CSC 427: Data Structures and Algorithm Analysis

Fall 2004

- C++ review (or What I Expect You to Know from 221/222)
  - program structure, input, output
  - variables, expressions, functions, parameters
  - control: if, if-else, while, for, recursion
  - predefined classes: string, vector, stack, queue
  - searching and sorting, algorithm efficiency (?)
  - user-defined classes, encapsulation, data hiding
  - pointers, dynamic memory, linked lists

Program structure and I/O

```
// ftoc.cpp
/////////////////////////////////////////////////////

#include <iostream>
using namespace std;

int main()
{
    double tempInFahr;
    cout << "Enter the temperature (in Fahrenheit): ";
    cin >> tempInFahr;

    cout << "You entered " << tempInFahr
         << " degrees Fahrenheit." << endl
         << "That's equivalent to "
         << (5.0/9.0 * (tempInFahr - 32))
         << " degrees Celsius" << endl;

   return 0;
}
```

- `#include` to load libraries
- every program must have `main` function (return type `int`)
- C++ comments use `//` or `/* … */`
- can output text using `cout` and `<<`

- declare a variable by specifying type, followed by variable name
  — types include `int`, `double`, `char`, `bool`, `string`*
  — can assign a value to a variable using =
  — can read a value into a variable using standard input stream `cin` and `>>` operator

Strings

```
// greet.cpp
//////////////////////////////////////////////////////

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

int main()
{
    string firstName, lastName;
    cout << "Enter your name (first then last): ";
    cin >> firstName >> lastName;

    cout << "Nice to meet you, " << firstName << " "<< lastName
         << ".  May I just call you " << firstName
         << "?" << endl;

   return 0;
}
```

- string type is defined in the library file `<string>`
  - note: if you used char* in 221/222, this is MUCH better!!!!
  - can read and write strings just like any other type
  - when reading a string value, delimited by whitespace

Expressions

- C++ provides various operators for constructing expressions

  `+ - * / %`

  (`+` can be applied to strings to concatenate)

- `cmath` library contains many useful routines

  `pow fabs sqrt`
  `floor ceil`

```
// change.cpp
//////////////////////////////////////////////////////

#include <iostream>
using namespace std;

int main()
{
    int amount;
    cout << "Enter an amount (in cents) to make change: ";
    cin >> amount;

    int quarters = amount/25;
    amount = amount%25;

    int dimes = amount/10;
    amount = amount%10;

    int nickels = amount/5;
    amount = amount%5;

    int pennies = amount;

    cout << "Optimal change:" << endl;
    cout << "    # of quarters =\t" << quarters  << endl;
    cout << "    # of dimes =\t" << dimes << endl;
    cout << "    # of nickels =\t" << nickels << endl;
    cout << "    # of pennies =\t" << pennies << endl;

    return 0;
}
```

Constants & formatted I/O

- constants define values that will not change
  - safer (compiler enforces)
  - easier to manage (global OK)
- `iomanip` library contains routines for formatting output
  - `setiosflags`
  - `setprecision`
  - `setw`

```
// pizza.cpp          Dave Reed          9/6/01
//
// This program determines the cost per sq. inch of a pizza.
////////////////////////////////////////////////////////////

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

const double PI = 3.14159;

int main()
{
    double pizzaDiameter, pizzaCost;
    cout << "Enter the diameter of the pizza (in inches): ";
    cin >> pizzaDiameter;
    cout << "Enter the price of the pizza (in dollars): ";
    cin >> pizzaCost;

    double pizzaArea = PI*(pizzaDiameter*pizzaDiameter)/4.0;
    double costPerSqInch = pizzaCost/pizzaArea;

    cout << setiosflags(ios::fixed);

    cout << "Total area of the pizza: "
         << setprecision(4) << pizzaArea << " square inches."
         << endl;
    cout << "    price per square inch = $"
         << setprecision(2) << costPerSqInch << "." << endl;

    return 0;
}
```


Functions

functions encapsulate computations

- define once, call many times
- abstract away details in main
- parameters in function store values passed in as arguments
- params & args match by position
- should always document any assumptions, return value

```
// lowhigh.cpp
////////////////////////////////////////////////

#include <iostream>
using namespace std;

double FahrToCelsius(double tempInFahr)
// Assumes: tempInFahr is a temperature in Fahrenheit
// Returns: equivalent temperature in Celsius
{
    return (5.0/9.0 * (tempInFahr - 32));
}

int main()
{
    double lowTemp, highTemp;
    cout << "Enter the forecasted low (in degrees Fahrenheit): ";
    cin >> lowTemp;
    cout << "Enter the forecasted high (in degrees Fahrenheit): ";
    cin >> highTemp;

    cout << endl
         <<"The forecasted low (in Celsius): " << FahrToCelsius(lowTemp) << endl
         <<"The forecasted high (in Celsius): " << FahrToCelsius(highTemp) << endl;

    return 0;
}
```

Functions calling functions

can place function prototypes above `main`, then function definitions in any order after `main`

```
// convert.cpp
///////////////////////////////

#include <iostream>
using namespace std;

double centimetersToInches(double cm);
double metersToFeet(double m);
double kilometersToMiles(double km);

int main()
{
  double distanceInKM;
  cout << "Enter a distance in kilometers: ";
  cin >> distanceInKM;

  cout << "That's equivalent to "
       << kilometersToMiles(distanceInKM)
       << " miles." << endl;

  return 0;
}

////////////////////////////////////////////
```

```
double centimetersToInches(double cm)
// Assumes: cm is a length in centimeters
// Returns: equivalent length in inches
{
    return cm/2.54;
}

double metersToFeet(double m)
// Assumes: m is a length in meters
// Returns: equivalent length in feet
{
    double cm = 100*m;
    double in = centimetersToInches(cm);
    return in/12;
}

double kilometersToMiles(double km)
// Assumes: km is a length in meters
// Returns: equivalent length in miles
{
    double m = 1000*km;
    double ft = metersToFeet(m);
    return ft/5280;
}
```


value vs. reference parameters

by default, parameters are passed *by-value*

- a copy of the input is stored in the parameter (a local variable)
- result: value passed in, no changes are passed out

& implies *by-reference*

- the parameter does not refer to a new piece of memory – it is an alias for the argument
- result: changes to the parameter simultaneously change the input

```
void foo(int x)
{
    x = 5;
    cout << x << endl;
}

int a = 3;
foo(a);

cout << a << endl;

foo(3);
```

note: input can be any value

```
void foo(int & x)
{
    x = 5;
    cout << x << endl;
}

int a = 3;
foo(a);

cout << a << endl;
```

note: input must be a variable

Advantages of functions

computational abstraction
- define & reuse
- ignore details

simplify repeated tasks
- avoids repeated code
- can generalize using params

can place useful functions in library
- reuse using `#include`

```
// oldmac.cpp
///////////////////////////////////

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

void Verse(string animal, string noise);

int main()
{
  Verse("cow", "moo");
  Verse("horse", "neigh");
  Verse("duck", "quack");

  return 0;
}

////////////////////////////////////////////////////////////////

void Verse(string animal, string noise)
// Assumes: animal is an animal name, and noise is the noise it makes
// Results: displays the corresponding OldMacDonald verse
{
  cout << "Old MacDonald had a farm, E-I-E-I-O." << endl;
  cout << "And on that farm he had a " << animal << ", E-I-E-I-O." << endl;
  cout << "With a " << noise << "-" << noise << " here, and a "
       << noise << "-" << noise << " there," << endl;
  cout << "  here a " << noise << ", there a " << noise << ", everywhere a "
       << noise << "-" << noise << "." << endl;
  cout << "Old MacDonald had a farm, E-I-E-I-O." << endl << endl;
}
```


If statements

- if statements provide for conditional execution
- simple if specifies code to be executed or not (depending on Boolean condition)

comparison operators

`==  !=  >  >=  <  <=`

logical connectives

`&&  ||  !`

```
// change.cpp        Dave Reed        9/20/01
/////////////////////////////////////////////////////////

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

void DisplayCoins(int numCoins, string coinType);

int main()
{
    // PROMPT USER & ASSIGN VARIABLES AS BEFORE

    cout << "Optimal change:" << endl;
    DisplayCoins(quarters, "quarters");
    DisplayCoins(dimes, "dimes");
    DisplayCoins(nickels, "nickels");
    DisplayCoins(pennies, "pennies");

    return 0;
}

///////////////////////////////////////////////////////////

void DisplayCoins(int numCoins, string coinType)
// Assumes: numCoins >= 0, coinType is a type of coin (e.g., nickel)
// Results: displays a message if numCoins > 0
{
    if (numCoins > 0) {
          cout << setw(4) << numCoins << " " << coinType << endl;
    }
}
```

2-way conditional

- if-else allows choice between two options
- can cascade if-else's to produce choose from more than two options

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

void DisplayCoins(int numCoins, string singleCoin, string pluralCoin);

int main()
{
    // PROMPT USER & ASSIGN VARIABLES AS BEFORE

    cout << "Optimal change:" << endl;
    DisplayCoins(quarters, "quarter", "quarters");
    DisplayCoins(dimes, "dime", "dimes");
    DisplayCoins(nickels, "nickel", "nickels");
    DisplayCoins(pennies, "penny", "pennies");

    return 0;
}

///////////////////////////////////////////////////////////

void DisplayCoins(int numCoins, string singleCoin, string pluralCoin)
// Assumes: numCoins >= 0, singleCoin is the name of a single coin
//          (e.g., penny), and pluralCoin is the plural (e.g., pennies)
// Results: displays a message if numCoins > 0
{
    if (numCoins == 1) {
        cout << setw(4) << numCoins << " " << singleCoin << endl;
    }
    else if (numCoins > 1) {
        cout << setw(4) << numCoins << " " << pluralCoin << endl;
    }
}
```


Local vs. global scope

scope = portion of program where variable is accessible

if declared in function or block { … }, then scope is limited to that function/block (i.e., *local*)

variables declared outside of functions are accessible to all (i.e., *global*)

✓variables should be as local as possible

✓global constants are OK

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

double WindChill(double temp, double wind);

int main()
{
    double temperature;
    cout << "Enter the temperature (in degrees Fahrenheit): ";
    cin >> temperature;

    if (temperature < -35 || temperature > 45) {
        cout << "That temperature is too extreme.  "
             << "Wind-chill is not defined." << endl;
    }
    else {
        int windSpeed;
        cout << "Enter the wind speed (in miles per hour): ";
        cin >> windSpeed;

        cout << endl << "The wind-chill factor is "
             << WindChill(temperature, windSpeed) << endl;
    }

    return 0;
}

double WindChill(double temp, double wind)
// Assumes: -35 <= temp <= 45, wind >= 0
// Returns: wind-chill factor given temperature and wind speed
{
    if (wind < 4) {
      return temp;
    }
    else {
      return 35.74 + 0.6215*temp +
             (0.4274*temp - 35.75)*pow(wind, 0.16);
    }
}
```

C++ libraries

C++ provides numerous libraries of useful code

- cmath functions for manipulating numbers, including:

```
double fabs(double x);
double sqrt(double x);
double ceil(double x);
double floor(double x);
double pow(double x, double y);
```

```
int numBits;
cin >> numBits;

cout << "With " << numBits << "bits, "
     << "you can represent "
     << pow(2,numBits) << "patterns.";
```

- cctype functions for testing and manipulating characters, including:

```
bool isalpha(char ch);
bool islower(char ch);
bool isupper(char ch);
bool isdigit(char ch);
bool isspace(char ch);


char tolower(char ch);
char toupper(char ch);
```

```
char response;
cout << "Do you want to play again? (y/n) ";
cin >> response;

if (tolower(response) == 'y') {
    PlayGame();
}
else {
    cout << "Thanks for playing." << endl;
}
```


Abstract data types

an *abstract data type (ADT)* is a collection of data and the associated operations that can be applied to that data

- e.g., a string is a sequence of characters enclosed in quotes with operations: concatenation, determine it length, access a character, access a substring, …

EXAMPLE: the C++ string class in `<string>`

DATA: a sequence of characters (enclosed in quotes)
MEMBER FUNCTIONS (METHODS):

```
+  >>  <<                         // operators for concatenation, input, and output
int length();                     // returns number of chars in string
char at(int index);               // returns character at index (first index is 0)
string substr(int pos, int len);  // returns substring starting at pos of length len
int find(string substr);          // returns position of first occurrence of substr,
                                  //      returns constant string::npos if not found
int find(char ch);                // similarly finds index of character ch
. . .
```

call a member function/method using '.', e.g., `str.length()`

Pig Latin (v. 1)

suppose we want to convert a word into Pig Latin

- simplest version
  nix → ixnay
  pig → igpay
  latin → atinlay
  banana → ananabay

- here, use
  `length`
  `at`
  `substr`
  `+`

```
// piglatin.cpp        Dave Reed        9/30/01
//
// First version of Pig Latin translator.
//////////////////////////////////////////////////////////

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

string PigLatin(string word);

int main()
{
    string word;
    cout << "Enter a word: ";
    cin >> word;

    cout << "That translates to: " << PigLatin(word) << endl;

    return 0;
}

/////////////////////////////////////////////////////////////

string PigLatin(string word)
// Assumes: word is a single word (no spaces)
// Returns: the Pig Latin translation of the word
{
    return word.substr(1, word.length()-1) + word.at(0) + "ay";
}
```


Pig Latin (v. 2)

need to recognize when starts with vowel
  apple → appleway
  isthmus → isthmusway

ugly way uses || to look for vowels

```
// piglatin.cpp      Dave Reed      9/30/01
//
// Second version of Pig Latin translator.
//////////////////////////////////////////

#include <iostream>
#include <string>
#include <cctype>
using namespace std;

string PigLatin(string word);
bool IsVowel(char ch);

int main()
{
    string word;
    cout << "Enter a word: ";
    cin >> word;

    cout << "That translates to: "
         << PigLatin(word) << endl;

    return 0;
}

//////////////////////////////////////////
```

```
string PigLatin(string word)
// Assumes: word is a single word (no spaces)
// Returns: the Pig Latin translation of the word
{
    if (IsVowel(word.at(0))) {
        return word + "way";
    }
    else {
        return word.substr(1, word.length()-1) +
                 word.at(0) + "ay";
    }
}

bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel ("aeiouAEIOU")
{
    ch = tolower(ch);

    return (ch == 'a' || ch == 'e' ||
            ch == 'i' || ch == 'o' || ch =='u');
}
```

Pig Latin (v. 3)

```
// piglatin.cpp      Dave Reed      9/30/01
//
// Third version of Pig Latin translator.
//////////////////////////////////////////

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

string PigLatin(string word);
bool IsVowel(char ch);

int main()
{
    string word;
    cout << "Enter a word: ";
    cin >> word;

    cout << "That translates to: "
         << PigLatin(word) << endl;

    return 0;
}

//////////////////////////////////////////
```

better way uses string search

- search for ch in a string of vowels

```
string PigLatin(string word)
// Assumes: word is a single word (no spaces)
// Returns: the Pig Latin translation of the word
{
    if (IsVowel(word.at(0))) {
        return word + "way";
    }
    else {
        return word.substr(1, word.length()-1) +
               word.at(0) + "ay";
    }
}

bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel ("aeiouAEIOU")
{
    const string VOWELS = "aeiouAEIOU";

    return (VOWELS.find(ch) != string::npos);
}
```


User-defined classes

similar to string, you can define your own classes (types) in C++

```
// Die.h

#ifndef _DIE_H
#define _DIE_H

#include <ctime>
using namespace std;

class Die
{
  public:
    Die(int sides = 6);
    int Roll();
    int NumSides() const;
    int NumRolls() const;
  private:
    int myRollCount;
    int mySides;

    static bool ourInitialized;
};

#endif
```

```
// Die.cpp

bool Die::ourInitialized = false;

Die::Die(int sides)
{
  if (!ourInitialized) {
    ourInitialized = true;     // call srand once
    srand(unsigned(time(0))); // randomize
  }
  myRollCount = 0;
  mySides = sides;
}

int Die::Roll()
{
  int dieRoll = int(rand()) % mySides + 1;
  myRollCount++;
  return dieRoll;
}

int Die::NumSides() const
{
  return mySides;
}

int Die::NumRolls() const
{
  return myRollCount;
}
```

Using user-defined classes

must #include .h file

must add .cpp to project

note:

- class constructor has default parameter
- Roll method is public
- # of sides, # of rolls private, but accessor methods are provided

```
// rollem.cpp
//
// Simulates rolling two dice
/////////////////////////////////////////

#include <iostream>
#include "Die.h"
using namespace std;

int main()
{
    Die sixSided, eightSided(8);

    int sum1 = sixSided.Roll() + sixSided.Roll();
    cout << "Two 6-sided dice: " << sum1 << endl;

    int sum2 = sixSided.Roll() + eightSided.Roll();
    cout << "6- and 8-sided dice:" << sum2 << endl;

    cout << "The " << sixSided.NumSides()
         << "-sided die has been rolled "
         << sixSided.NumRolls() << " times." << endl;

    return 0;
}
```


While loops

provide for conditional repetition

pseudo-code:

```
ROLL DICE;
DISPLAY RESULTS;

while (DICE ARE DIFFERENT){
  ROLL DICE AGAIN;
  DISPLAY RESULTS AGAIN;
}

DISPLAY NUMBER OF ROLLS;
```

again, Die member functions are useful

```
// doubles.cpp
//
// Simulates rolling two dice until doubles.
/////////////////////////////////////////////

#include <iostream>
#include "Die.h"
using namespace std;

int main()
{
    Die d1, d2;

    int roll1 = d1.Roll();
    int roll2 = d2.Roll();
    cout << "You rolled " << roll1
         << " and " << roll2 << endl;

    while (roll1 != roll2) {
        roll1 = d1.Roll();
        roll2 = d2.Roll();
        cout << "You rolled " << roll1
             << " and " << roll2 << endl;
    }

    cout << "It took " << d1.NumRolls() << " rolls."
         << endl;

    return 0;
}
```

Priming a loop

can avoid redundancy with a KLUDGE (a quick-and-dirty trick for making code work)

- only roll the dice inside the loop
- initialize the roll variables so that the loop test succeeds the first time
- after the first kludgy time, the loop behaves as before

```
// doubles.cpp
//
// Simulates rolling two dice until doubles.
/////////////////////////////////////////////

#include <iostream>
#include "Die.h"
using namespace std;

int main()
{
    Die d1, d2;

    int roll1 = -1;   // KLUDGE: initializes rolls
    int roll2 = -2;   // so that loop is entered

    while (roll1 != roll2) {
        roll1 = d1.Roll();
        roll2 = d2.Roll();
        cout << "You rolled " << roll1
             << " and " << roll2 << endl;
    }

    cout << "It took " << d1.NumRolls() << " rolls."
         << endl;

    return 0;
}
```


Counters

counters keep track of number of occurrences of some event

- must initialize to zero
- increment when event occurs

```
INITIALIZE 7-COUNT TO 0;

while (HAVEN'T FINISHED ROLLING) {
  ROLL DICE;
  IF ROLLED 7, UPDATE 7-COUNT;
}

DISPLAY 7-COUNT;
```

arithmetic assignments are handy

++   --   +=   -=   *=

```
// stats.cpp
//
// Simulates rolling two dice, counts 7's.
//////////////////////////////////////////

#include <iostream>
#include "Die.h"
using namespace std;

const int NUM_ROLLS = 1000;

int main()
{
    Die d1, d2;

    int sevenCount = 0;

    while (d1.NumRolls() < NUM_ROLLS) {
        int roll1 = d1.Roll();
        int roll2 = d2.Roll();

        if (roll1 + roll2 == 7) {
            sevenCount = sevenCount + 1;
        }
    }

    cout << "Out of " << NUM_ROLLS << " rolls, "
         << sevenCount << " were sevens." << endl;

    return 0;
}
```

While loops vs. for loops

use for loop when you know the number of repetitions ahead of time
use while loop when the number of repetitions is unpredictable

```
int i = 0;
while (i < MAX) {
    DO SOMETHING;
    i++;
}
```

```
for (int i = 0; i < MAX; i++) {
    DO SOMETHING;
}
```

while loop version:

```
int numTimes = 0;
while (numTimes < 10) {
  cout << "Howdy" << endl;
  numTimes++;
}
```

```
int i = 1, sum = 0;
while (i <= 100) {
  sum += i;
  i++;
}
```

for loop version:

```
for (int numTimes = 0; numTimes < 10; numTimes++) {
  cout << "Howdy" << endl;
}
```

```
int sum = 0;
for (i = 1; i <= 100; i++) {
  sum += i;
}
```


Pig Latin (final version)

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

string PigLatin(string word);
bool IsVowel(char ch);
int FindVowel(string str);

int main()
{
  string word;
  cout << "Enter a word: ";
  cin >> word;

  cout << "That translates to: "
       << PigLatin(word) << endl;

  return 0;
}

////////////////////////////////

bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel
    ("aeiouAEIOU")
{
  const string VOWELS = "aeiouAEIOU";
  return VOWELS.find(ch) != string::npos;
}
```

```
string PigLatin(string word)
// Assumes: word is a single word (no spaces)
// Returns: the Pig Latin translation of the word
{
  int vowelIndex = FindVowel(word);

  if (vowelIndex == 0 || vowelIndex == string::npos) {
    return word + "way";
  }
  else {
    return word.substr(vowelIndex,
                       word.length()-vowelIndex) +
           word.substr(0, vowelIndex) + "ay";
  }
}

int FindVowel(string str)
// Assumes: str is a single word (no spaces)
// Returns: the index of the first vowel in str, or
//          string::npos if no vowel is found
{
  for (int index = 0; index < str.length(); index++) {
    if (IsVowel(str.at(index))) {
      return index;
    }
  }

  return string::npos;
}
```

Vectors (flexible arrays)

the vector class is defined in the `<vector>` library

- can declare size at run-time, can even resize on fly
- performs bounds-checking on indexing
- parameter passing is normal

```
#include <vector>                  // loads definition of the vector class
using namespace std;


vector<int> nums(100);             // declares a vector of 100 ints
                                   // equiv. to  int nums[100];

for (int i = 0; i < 100; i++) {   // vector elements are accessible via
    nums[i] = 0;                   // an index (same as with arrays)
}


vector<string> words(10);          // declares a vector of 10 strings
                                   // equiv. to string words[10];


int numGrades;
cout << "How many grades are there? ";
cin >> numGrades;

vector<double> grades(numGrades); // declares a vector of numGrades
                                   // doubles (can't be done with arrays)
```


Reversing an array vs. Reversing a vector

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


const int MAX_SIZE = 100;

int main()
{
  string words[MAX_SIZE];
  int numWords = 0;

  string input;
  while (numWords < MAX_SIZE && cin >> input) {
    words[numWords] = input;
    numWords++;
  }




  for (int i = numWords-1; i >= 0; i--) {
    cout << words[i] << endl;
  }

  return 0;
}
```

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

const int INITIAL_SIZE = 10;

int main()
{
  vector<string> words(INITIAL_SIZE);
  int numWords = 0;

  string input;
  while (cin >> input) {
    if (numWords == words.size()) {
      words.resize(2*words.size());
    }
    words[numWords] = input;
    numWords++;
  }

  for (int i = numWords-1; i >= 0; i--) {
    cout << words[i] << endl;
  }

  return 0;
}
```

stops processing when array is full

simply resizes vector when full, goes on

Vectors as parameters

unlike arrays, vector parameters behave as any other type

```
void foo(vector<int> nums)
{
   nums[0] = 999;
}

vector<int> numbers(10);

foo(numbers);
```

RESULT: nums is a copy of the vector, no change to numbers[0]

```
void foo(vector<int> & nums)
{
  nums[0] = 999;
}

vector<int> numbers(10);

foo(numbers);
```

RESULT: nums is an alias for the numbers vector simultaneously changes numbers[0]

when passing large objects, alternative to by-value exists

```
void Display(const vector<int> & nums)       // & implies reference to
{                                            //   original vector is passed
   for (int i = 0; i < nums.size(); i++) {   // const ensures no changes made
       cout << nums[i] << endl;
   }
}
```


Grade cutoff example

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

const int INITIAL_SIZE = 20;

void ReadGrades(vector<int> & grades,
                int & numGrades);
int CountAbove(vector<int> grades,
               int numGrades, int cutoff);

int main()
{
  vector<int> grades(INITIAL_SIZE);
  int numGrades;
  ReadGrades(grades, numGrades);

  int cutoff;
  cout << "Enter the desired grade cutoff: ";
  cin >> cutoff;

  cout << "There are "
       << CountAbove(grades, numGrades, cutoff)
       << " grades above " << cutoff << endl;

  return 0;
}

/////////////////////////////////////////////
```

```
void ReadGrades(vector<int> & grades,
                int & numGrades)
// Results: reads grades and stores in vector
//          numGrades is set to the # of grades
{
    string filename;
    cout << "Enter the grades file name: ";
    cin >> filename;

    ifstream myin;
    myin.open( filename.c_str() );

    while (!myin) {
      cout << "File not found.  Try again: ";
      cin >> filename;
      myopen.clear();
      myin.open( filename.c_str() );
    }

    numGrades = 0;

    int grade;
    while (myin >> grade) {
      if (numGrades == grades.size()) {
        grades.resize(2*grades.size());
      }
      grades[numGrades] = grade;
      numGrades++;
    }
    myin.close();
}
```

Grade cutoff (using push_back)

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

void ReadGrades(vector<int> & grades);
int CountAbove(vector<int> grades, int cutoff);

int main()
{
  vector<int> grades;
  ReadGrades(grades);

  int cutoff;
  cout << "Enter the desired grade cutoff: ";
  cin >> cutoff;

  cout << "There are "
       << CountAbove(grades, cutoff)
       << " grades above " << cutoff << endl;

  return 0;
}

////////////////////////////////////////////////
```

```
void ReadGrades(vector<int> & grades)
{
    string filename;
    cout << "Enter the grades file name: ";
    cin >> filename;

    ifstream myin;
    myin.open( filename.c_str() );

    while (!myin) {
      cout << "File not found.  Try again: ";
      cin >> filename;
      myin.clear();
      myin.open( filename.c_str() );
    }

    int grade;
    while (myin >> grade) {
      grades.push_back(grade);
    }
    myin.close();
}

int CountAbove(const vector<int> & grades,
               int cutoff)
{
    int numAbove = 0;
    for(int i = 0; i < grades.size(); i++) {
        if (grades[i] >= cutoff) {
            numAbove++;
        }
    }

    return numAbove;
}
```


Class use

```
class Card {
  public:
    Card();
    char GetSuit();
    char GetRank();
    int RankValue();
    void Assign(char r, char s);
  private:
    char rank;
    char suit;
};


class DeckOfCards {
  public:
    DeckOfCards();
    void Shuffle();
    Card DrawFromTop();
    void PlaceOnBottom(Card c);
    bool IsEmpty();
  private:
    vector<Card> cards;
    int numCards;
};
```

given the interface for a class, you should be able to use that class (without caring about implementation details)

```
DeckOfCards deck;

deck.Shuffle();

for (int i = 0; i < 5; i++) {
  Card c = DrawFromTop();

  cout << c.GetRank() << " "
       << c.GetSuit() << endl;

  deck.PlaceOnBottom(c);
}
```

Stacks and queues

a stack is a list where all additions, deletions, accesses occur at one end

```
#include <stack>                // C++ provides class in library

void push(const TYPE & item);  // adds item to top of stack
void pop();                    // removes item at top of stack
TYPE & top() const;            // returns item at top of stack
bool empty() const;            // returns true if stack is empty
int size() const;              // returns size of stack
```

a queue is a list where additions occur at one end, deletions and accesses occur at the other end

```
#include <queue>                // C++ provides class in library

void push(const TYPE & item);  // adds item to back of queue
void pop();                    // removes item from front of queue
TYPE & front() const;          // returns item at front of queue
bool empty() const;            // returns true if queue is empty
int size() const;              // returns size of queue
```


Pointers

a pointer is nothing more than an address (i.e., an integer)

- can declare a pointer to a particular type of value using *

```
int * p;            p [•]--> [???]

string * q;         q [•]--> [  ???  ]
```

operations on pointers:

- *dereference operator:* given a pointer to some memory location, can access the value stored there using *

```
p [•]--> [4]          cout << *p << endl;
```

- *address-of operator:* given a memory cell, can get its address (i.e., a pointer to it) using &

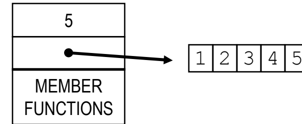
```
x [7]                 p = &x;
```

Vectors and dynamic arrays

the underlying data structure of a vector is a dynamically-allocated array

```
template <class Type>
class vector
{
  public:
    vector(int size = 0) {
      vecLength = size;
      vecList = new Type[size];
    }

    int size() {
      return vecLength;
    }

    Type & operator[](int index) {
      return vecList[index];
    }

    // OTHER MEMBER FUNCTIONS

  private:
    Type * vecList;
    int vecLength;
};
```

vector object



classes with dynamic data fields should have:

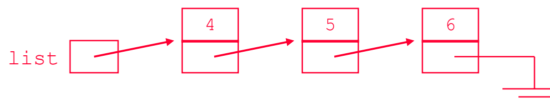
- copy constructor: specifies how to make a deep copy of an object (instead of just copying pointers)
- destructor: automatically called when the object's lifetime ends, to reclaim dynamic memory

Node class

```
template <class Item> class Node
{
  public:
    Node<Item>(Item item, Node<Item> * ptr = NULL)
    {
        value = item;
        next = ptr;
    }

    void setValue(Item item)
    {
        value = item;
    }

    void setNext(Node<Item> * ptr)
    {
        next = ptr;
    }

    Item getValue()
    {
        return value;
    }

    Node<Item> * getNext()
    {
        return next;
    }
  private:
    Item value;
    Node<Item> * next;
};
```

can define a generic Node class – useful for constructing a linked list

ADVANTAGES?




```
Node<int> * list = new Node<int>(5, NULL);

Node<int> * temp = new Node<int>(4, list);
list = temp;

Node<int> * second = list->getNext();
second->setNext(new Node<int>(6, NULL));
```