
- Must have 3 features
  - Branch (usually in an if) that makes the recursive call
  - Branch (usually in an if) that does not recurse (i.e., terminates the recursion) --condition may be implicit rather than explicit
  - Input must change with each call
- Theoretically, recursion may be written as a loop
  - There is an existence proof of this – but it's not a constructive proof

Recursion Example 1

- (Print the digits of an integer one at a time)

```
void forward(int number)
{
if (number != 0)
{
forward(number / 10);
cout << number % 10;
}
}
```

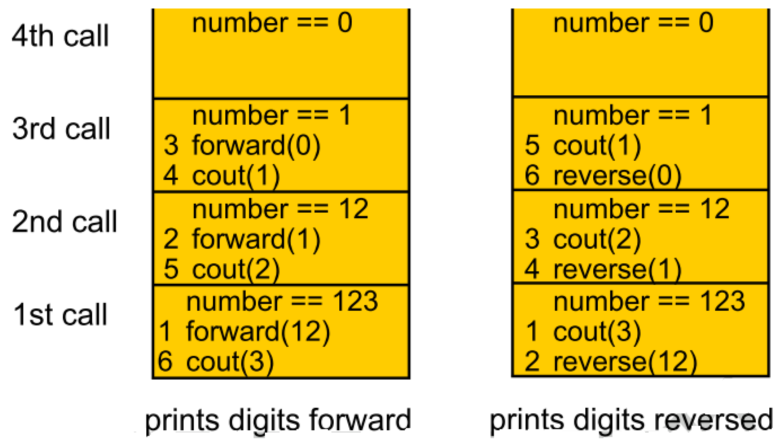
Recursion Example 2

- (Print the digits of an integer in reverse order)

```
void reverse(int number)
   {
   if (number != 0)
   {
   cout << number % 10;
   reverse(number / 10);
   }
   }
```


Graphical Representation

- Activation records and statement sequencing

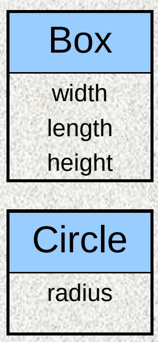


Lecture Eight
Struct

structs

- Aggregating associated data into a single variable

```
int main()
{
        Box mybox;
        Circle c;

        mybox.width = 10;
        mybox.length = 30;
        mybox.height = 10;
        c.radius = 10;
}
```



The idea

- I want to describe a box. I need variables for the width, length, and height.
- I can use three variables, but wouldn't it be better if I had a single variable to describe a box?
- That variable can have three parts, the width, length, and height.

Box
width
length
height

Structs

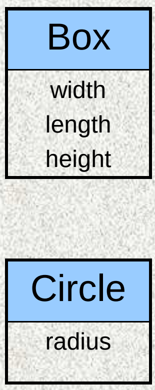
- A struct (short for structure) in C is a grouping of variables together into a single type.

```
struct nameOfStruct
{
    type member;
    type member;
    …
};
```

Note the semicolon at the end.
To declare a variable:

`struct nameOfStruct variable_name;`

Example



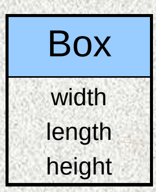

```
#include <stdio.h>
struct Box
{
        int width;
        int length;
        int height;
};
struct Circle
{
        double radius;
};
int main()
{
        struct Box b;
        struct Circle c;
}
```

Data structure definition

You can declare variables

Example

```
#include <stdio.h>
struct Box
{
        int width;
        int length;
        int height;
};
```




```
int main() {
        struct Box b;

        b.width = 10;
        b.length = 30;
        b.height = 10;
}
```

You can assign values to each member

We use a period "." to get to the elements of a struct.

If x is a struct, x.width is an element in a struct.

Another Example

- You can use mixed data types within the struct (int, float, char [])

```
struct bankRecordStruct
{
   char name[50];
   float balance;
};

struct bankRecordStruct billsAcc;
```


Accessing values

Access values in a struct using a period: "."

```
struct bankRecordStruct
{
   char name[50];
   float balance;
};

struct bankRecordStruct billsAcc;

cout<<<<"My balance is: %f\n", billsAcc.balance;

float bal = billsAcc.balance;
```

Assign Values using cin>>

```
struct BankRecord
{
    char name[50];
    float balance;
};


int main()
{
   struct BankRecord newAcc;  /* create new bank record */

   cout<<"Enter account name: ";
   cin>> "%50s", newAcc.name;
   cout<<"Enter account balance: ";
   cin>>"%d", &newAcc.balance;
}
```


Copy via =

- You can set two struct type variables equal to each other and each element will be copied

```
struct Box { int width, length, height; };

int main()
{
   struct Box b, c;
   b.width = 5; b.length=1; b.height = 2;
   c = b;          // copies all elements of b to c
   cout<<"%d %d %d\n", c.width, c.length, c.height;
}
```

Passing Struct to a function

- You can pass a struct to a function. All the elements are copied
- If an element is a pointer, the pointer is copied but not what it points to!

```
int myFunction(struct Person p)
{
…
}
```


Using Structs in Functions

- Write a program that
  - Prompts the user to enter the dimensions of a 3D box and a circle
  - Prints the volume of the box and area of the circle
- Sample run:

```
Enter the box dimensions (width,length,height): 1 2 3
Enter the radius of the circle: 0.8

Box volume = 6
Circle area = 2.01
```

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

struct Box { int width, height , length; };

int GetVolume(struct Box b)
{
     return b.width * b.height * b.length;
}

int main()
{
     struct Box b;

     cout<<"Enter the box dimensions (width length height): “;
     cin>>"%d %d %d", &b.width, &b.length, &b.height;

     cout<<"Box volume = %d\n", GetVolume(b);
}
```


Note: == Comparison doesn’t work

```
struct Box { int width, length, height; };

int main()
{
        struct Box b, c;
        b.width = 5; b.length=1; b.height = 2;
        c = b;
        if (c == b)      /* Error when you compile! */
                cout<<“c and b are identical\n”;
        else
                cout<<“c and b are different\n”;
}
```

Error message: invalid operands to binary == (have 'Box' and 'Box')

Create your own equality test

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

struct Box { int width, height , length; };

int IsEqual(struct Box b, struct Box c)
{
   if (b.width==c.width &&
       b.length==c.length &&
       b.height==c.height)
        return 1;
   else
        return 0;
}
```

```
    struct Box b, c;
    b.width = 5; b.length=1; b.height
= 2;
    c = b;

    if (IsEqual(b,c))
        cout<<"c and b are
identical\n”;
    else
        cout<<"c and b are
different\n”;
```


typedef

- typedef is a way in C to give a name to a custom type.

```
typedef type newname;

typedef int dollars;
typedef unsigned char Byte;
```

I can declare variables like:

dollars d;
Byte b, c;

It’s as if the type already existed.

typedef for Arrays

- There is a special syntax for arrays:

```
typedef char Names[40];
typedef double Vector[4];
typedef double Mat4x4[4][4];
```

- Now, instead of:

```
double mat[4][4];
```

- I can do:

```
Mat4x4 mat;
```

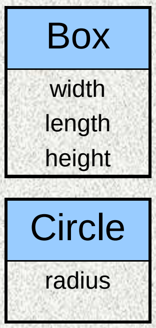

Using Structs with Typedef

```
typedef struct [nameOfStruct]
{
    type member;
    type member;
     …
} TypeName;
```

optional

- To declare a variable: TypeName variable_name;

Example




```
#include <stdio.h>
typedef struct
{
        int width;
        int length;
        int height;
} Box;
typedef struct {  double radius; } Circle;
int main()
{
        Box b;    /* instead of struct Box */
        Circle c; /* instead of struct Circle */
        b.width = 10;
        b.length = 30;
        b.height = 10;
        c.radius = 10;
}
```


Arrays of structs

- You can declare an array of a structure and manipulate each one

```
typedef struct
{
  double radius;
  int x;
  int y;
  char name[10];
} Circle;

Circle circles[5];
```

Size of a Struct: sizeof

```
typedef struct
{
  double radius;       /* 8 bytes */
  int x;               /* 4 bytes */
  int y;               /* 4 bytes */
  char name[10];       /* 10 bytes */
} Circle;

cout<<"Size of Circle struct is %d\n",
        sizeof(Circle);
```


Size of a Struct

- 8 + 4 + 4 + 10 = 26
  - But sizeof() reports 28 bytes!!!
- Most machines require alignment on 4-byte boundary (a word)
  - last word is not filled by the char (2 bytes used, 2 left over)

D D D D	D D D D	I I I I	I I I I	C C C C	C C C C	C C X X
8 byte, 2 word double		4 byte, 1 word integer	4 byte, 1 word integer	10 byte char array, 2 bytes of the last word unused		

Pointers to structs

```
typedef struct
{
   int width;
   int length;
   int height;
} Box;
```

```
Box b;  /* A variable of type Box */
Box *c; /* A pointer to a Box */
double  w;

b.width = 5;  b.height = 7;  b.length = 3;

c = &b; /* Same as before */

w = c->width;
```

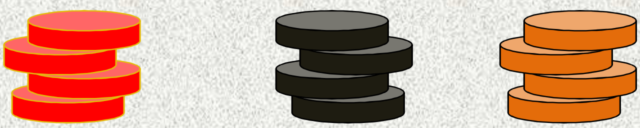
- To access the members of a struct, we use: . for a variable of the struct's type, -> for a pointer to a struct

struct Concepts

```
struct Box
{
   double wid, hit;
};

typedef struct
{
  double radius;
  int x;
  int y;
  char name[10];
} Circle;
```

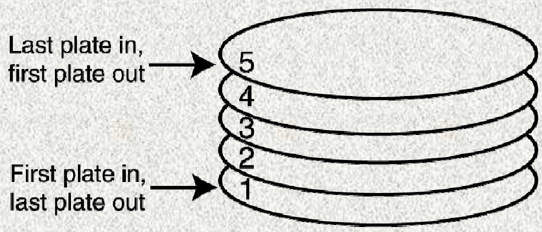
```
struct Box b;        /* No typedef */
Circle c;            /* typedef */

struct Box *pBox; /* Pointer to Box */
Circle *pCirc;       /* Pointer to Circle */

pBox = &b;           /* Get pointer to a Box */
b.wid = 3;
pBox->wid = 7;

pCirc = &c;
(*pCirc).radius = 9;
```

Lecture Eight
Stacks



Stack

- A stack is a data structure that stores and retrieves items in a last-in-first-out (LIFO) manner.



Applications of Stacks

- Computer systems use stacks during a program's execution to store function return addresses, local variables, etc.
- Some calculators use stacks for performing mathematical operations.

Static and Dynamic Stacks

- Static Stacks
  - Fixed size
  - Can be implemented with an array
- Dynamic Stacks
  - Grow in size as needed
  - Can be implemented with a linked list

Stack Operations

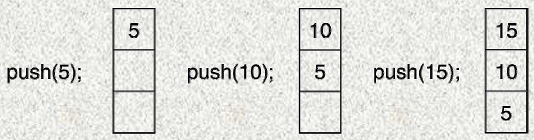
- Push
  - causes a value to be stored in (pushed onto) the stack
- Pop
  - retrieves and removes a value from the stack

The Push Operation

- Suppose we have an empty integer stack that is capable of holding a maximum of three values. With that stack we execute the following push operations.
  - push(5);
  - push(10);
  - push(15);

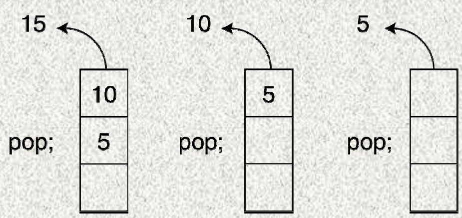
The Push Operation

- The state of the stack after each of the push operations:



The Pop Operation

- Now, suppose we execute three consecutive pop operations on the same stack:



Other Stack Operations

- isFull: A Boolean operation needed for static stacks. Returns true if the stack is full. Otherwise, returns false.
- isEmpty: A Boolean operation needed for all stacks. Returns true if the stack is empty. Otherwise, returns false.

Stacks

- Problem:
  - What happens if we try to pop an item off the stack when the stack is empty?
    - This is called a stack underflow. The pop method needs some way of telling us that this has happened. In java we use the java.util.EmptyStackException

Implementing a Stack

- There are two ways we can implement a stack:
  - Using an array
  - Using a linked list

Implementing a Stack

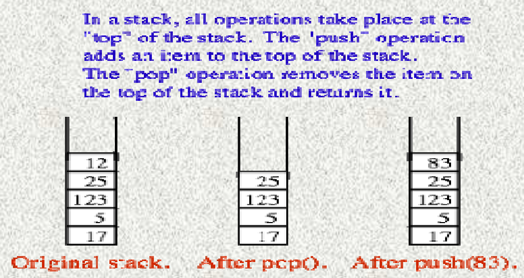
- Implementing a stack using an array is fairly easy.
  - The bottom of the stack is at data[0]
  - The top of the stack is at data[numItems-1]
  - push onto the stack at data[numItems]
  - pop off of the stack at data[numItems-1]

Implementing a Stack

- Implementing a stack using a linked list isn't that bad either…
  - Store the items in the stack in a linked list
  - The top of the stack is the head node, the bottom of the stack is the end of the list
  - push by adding to the front of the list
  - pop by removing from the front of the list

C program to implement Stack [using Array]

- Stack is maintained in (LIFO) Last In First Out manner.
- Push and pop are done at the front end so we require a variable top which will give Status of current position.
- Initially value of top is -1 which indicates Stack is empty.



C++ program to implement Stack [using Array]

```
#include"iostream.h"
#define max 5      // size of the Stack
int top,a[max];
void push(void)
{
        int x;
        if(top==max-1)     // Condition for checking If Stack is Full
        {
               cout<<"stack overflow\n";
               return;
        }
        cout<<"enter a no\n";
        cin>>"%d",&x;
        a[++top]=x;    //increment the top and inserting element
        cout<<"%d succ. pushed\n",x;
        return;
}
```


C++ program to implement Stack [using Array]

```
void pop(void)
{
        int y;
   if(top==-1) // Condition for checking If Stack is Empty
        {
               cout<<"stack underflow\n";
               return;
        }
        y=a[top];
       a[top--]=0;
//insert 0 at place of removing element and decrement the top
        cout<<"%d succ.poped\n",y;
        return;
}
```

C++ program to implement Stack [using Array]

```
void display(void)
{
        int i;
        if(top==-1)
        {
              cout<<"stack is empty\n”;
              return;
        }
        cout<<"elements of Stack are :\n”;
        for(i=0;i<=top;i++)
        {
               cout<<"%d\n",a[i];
         }
         return;
}
```


C++ program to implement Stack [using Array]

```
void main(void)
    {
          int c;  top=-1;
           do
          {
                cout<<"1:push\n2:pop\n3:display\n4:exit\nchoice:”;
                cin>>"%d",&c;
                 switch(c)
                {
                        case 1:push();
                        break;
                        case 2:pop();
                        break;
                        case 3:display();
                        break;
                        case 4:cout<<"program ends\n”;
                        break;
                        default :cout<<"wrong choice\n”;
                        break;
                 }
          }while(c!=4);
    }
```

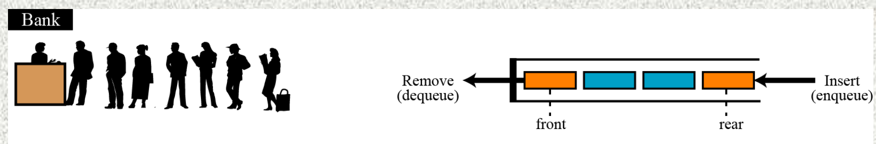
Reversing a Word

- We can use a stack to reverse the letters in a word.
- How?

Reversing a Word

- Read each letter in the word and push it onto the stack
- When you reach the end of the word, pop the letters off the stack and print them out.

Lecture Eight
Queues



A queue of people

A computer queue

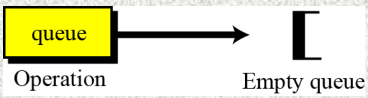
Queue

- A queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front.
- These restrictions ensure that the data is processed through the queue in the order in which it is received.
- In other words, a queue is a first in, first out (FIFO) structure.

Operations on queues

- Although we can define many operations for a queue, four are the basic: queue, enqueue, dequeue and empty.
- The queue operation creates an empty queue. The following shows the format.

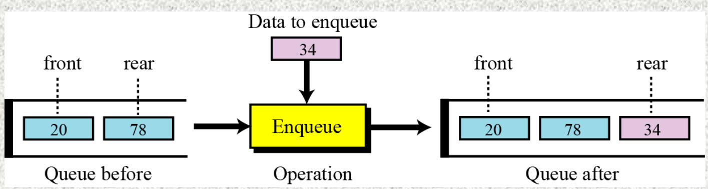
queue (queueName)



The enqueue operation

- The enqueue operation inserts an item at the rear of the queue. The following shows the format.

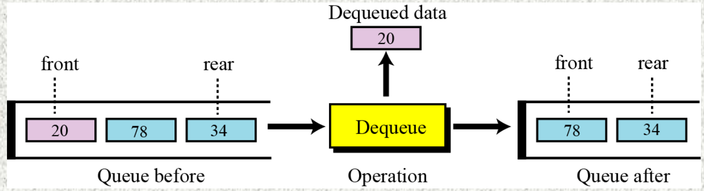
enqueue (queueName, dataItem)



The dequeue operation

- The dequeue operation deletes the item at the front of the queue. The following shows the format.

dequeue (queueName, dataItem)



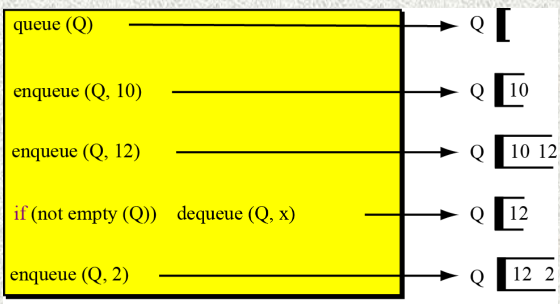
The empty operation

- The empty operation checks the status of the queue. The following shows the format.
- This operation returns true if the queue is empty and false if the queue is not empty.

empty (queueName)

Example

- The following shows a segment of an algorithm that applies the previously defined operations on a queue Q.



An algorithm segment

Queue applications

- Queues are one of the most common of all data processing structures.
- They are found in virtually every operating system and network and in countless other areas
- For example, queues are used in online business applications such as processing customer requests, jobs and orders.
- In a computer system, a queue is needed to process jobs and for system services such as print spools.

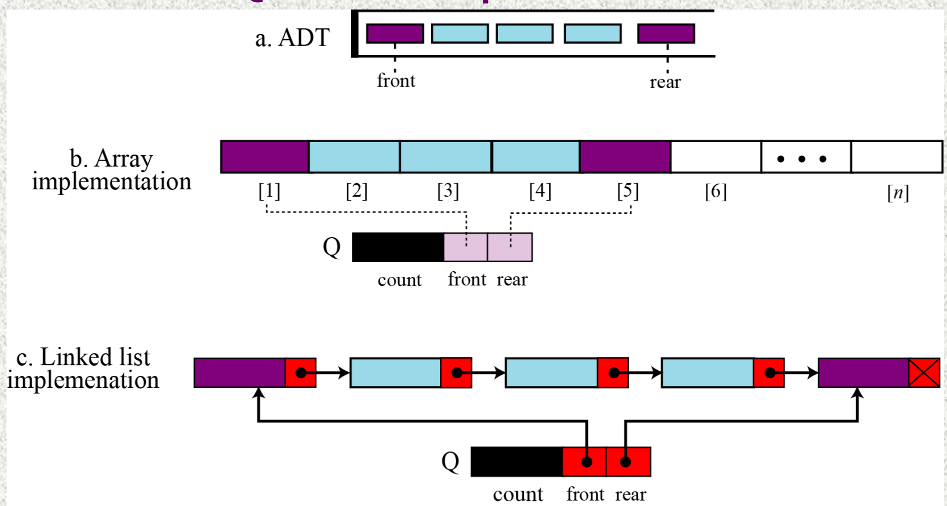
Queue applications

- Another common application of a queue is to adjust and create a balance between a fast producer of data and a slow consumer of data.
- For example, assume that a CPU is connected to a printer. The speed of a printer is not comparable with the speed of a CPU. If the CPU waits for the printer to print some data created by the CPU, the CPU would be idle for a long time.
- The solution is a queue. The CPU creates as many chunks of data as the queue can hold and sends them to the queue. The CPU is now free to do other jobs.
- The chunks are dequeued slowly and printed by the printer. The queue used for this purpose is normally referred to as a spool queue.

Queue implementation

- A queue ADT can be implemented using either an array or a linked list.
- In the array implementation we have a record with three fields. The first field can be used to store information about the queue.
- The linked list implementation is similar: we have an extra node that has the name of the queue. This node also has three fields: a count, a pointer that points to the front element and a pointer that points to the rear element.

Queue implementation



C++ program to implement Queue[using Array]

```
#include <iostream.h>
 #define MAX 50
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
   int choice;
   while (1)
   {
      cout<<"1.Insert element to queue \n”;
      cout<<"2.Delete element from queue \n”;
      cout<<"3.Display all elements of queue \n”;
      cout<<"4.Quit \n”;
      cout<<"Enter your choice : “;
      cin>>"%d", &choice;
```

C++ program to implement Queue[using Array]

```
switch (choice)
      {
          case 1:        insert();
                         break;
          case 2:        delete();
                         break;
          case 3:        display();
                         break;
          case 4:        exit(1);
          default:       cout<<"Wrong choice \n";
      } /*End of switch*/
   } /*End of while*/
} /*End of main()*/
```


C++ program to implement Queue[using Array]

```
insert()
{
   int add_item;
   if (rear == MAX - 1)
   cout<<"Queue Overflow \n";
   else
   {
      if (front == - 1)       /*If queue is initially empty */
      front = 0;
      cout<<"Inset the element in queue : ";
      cin>>"%d", &add_item;
      rear = rear + 1;
      queue_array[rear] = add_item;
   }
} /*End of insert()*/
```

C++ program to implement Queue[using Array]

```
delete()
{
  if (front == - 1 || front > rear)
  {
    cou<<"Queue Underflow \n";
    return ;
  }
  else
  {
    cout<<"Element deleted from queue is : %d\n", queue_array[front];
    front = front + 1;
  }
} /*End of delete() */
```


C++ program to implement Queue[using Array]

```
display()
{
  int i;
  if (front == - 1)
    cout<<"Queue is empty \n”;
  else
  {
    cout<<"Queue is : \n”;
    for (i = front; i <= rear; i++)
      cout<<"%d ", queue_array[i];
    cout<<"\n";
  }
} /*End of display() */
```

Lecture Eleven
Linked List

Introduction to the Linked List ADT

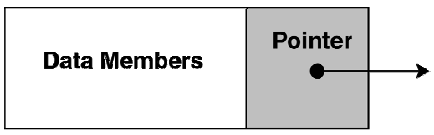
- A linked list is a series of connected nodes, where each node is a data structure.
- A linked list can grow or shrink in size as the program runs.

Advantages of Linked Lists over Arrays

- A linked list can easily grow or shrink in size.
- Insertion and deletion of nodes is quicker with linked lists than with vectors.

The composition of a Linked List

- Each node in a linked list contains one or more members that represent data.
- In addition to the data, each node contains a pointer, which can point to another node.



The composition of a Linked List

- A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.

Declarations

- First you must declare a data structure that will be used for the nodes. For example, the following struct could be used to create a list where each node holds a float:

```
struct ListNode
{
    float value;
    struct ListNode *next;
};
```

Declarations

- The next step is to declare a pointer to serve as the list head, as shown below.

```
ListNode *head;
```

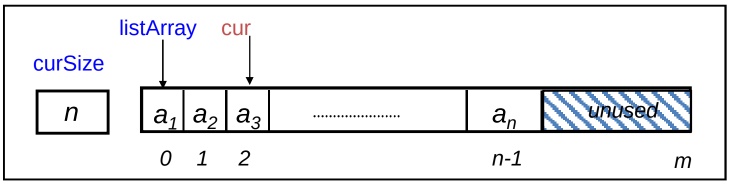
- Once you have declared a node data structure and have created a NULL head pointer, you have an empty linked list.
- The next step is to implement operations with the list.

Linked List Operations

- Typical operations:
  - Creation
  - Insert / remove an element
  - Test for emptiness
  - Find an item/element
  - Current element / next / previous
  - Find k-th element
  - Print the entire list

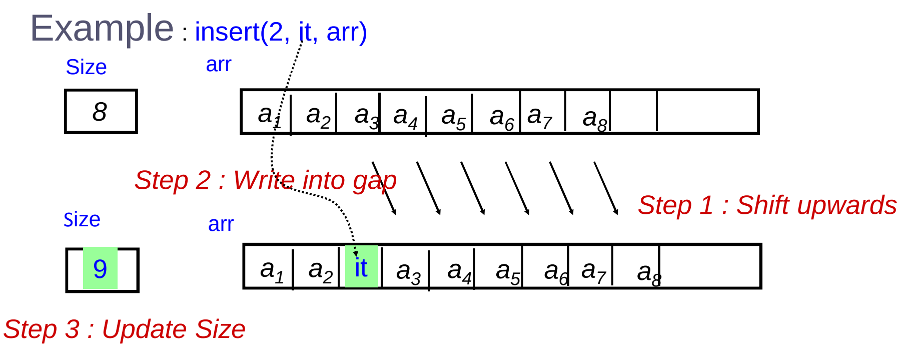
Array-Based List Implementation

- One simple implementation is to use arrays
  - A sequence of n-elements
- Maximum size is anticipated a priori.
- Internal variables:
  - Maximum size maxSize (m)
  - Current size curSize (n)
  - Current index cur
  - Array of elements listArray



Inserting Into an Array

- While retrieval is very fast, insertion and deletion are very slow
  - Insert has to shift upwards to create gap



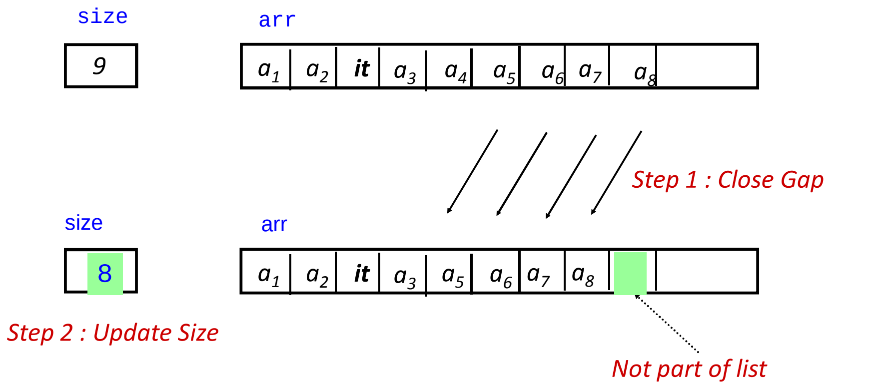
Coding

```
typedef struct {
    int arr[MAX];
    int max;
    int size;
} LIST

void insert(int j, int it, LIST *pl)
  {

       int i;

       for (i=pl->size; i>=j; i=i-1)
                                 // Step 1: Create gap
         { pl->arr[i+1]= pl->arr[i]; };

       pl->arr[j]= it;        // Step 2: Write to gap

       pl->size = pl->size + 1; // Step 3: Update size
  }
```


Deleting from an Array

- Delete has to shift downwards to close gap of deleted item

Example: `deleteItem(4, arr)`

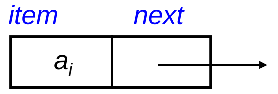


Coding

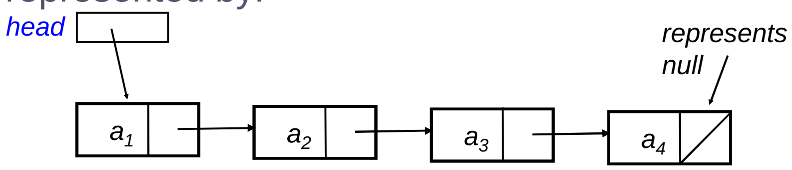
```
void delete(int j, LIST *pl)
{
  for (i=j+1; i<=pl->size; i=i+1)
                      // Step1: Close gap
    { pl->arr[i-i]=pl->arr[i]; };
                      // Step 2: Update
   size
    pl->size = pl->size - 1;
  }
```


Linked List Approach

- Main problem of array is the slow deletion/insertion since it has to shift items in its contiguous memory
- Solution: linked list where items need not be contiguous with nodes of the form



- Sequence (list) of four items < a1,a2 ,a3 ,a4 > can be represented by:



Pointer-Based Linked Lists

- A node in a linked list is usually a struct

```
struct Node
{ int item
  Node *next;
}; //end struct
```

- A node is dynamically allocated

```
Node *p;
p = malloc(sizeof(Node));
```

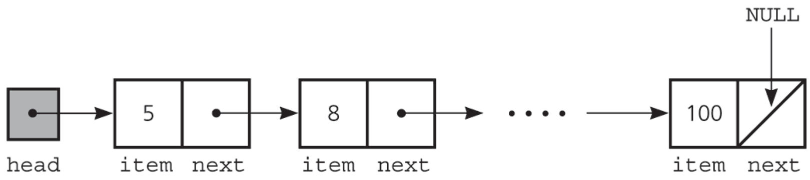

Pointer-Based Linked Lists

- The head pointer points to the first node in a linked list
- If head is `NULL`, the linked list is empty

```
head=NULL
```

- `head=malloc(sizeof(Node))`

A Sample Linked List



Traverse a Linked List

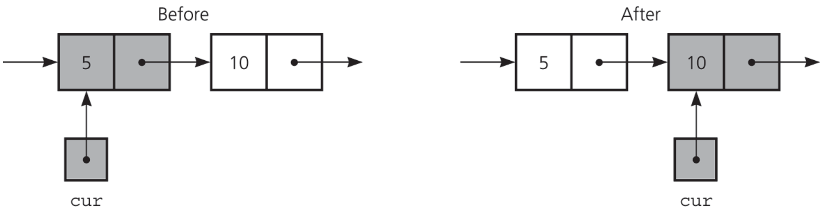
- Reference a node member with the -> operator

```
p->item;
```

- A traverse operation visits each node in the linked list
  - A pointer variable cur keeps track of the current node

```
for (Node *cur = head;
     cur != NULL; cur = cur->next)
     x = cur->item;
```

Traverse a Linked List

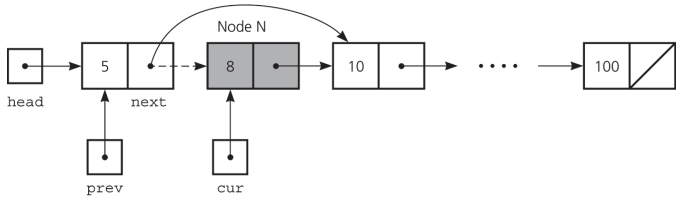


The effect of the assignment *cur = cur->next*

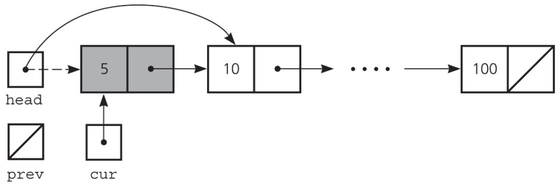
Delete a Node from a Linked List

- Deleting an interior/last node
```
prev->next=cur->next;
```
- Deleting the first node
```
head=head->next;
```
- Return deleted node to system
```
cur->next = NULL;
free(cur);
cur=NULL;
```

Delete a Node from a Linked List



Deleting a node from a linked list

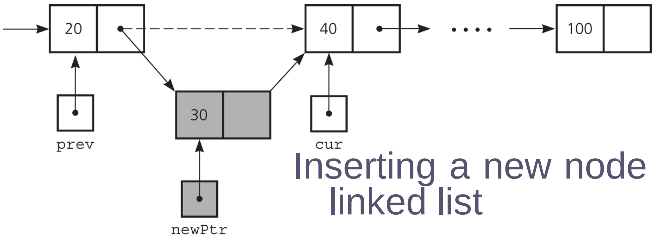


Deleting the first node

Insert a Node into a Linked List

- To insert a node between two nodes

```
newPtr->next = cur;
prev->next = newPtr;
```

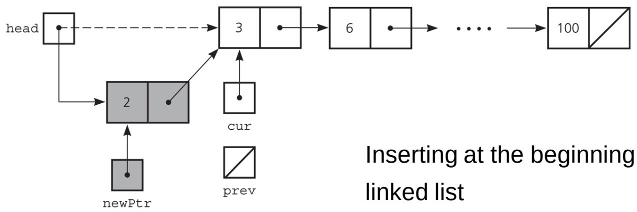


Inserting a new node into a linked list

Insert a Node into a Linked List

- To insert a node at the beginning of a linked list

```
newPtr->next = head;
head = newPtr;
```

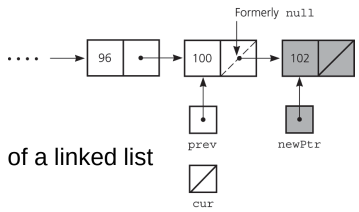


Inserting at the beginning of a linked list

Insert a Node into a Linked List

- Inserting at the end of a linked list is not a special case if `cur` is `NULL`

```
newPtr->next = cur;
prev->next = newPtr;
```



Inserting at the end of a linked list

Look up

```
BOOLEAN lookup (int x, Node *L)
  {  if (L == NULL)
 return FALSE
     else if (x == L->item)
    return TRUE
     else
 return lookup(x, L-next);
  }
```


A Practical C Linked List Example

```
struct test_struct
{
    int val;
    struct test_struct *next;
};
struct test_struct *head = NULL;
struct test_struct *curr = NULL;
```

```
struct test_struct* create_list(int val)
{
    cout<<"\n creating list with headnode as
 [%d]\n",val;
    struct test_struct *ptr = (struct
 test_struct*)malloc(sizeof(struct test_struct));
    if(NULL == ptr)
    {
        cout<<"\n Node creation failed \n”;
        return NULL;
    }
    ptr->val = val;
    ptr->next = NULL;
    head = curr = ptr;
    return ptr;
}
```



```
struct test_struct* add_to_list(int val, bool add_to_end)
{   if(NULL == head)
    {    return (create_list(val));    }
    if(add_to_end)
        cout<<"\n Adding node to end of list with value
  [%d]\n“<<val;
    else
        cout<<"\n Adding node to beginning of list with value
  [%d]\n“<<val;
    struct test_struct *ptr = (struct
  test_struct*)malloc(sizeof(struct test_struct));
    if(NULL == ptr)
    {   cout<<"\n Node creation failed \n”;
        return NULL;    }
    ptr->val = val;
    ptr->next = NULL;
    if(add_to_end)
    {   curr->next = ptr;
        curr = ptr;    }
    else    {  ptr->next = head;
               head = ptr;  }
    return ptr; }
```

```
struct test_struct* search_in_list(int val, struct test_struct
  **prev)
{   struct test_struct *ptr = head;
    struct test_struct *tmp = NULL;
    bool found = false;
    cout<<"\n Searching the list for value [%d] \n“<<val;
    while(ptr != NULL)
    {   if(ptr->val == val)
        {   found = true;
            break;  }
        else
        {   tmp = ptr;
            ptr = ptr->next;}
    }
    if(true == found)
    {   if(prev)
            *prev = tmp;
        return ptr;   }
    else
    { return NULL; }
}
```



```
int delete_from_list(int val)
{   struct test_struct *prev = NULL;
    struct test_struct *del = NULL;
    cout<<"\n Deleting value [%d] from list\n“<<val;
    del = search_in_list(val,&prev);
    if(del == NULL)
       return -1;
    else
    {   if(prev != NULL)
            prev->next = del->next;
        if(del == curr)
           curr = prev;
        else if(del == head)
            head = del->next;
     }
    free(del);
    del = NULL;
    return 0;
}
```

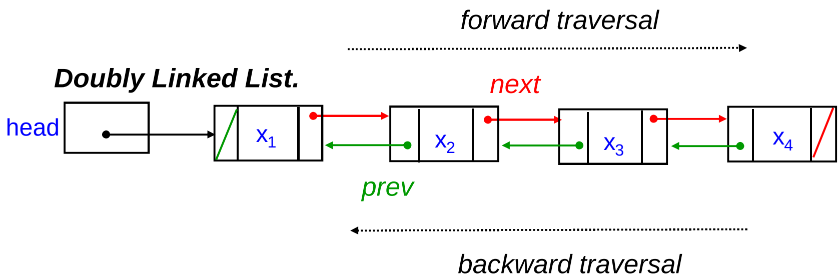
```
void print_list(void)
{
    struct test_struct *ptr = head;
    cout<<“\n -------Printing list
 Start------- \n”;
    while(ptr != NULL)
    {
        cout<<"\n [%d] \n“<<ptr->val;
        ptr = ptr->next;
    }
    cout<<"\n -------Printing list End-
 ------ \n”;
    return;
}
```



```
int main(void)
{   int i = 0, ret = 0;
    struct test_struct *ptr = NULL;
    print_list();
    for(i = 5; i<10; i++)
        add_to_list(i,true);
    print_list();
    for(i = 4; i>0; i--)
        add_to_list(i,false);
    print_list();
    for(i = 1; i<10; i += 4)
    {   ptr = search_in_list(i, NULL);
        if(NULL == ptr)
           cout<<"\n Search [val = %d] failed, no such element found\n“<<i;
        else
           cout<<"\n Search passed [val = %d]\n“<<ptr->val;
        print_list();
        ret = delete_from_list(i);
        if(ret != 0)
            cout<<"\n delete [val = %d] failed, no such element found\n“<<i;
        else
            cout<<"\n delete [val = %d]  passed \n“<<i;
        print_list();
    return 0;
}
```

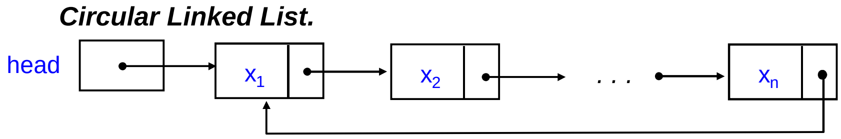
Doubly Liked Lists

- Frequently, we need to traverse a sequence in BOTH directions efficiently
- Solution : Use doubly-linked list where each node has two pointers



Circular Linked Lists

- May need to cycle through a list repeatedly, e.g. round robin system for a shared resource
- Solution : Have the last node point to the first node

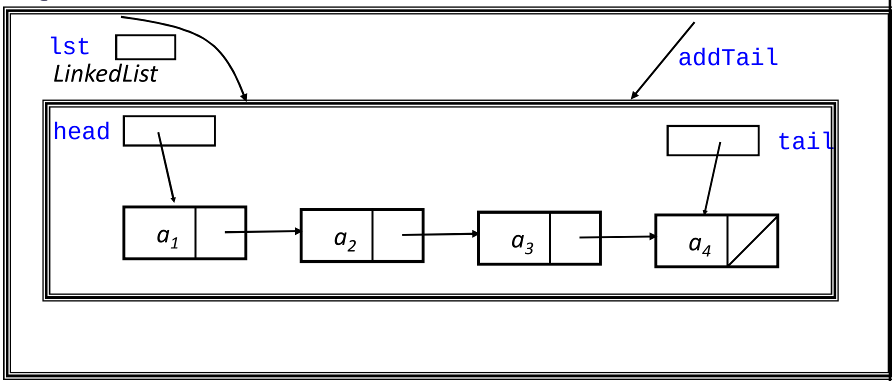


Lecture Thirteen
Queue Implementation by Linked List

Implementation of Queue (Linked List)

- Can use LinkedListItr as underlying implementation of Queues

Queue



Implementation by Linked Lists

```
/* C Program to Implement Queue Data Structure using
Linked List */
#include <iostream.h>
#include <stdlib.h>
struct node
{   int info;
    struct node *ptr;
}*front,*rear,*temp,*front1;
int frontelement();
void enq(int data);
void deq();
void empty();
void display();
void create();
void queuesize();
int count = 0;
```



```
void main()
{
    int no, ch, e;

    cout<<"\n 1 – Enque”;
    cout<<"\n 2 – Deque”;
    cout<<"\n 3 - Front element”;
    cout<<"\n 4 – Empty”;
    cout<<"\n 5 – Exit”;
    cout<<"\n 6 – Display”;
    cout<<"\n 7 - Queue size”;
    create();
while (1)
  {
    cout<<"\n Enter choice : “;
    cin>>"%d", &ch;
    switch (ch)
    {
    case 1:
      cout<<"Enter data : “;
      cin>>no;
      enq(no);
      break;
```

```
 case 2:    deq();
            break;
 case 3:    e = frontelement();
            if (e != 0)
                cout<<"Front element : %d“<< e;
            else
                cout<<"\n No front element in Queue as
queue is empty”;
            break;
case 4:     empty();
            break;
case 5:     exit(0);
case 6:     display();
            break;
case 7:     queuesize();
            break;
Default:    cout<<"Wrong choice, Please enter correct
choice  “;
            break;   }
}   }
```



```
 /* Create an empty queue */
void create()
{
    front = rear = NULL;
}

/* Returns queue size */
void queuesize()
{
    cout<<"\n Queue size : %d“<< count;
}
```

```
 /* Enqueing the queue */
void enq(int data)
{   if (rear == NULL)
    { rear = (struct node
*)malloc(1*sizeof(struct node));
        rear->ptr = NULL;
        rear->info = data;
        front = rear;    }
    else
    {   temp=(struct node
*)malloc(1*sizeof(struct node));
        rear->ptr = temp;
        temp->info = data;
        temp->ptr = NULL;
        rear = temp;    }
    count++;
}
```



```
 /* Displaying the queue elements */
void display()
{
    front1 = front;
    if ((front1 == NULL) && (rear == NULL))
    {
        cout<<"Queue is empty”;
        return;
    }
    while (front1 != rear)
    {
        cout<<"%d ", front1->info;
        front1 = front1->ptr;
    }
    if (front1 == rear)
        cout<<"%d“<< front1->info;
}
```

```
 /* Dequeing the queue */
void deq()
{   front1 = front;
    if (front1 == NULL)
    {  cout<<"\n Error: Trying to display elements from
empty queue”;
        return;    }
    else
        if (front1->ptr != NULL)
        {   front1 = front1->ptr;
            cout<<"\n Dequed value : %d“<< front->info;
            free(front);
            front = front1;      }
        else
        {  cout<<"\n Dequed value : %d“<< front->info;
            free(front);
            front = NULL;
            rear = NULL;        }
        count--;
}
```

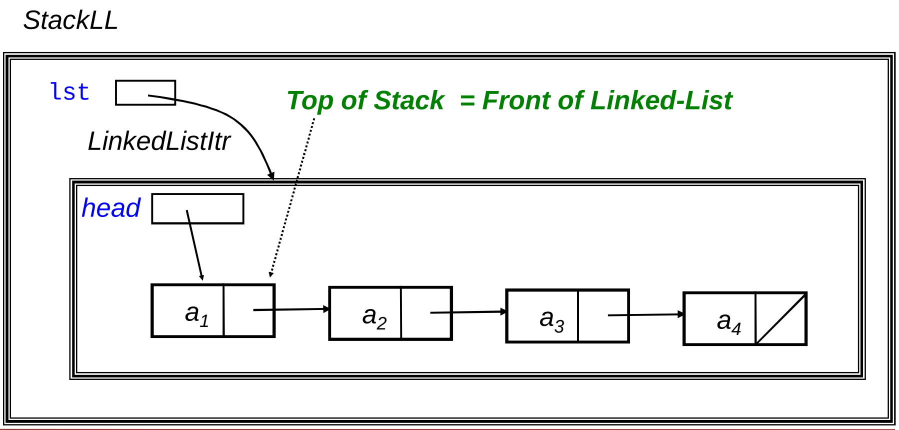


```
 /* Returns the front element of queue */
int frontelement()
{
    if ((front != NULL) && (rear != NULL))
        return(front->info);
    else
        return 0;
}


/* Display if queue is empty or not */
void empty()
{
     if ((front == NULL) && (rear == NULL))
        cout<<"\n Queue empty”;
    else
       cout<<"Queue not empty”;
}
```

Lecture Twelve
Stack Implementation by Linked List

Implementation by Linked Lists

- Can use a Linked List as implementation of stack



Implementation by Linked Lists

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
#include<stdlib.h>
#include<alloc.h>

void Push(int, node **);
void Display(node **);
int Pop(node **);
int Sempty(node *);

typedef struct stack {
   int data;
   struct stack *next;
} node;
```



```
void main() {
   node *top;
   int data, item, choice;
   char ans, ch;
   clrscr();
   top = NULL;
   cout<<"\nStack Using Linked List : nn”;
   do {
      cout<<"\n\n The main menu”;
      cout<<"\n1.Push \n2.Pop \n3.Display \n4.Exit”;
      cout<<"\n Enter Your Choice”;
      cin>>"%d", &choice;
       switch (choice) {
      case 1:  cout<<"\nEnter the data”;
               cin>>"%d", &data;
                      Push(data, &top);
                      break;
      case 2: if (Sempty(top))
                     cout<<"\nStack underflow!”;
               else {
                     item = Pop(&top);
                     cout<<"\nThe popped node is%d", item;}
              break;
```

```
case 3:    Display(&top);
           break;
case 4:    cout<<"\nDo You want To Quit?(y/n)”;
           ch = getche();
           if (ch == 'y')
                        exit(0);
           else
                  break;
     cout<<"\nDo you want to continue?”;
     ans = getche();
     getch();
     clrscr();
  } while (ans == 'Y' || ans == 'y');
  getch();
}
```



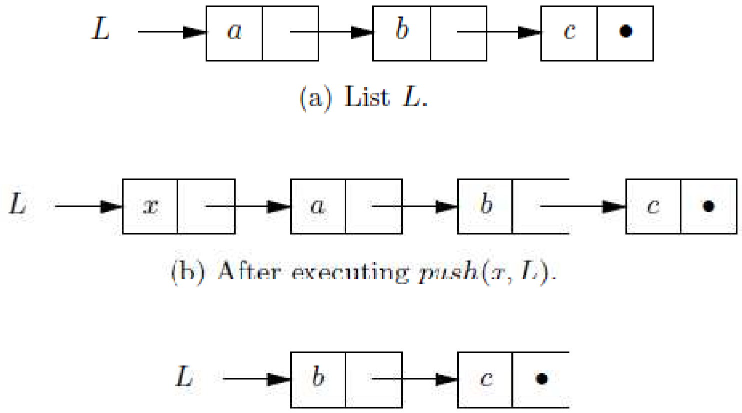
```
 void Push(int Item, node **top) {
   node *New;
   node * get_node(int);
   New = get_node(Item);
   New->next = *top;
   *top = New;
}

node * get_node(int item) {
   node * temp;
   temp = (node *) malloc(sizeof(node));
   if (temp == NULL)
      cout<<"\nMemory Cannot be allocated”;
   temp->data = item;
   temp->next = NULL;
   return (temp);
}
```

```
int Sempty(node *temp) {
  if (temp == NULL)
     return 1;
  else
     return 0;
}

int Pop(node **top) {
   int item;
   node *temp;
   item = (*top)->data;
   temp = *top;
   *top = (*top)->next;
   free(temp);
   return (item);
}
```



```
void Display(node **head) {
  node *temp;
  temp = *head;
  if (Sempty(temp))
     cout<<"\nThe stack is empty!”;
  else {
     while (temp != NULL) {
        cout<<"%d\n", temp->data;
        temp = temp->next;
     }
  }
  getch();
}
```

Effects



(a) List L.

(b) After executing $push(x, L)$.

(c) After executing $pop(L, x)$ on list L of (a).

Applications

- Many application areas use stacks:
  - line editing
  - bracket matching
  - postfix calculation
  - function call stack

Line Editing

- A line editor would place characters read into a buffer but may use a backspace symbol (denoted by ←) to do error correction
- *Refined Task*
  - read in a line
  - correct the errors via backspace
  - print the corrected line in reverse

Input : `abc_defgh←2klpqr←←wxyz`

Corrected Input : `abc_defg2klpwxyz`

Reversed Output : `zyxwplk2gfed_cba`

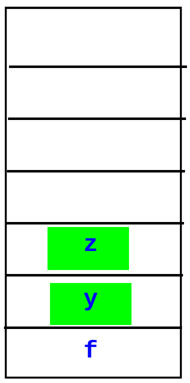
The Procedure

- Initialize a new stack
- For each character read:
  - if it is a backspace, *pop out last char entered*
  - if not a backspace, *push the char into stack*
- To print in reverse, pop out each char for output

Input : `fgh←r←←yz`

Corrected Input : `fyz`

Reversed Output : `zyf`



Stack

Bracket Matching Problem

- Ensures that pairs of brackets are properly matched
  - An Example: `{a,(b+f[4])*3,d+f[5]}`
  - Bad Examples:

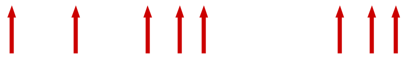
```
(..)..)        // too many closing brackets
(..(..)        // too many open brackets
[..(..]..)     // mismatched brackets
```

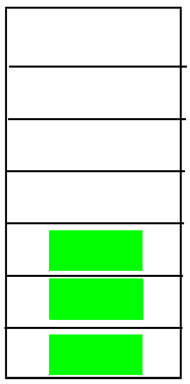

Informal Procedure

Initialize the stack to empty
For every char read
  if open bracket then *push onto stack*
  if close bracket, then
    return & remove most recent item
      from *the stack*
    if doesn't match then *flag error*
  if non-bracket, *skip the char read*

Example

`{a,(b+f[4])*3,d+f[5]}`



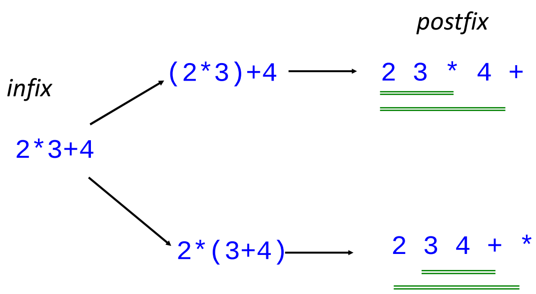


Stack

Postfix Calculator

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of a stack.

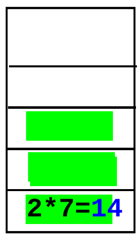
Infix - arg1 **op** arg2
Prefix - **op** arg1 arg2
Postfix - arg1 arg2 **op**



Informal Procedure

Initialise stack S
For each item read.
 If it is an operand,
  push on the stack
 If it is an operator,
  pop arguments from stack;
  perform operation;
  push result onto the stack

Expr	
2	push(S, 2)
3	push(S, 3)
4	push(S, 4)
+	arg2=topAndPop(S) arg1=topAndPop(S) push(S, arg1+arg2)
*	arg2=topAndPop(S) arg1=topAndPop(S) push(S, arg1*arg2)



Stack

Lecture Fourteen
Infix, Prefix and Postfix Expressions

Algebraic Expression

- An algebraic expression is a legal combination of operands and the operators.
- Operand is the quantity (unit of data) on which a mathematical operation is performed.
- Operand may be a variable like x, y, z or a constant like 5, 4,0,9,1 etc.
- Operator is a symbol which signifies a mathematical or logical operation between the operands. Example of familiar operators include +,-,*, /, ^
- An example of expression as x+y*z.

Infix, Postfix and Prefix Expressions

- INFIX: the expressions in which operands surround the operator, e.g. x+y, 6*3 etc this way of writing the Expressions is called infix notation.
- POSTFIX: Postfix notation are also Known as Reverse Polish Notation (RPN). They are different from the infix and prefix notations in the sense that in the postfix notation, operator comes after the operands, e.g. xy+, xyz+* etc.
- PREFIX: Prefix notation also Known as Polish notation. In the prefix notation, operator comes before the operands, e.g. +xy, *+xyz etc.

Operator Priorities

- How do you figure out the operands of an operator?
  - a + b * c
  - a * b + c / d
- This is done by assigning operator priorities.
  - priority(*) = priority(/) > priority(+) = priority(-)
- When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.
  - a + b - c
  - a * b / c / d

Delimiters

- Sub expression within delimiters is treated as a single operand, independent from the remainder of the expression.
  - (a + b) * (c – d) / (e – f)

WHY??

- Why to use PREFIX and POSTFIX notations when we have simple INFIX notation?
- INFIX notations are not as simple as they seem specially while evaluating them. To evaluate an infix expression we need to consider Operators' Priority and Associative property
  - E.g. expression 3+5*4 evaluate to 32 i.e. (3+5)*4 or to 23 i.e. 3+(5*4).
- To solve this problem Precedence or Priority of the operators were defined. Operator precedence governs evaluation order. An operator with higher precedence is applied before an operator with lower precedence.

Infix Expression is Hard To Parse

- Need operator priorities, tie breaker, and delimiters.
- This makes computer evaluation more difficult than is necessary.
- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- Both prefix and postfix notations have an advantage over infix that while evaluating an expression in prefix or postfix form we need not consider the Priority and Associative property (order of brackets).
  - E.g. x/y*z becomes */xyz in prefix and xy/z* in postfix. Both prefix and postfix notations make Expression Evaluation a lot easier.
- So it is easier to evaluate expressions that are in these forms.

Examples of infix to prefix and post fix

Infix	Postfix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Example: postfix expressions

- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.
  - infix: 2+5 postfix: 2 5 +
- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

Suppose that we would like to rewrite A+B*C in postfix

- Applying the rules of precedence, we obtained

A+B*C

A+(B*C) Parentheses for emphasis

A+(BC*) Convert the multiplication,Let D=BC*

A+D Convert the addition

A(D)+

ABC*+ Postfix Form

Postfix Examples

Infix	Postfix	Evaluation
2 - 3 * 4 + 5	2 3 4 * - 5 +	-5
(2 - 3) * (4 + 5)	2 3 - 4 5 + *	-9
2- (3 * 4 +5)	2 3 4 * 5 + -	-15

- Why ? No brackets necessary !

Algorithm for Infix to Postfix

- Examine the next element in the input.
- If it is operand, output it.
- If it is opening parenthesis, push it on stack.
- If it is an operator, then
  - i) If stack is empty, push operator on stack.
  - ii) If the top of stack is opening parenthesis, push operator on stack
  - iii) If it has higher priority than the top of stack, push operator on stack.
  - iv) Else pop the operator from the stack and output it, repeat step 4
- If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
- If there is more input go to step 1
- If there is no more input, pop the remaining operators to output.

Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form,

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

Example

- (5 + 6) * 9 +10

will be

- 5 6 + 9 * 10 +

Evaluation a postfix expression

- Each operator in a postfix string refers to the previous two operands in the string.
- Suppose that each time we read an operand we push it into a stack. When we reach an operator, its operands will then be top two elements on the stack
- We can then pop these two elements, perform the indicated operation on them, and push the result on the stack.
- So that it will be available for use as an operand of the next operator.

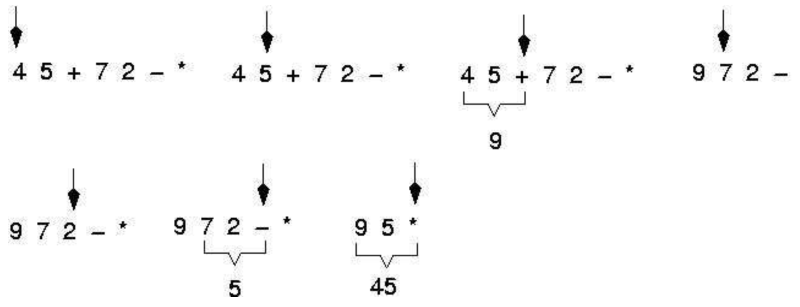
Evaluating Postfix Notation

- Use a stack to evaluate an expression in postfix notation.
- The postfix expression to be evaluated is scanned from left to right.
- Variables or constants are pushed onto the stack.
- When an operator is encountered, the indicated action is performed using the top elements of the stack, and the result replaces the operands on the stack.

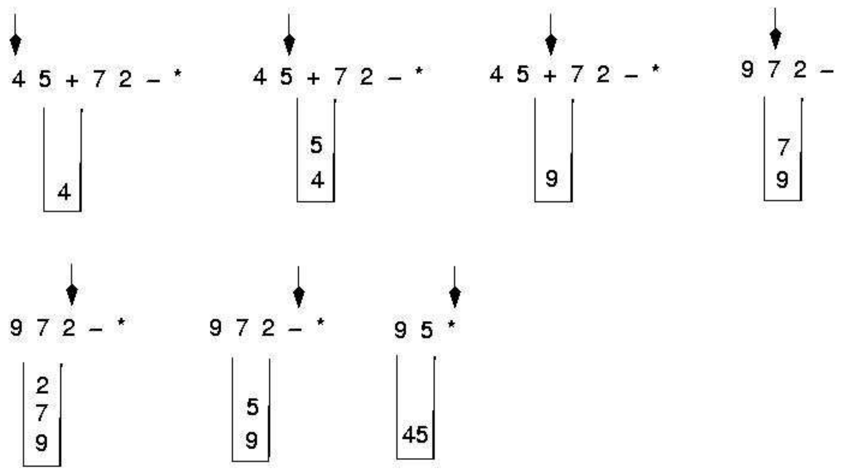
Evaluating a postfix expression

- Initialise an empty stack
- While token remain in the input stream
  - Read next token
  - If token is a number, push it into the stack
  - Else, if token is an operator, pop top two tokens off the stack, apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Example: Postfix Expressions



Postfix expressions: Algorithm using stacks (cont.)



Algorithm for evaluating a postfix expression (Cond.)

```
WHILE more input items exist
{
    If symb is an operand
        then push (opndstk,symb)
    else  //symbol is an operator
    {
        Opnd1=pop(opndstk);
        Opnd2=pop(opndnstk);
        Value = result of applying symb to opnd1 & opnd2
        Push(opndstk,value);
    }          //End of else
} // end while
Result = pop (opndstk);
```


Question : Evaluate the following expression in postfix : 623+-382/+*2^3+

Final answer is

- 49
- 51
- 52
- 7
- None of these

Evaluate- 623+-382/+*2^3+

Symbol	opnd1	opnd2	value	opndstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3

Evaluate- 623+-382/+*2^3+

Symbol	opnd1	opnd2	value	opndstk
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
^	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Evaluating a Postfix Expression in C++

```
#include "stackd.h"
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#define MAX_SIZE_EXPRESSION 100
int main(void);
int evaluate_postfix(char *expression,
double *result);
int evaluate_operator(int operator_symbol,
double first_operand, double second_operand,
double *result);
```


Evaluating a Postfix Expression in C++

```
#include "stackd.h"
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#define MAX_SIZE_EXPRESSION 100
int main(void);
int evaluate_postfix(char *expression,
double *result);
int evaluate_operator(int operator_symbol,
double first_operand, double second_operand,
double *result);
```

Evaluating a Postfix Expression in C++

```
int main(void)
{
char expression[MAX_SIZE_EXPRESSION];
int position = 0;
double value;
do
{
expression[position] = getchar();
position++;
} while (expression[position - 1] != '\n'
&& position < MAX_SIZE_EXPRESSION);
```


Evaluating a Postfix Expression in C++

```
expression[position - 1] = '\0';
if (!evaluate_postfix(expresssion, &value))
{
return 1;
}
printf("Expression \"%s\" evaluates to %g.\n",
expresssion, value);
return 0;
}
```

Evaluating a Postfix Expression in C++

```
int evaluate_postfix(char *expression,
double *result)
{
int position;
stackd operand_stack;
static char *digits = "0123456789";
assert(NULL != result && NULL != expression);
if (!stackd_init(&operand_stack, 0))
return FALSE;
```


Evaluating a Postfix Expression in C++

```
for (position = 0;
'\0' != expression[position]; position++)
{
if (NULL !=
strchr(digits, expression[position]))
{
if (!stackd_push(&operand_stack,
(double)(strchr(digits,
expression[position]) - &digits[0])))
break;
}
```

Evaluating a Postfix Expression in C++

```
else
{
double first_operand, second_operand;
double value;
if (stackd_empty(&operand_stack))
break;
second_operand =
stackd_pop(&operand_stack);
if (stackd_empty(&operand_stack))
break;
first_operand =
stackd_pop(&operand_stack);
```


Evaluating a Postfix Expression in C++

```
if (!evaluate_operator(
expression[position], first_operand,
second_operand, &value))
break;
if (!stackd_push(&operand_stack, value))
break;
} /* end else */
} /* end for */
```

Evaluating a Postfix Expression in C++

```
if ('\0' != expression[position]
|| stackd_empty(&operand_stack))
{
printf("syntax error.\n");
stackd_deinit(&operand_stack);
return FALSE;
}
*result = stackd_pop(&operand_stack);
```


Evaluating a Postfix Expression in C++

```
int evaluate_operator(int operator_symbol,
double first_operand, double second_operand,
double *result)
{
assert(NULL != result);
switch (operator_symbol)
{
case '+':
*result = first_operand + second_operand;
return TRUE;
```

Evaluating a Postfix Expression in C++

```
case '-':
*result = first_operand - second_operand;
return TRUE;
case '*':
*result = first_operand * second_operand;
return TRUE;
case '/':
if (0.0 == second_operand)
{
return FALSE;
}
else
{
*result = first_operand
/ second_operand;
return TRUE;
}
```

Saturday, April 18, 2015 Data Structure 35

Evaluating a Postfix Expression in C++

```
case '^':
if (first_operand <= 0.0)
{
return FALSE;
}
else
{
*result = pow(first_operand,
second_operand);
return TRUE;
}
```

Saturday, April 18, 2015 Data Structure 36

Evaluating a Postfix Expression in C++

```
default:
return FALSE;
} /* end switch */
return FALSE; /* unreachable code */
}
```

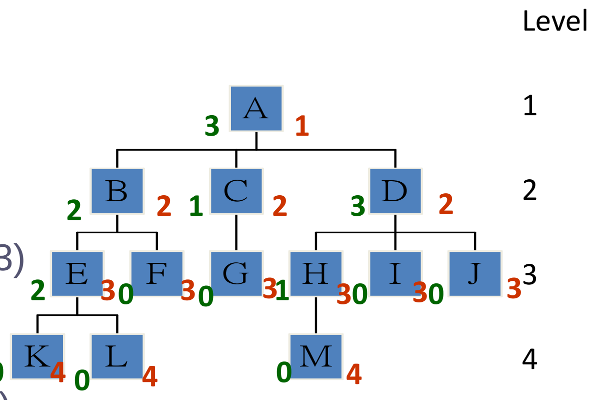
Lecture Fifteen
Tree and Binary Search Tree

Definition of Tree

- A tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root.
  - The remaining nodes are partitioned into n>=0 disjoint sets T1, ..., Tn, where each of these sets is a tree.
  - We call T1, ..., Tn the subtrees of the root.

Level and Depth

- node (13)
- degree of a node
- leaf (terminal)
- Non terminal
- parent
- children
- sibling
- degree of a tree (3)
- ancestor
- level of a node
- height of a tree (4)

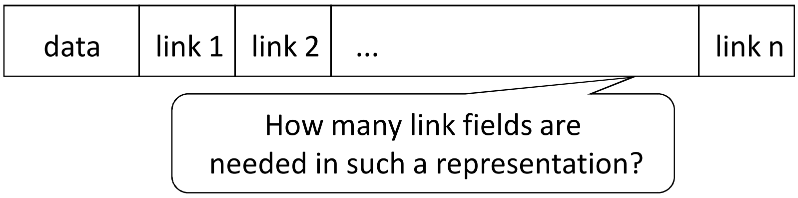


Terminology

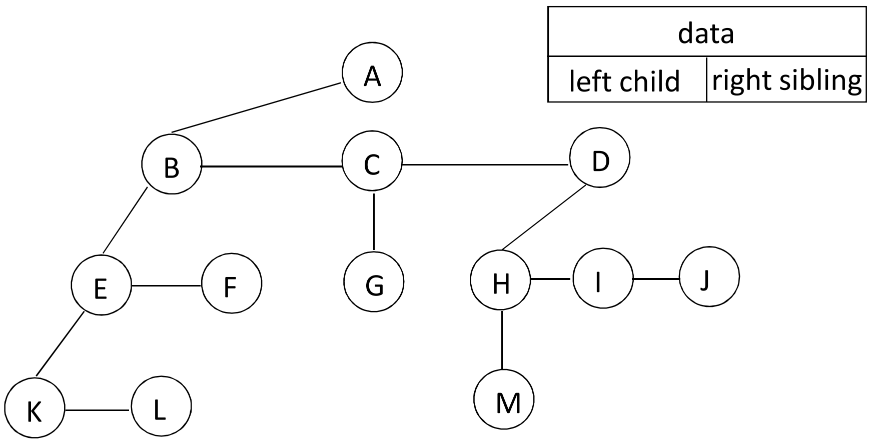
- The degree of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the parent of the roots of the subtrees.
- The roots of these subtrees are the children of the node.
- Children of the same parent are siblings.
- The ancestors of a node are all the nodes along the path from the root to the node.

Representation of Trees

- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of sub-trees



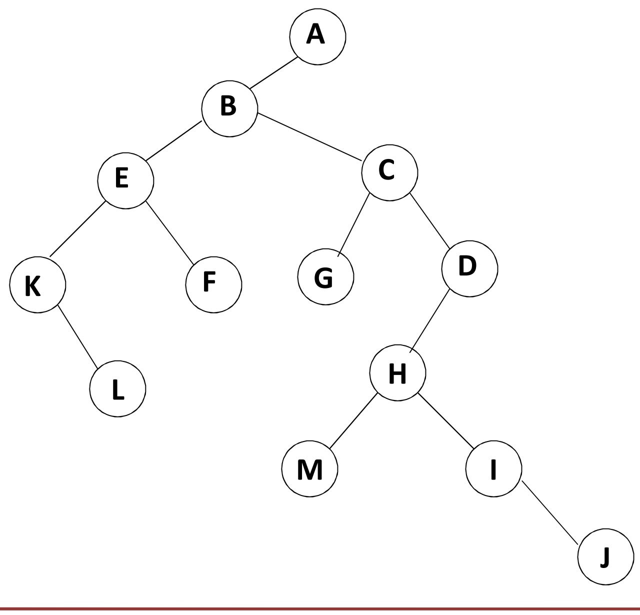
Left Child - Right Sibling



Binary Trees

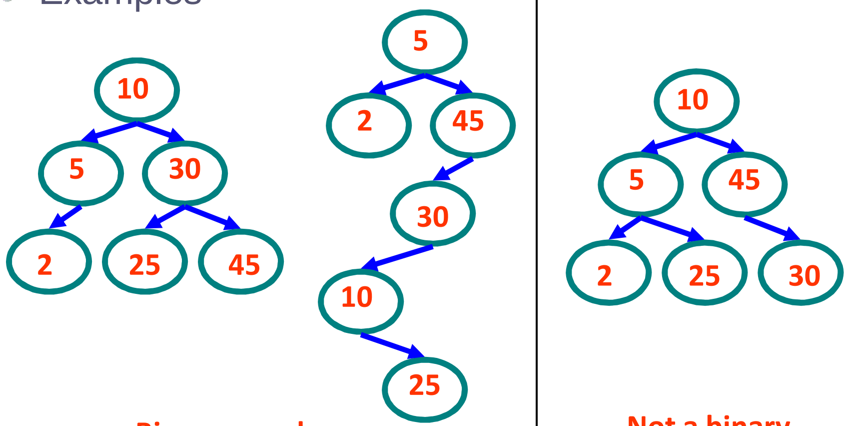
- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.

Left child-right child tree representation of a tree



Binary Search Trees

- Examples

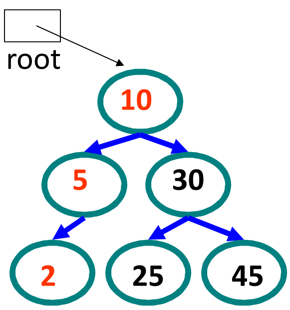


Binary search trees

Not a binary search tree

Example Binary Searches

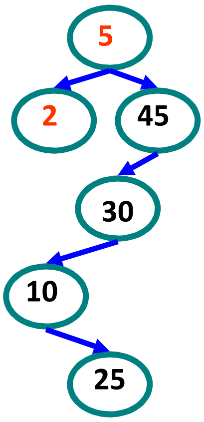
- Find (root, 2)



10 > 2, left

5 > 2, left

2 = 2, found

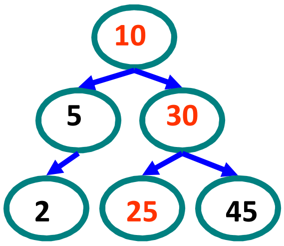


5 > 2, left

2 = 2, found

Example Binary Searches

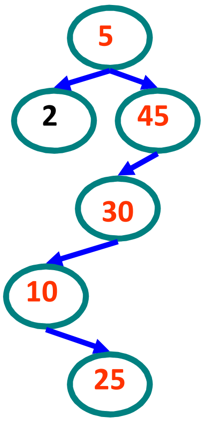
- Find (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

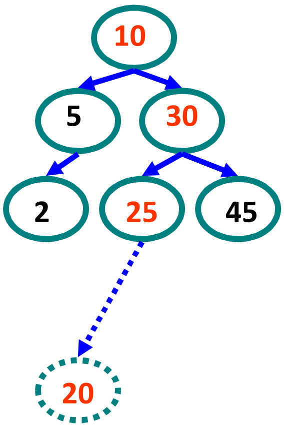
30 > 25, left

10 < 25, right

25 = 25, found

Example Insertion

- Insert (20)



10 < 20, right

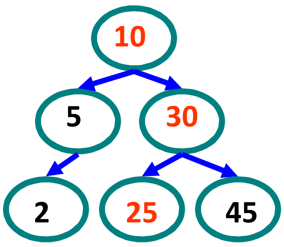
30 > 20, left

25 > 20, left

Insert 20 on left

Example Deletion (Leaf)

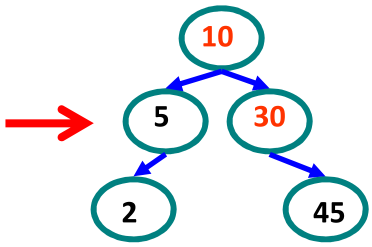
- Delete (25)



10 < 25, right

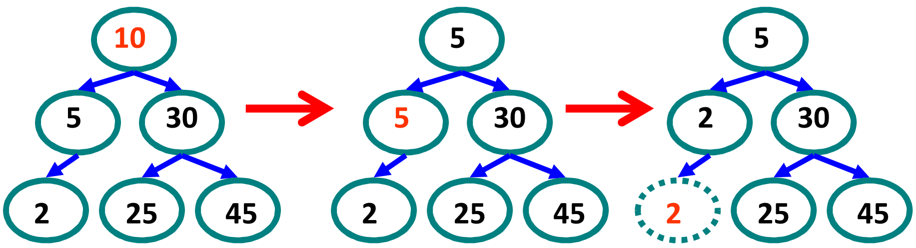
30 > 25, left

25 = 25, delete



Example Deletion (Internal Node)

- Delete (10)



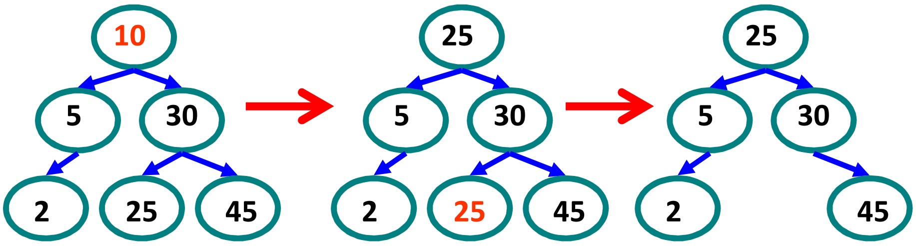
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

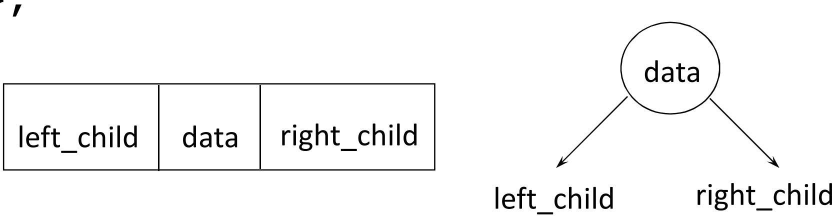
- Delete (10)



Replacing 10 with smallest value in right subtree | **Deleting leaf** | **Resulting tree**

Linked Representation

```
typedef struct node *tree_pointer;
typedef struct node {
 int data;
 tree_pointer left_child, right_child;
};
```



Binary Tree Traversals

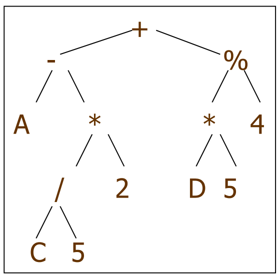
- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal
  - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LVR, LRV, VLR
  - inorder, postorder, preorder

Parse Trees

- Expressions, programs, etc can be represented by tree structures
  - E.g. Arithmetic Expression Tree
  - A-(C/5 * 2) + (D*5 % 4)

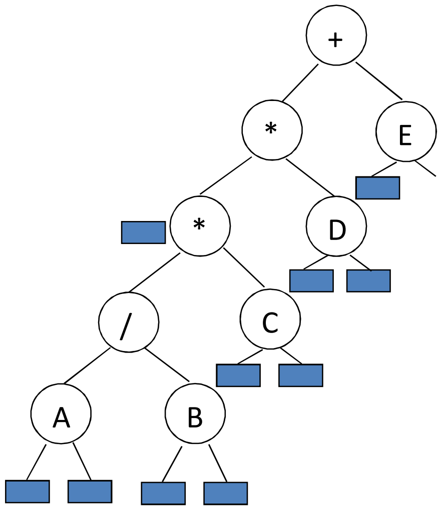
Output preorder: + - A * / C 5 2 % * D 5 4

Output postorder: A C 5 / 2 * - D 5 * 4 % +



Arithmetic Expression Using BT

inorder traversal
A / B * C * D + E
infix expression
preorder traversal
+ * * / A B C D E
prefix expression
postorder traversal
A B / C * D * E +
postfix expression
level order traversal
+ * E * D / C A B



Inorder Traversal

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left_child);
        cout<<ptr->data;
        indorder(ptr-
  >right_child);
    }
}
```

A / B * C * D + E

Preorder Traversal

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        cout<<ptr->data;
        preorder(ptr->left_child);
        predorder(ptr->right_child);
    }
}
```

+ * * / A B C D E

Postorder Traversal

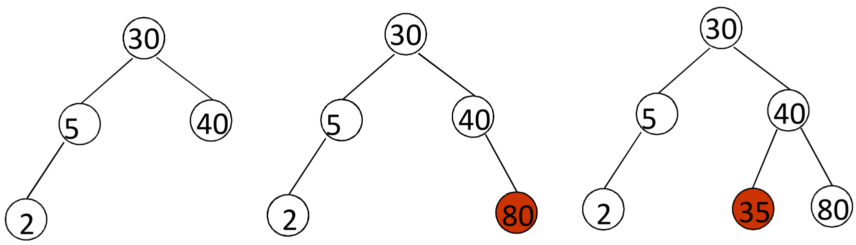
```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left_child);
        postdorder(ptr->right_child);
        cout<<ptr->data;
    }
}
```

A B / C * D * E +

Trace Operations of Inorder Traversal

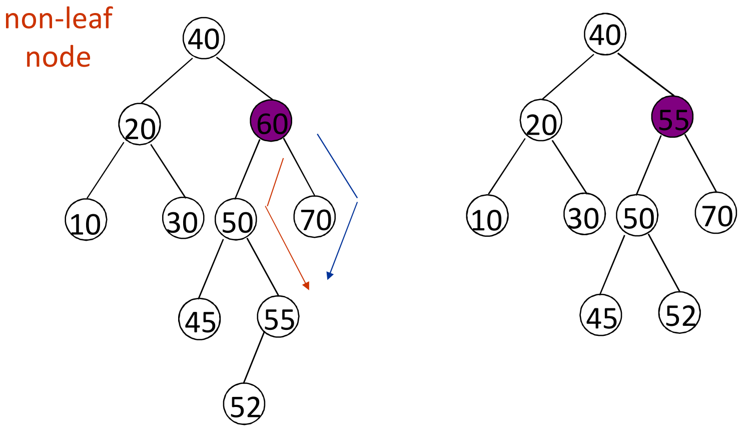
Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	cout
4	/		13	NULL	
5	A		2	*	cout
6	NULL		14	D	
5	A	cout	15	NULL	
7	NULL		14	D	cout
4	/	cout	16	NULL	
8	B		1	+	cout
9	NULL		17	E	
8	B	cout	18	NULL	
10	NULL		17	E	cout
3	*	cout	19	NULL	

Insert Node in Binary Search Tree



Insert 80 Insert 35

Deletion for A Binary Search Tree



Before deleting 60 After deleting 60

Inorder, Preorder and Postorder Program in C++

```
typedef struct BST {
  int data;
  struct BST *lchild, *rchild;
} node;
 void insert(node *, node *);
void inorder(node *);
void preorder(node *);
void postorder(node *);
node *search(node *, int, node **);
 void main() {
  int choice;
  char ans = 'N';
  int key;
  node *new_node, *root, *tmp, *parent;
  node *get_node();
  root = NULL;
  clrscr();   cout<<"\nProgram For Binary Search Tree ";
```

```
do {
  cout<<"\n1.Create";
  cout<<"\n2.Search";
  cout<<"\n3.Recursive Traversals";
  cout<<"\n4.Exit";
  cout<<"\nEnter your choice :;
  cin>>"%d", &choice;
   switch (choice) {
  case 1:        do {
      new_node = get_node();
      cout<<"\nEnter The Element ";
      cin>>"%d", &new_node->data;
       if (root == NULL) /* Tree is not Created */
         root = new_node;
      else
         insert(root, new_node);
       cout<<"\nWant To enter More Elements?(y/n)";
      ans = getch();
    } while (ans == 'y');        break;
```



```
case 2:
       cout<<"\nEnter Element to be searched :";
       cin>>"%d", &key;
        tmp = search(root, key, &parent);
       cout<<"\nParent of node %d is %d", tmp->data, parent->data;
       break;
     case 3:
       if (root == NULL)
          cout<<"Tree Is Not Created";
       else {
          cout<<"\nThe Inorder display : ";
          inorder(root);
          cout<<"\nThe Preorder display : ";
          preorder(root);
          cout<<"\nThe Postorder display : ";
          postorder(root);
       }
       break;      }
  } while (choice != 4); }
```

```
/* Get new Node*/
node *get_node() {
  node *temp;
  temp = (node *) malloc(sizeof(node));
  temp->lchild = NULL;
  temp->rchild = NULL;
  return temp;
}
```



```
/* This function is for creating a binary search tree */
void insert(node *root, node *new_node) {
  if (new_node->data < root->data) {
    if (root->lchild == NULL)
      root->lchild = new_node;
    else
      insert(root->lchild, new_node);
  }
   if (new_node->data > root->data) {
    if (root->rchild == NULL)
      root->rchild = new_node;
    else
      insert(root->rchild, new_node);
  } }
```

```
/* This function is for searching the node from binary Search Tree */
node *search(node *root, int key, node **parent) {
  node *temp;
  temp = root;
  while (temp != NULL) {
    if (temp->data == key) {
      cout<<"\nThe %d Element is Present", temp->data;
      return temp;      }
    *parent = temp;
     if (temp->data > key)
      temp = temp->lchild;
    else
      temp = temp->rchild;    }
  return NULL;
}
```



```
/* This function displays the tree in inorder
   fashion */
void inorder(node *temp) {
  if (temp != NULL) {
    inorder(temp->lchild);
    cout<<"%d", temp->data;
    inorder(temp->rchild);
  } }
```

```
/*This function displays the tree in
  preorder fashion */

void preorder(node *temp) {
  if (temp != NULL) {
    cout<<temp->data;
    preorder(temp->lchild);
    preorder(temp->rchild);
  } }
```



```
/* This function displays the tree in
  postorder fashion */

void postorder(node *temp) {
  if (temp != NULL) {
    postorder(temp->lchild);
    postorder(temp->rchild);
    cout<<temp->data;
  } }
```