

IT 211
DATA STRUCTURES
MIDTERM
REQUIREMENT



1. Write a program to enter the name and votes of 5 candidates. The program should display each candidate's name, votes and percentage of votes. The program should also declare the winner.

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

class ElectionResults
{
public:
    string name;
    int votes;
};

int main()
{
    ElectionResults res[5];
    int voteTotal = 0;
    int winner = 0;
    int byVotes = 0;
    float percent = 0.0;

    for(int i = 0; i < 5; i++)
    {
        cout << "Enter Last Name of Candidate "<<i+1<<"      : ";
        cin >> res[i].name;

        cout<<"Enter the Votes Received for Candidate "<<i+1<<" : ";
        cin >> res[i].votes;
        cout<<"-----"<<endl;

        voteTotal += res[i].votes;
        if( res[i].votes > byVotes)
            winner = i;
    }
}
```



IT 211 DATA STRUCTURES
MIDTERM REQUIREMENT

```
cout<<"Name of candidate\tVotes received\tPercentage"<<endl;

for(int i = 0; i < 5; i++)
{
    cout<<res[i].name<<"\t\t"<< res[i].votes
    <<"\t\t"<<(res[i].votes*100)/voteTotal << "%." <<endl;
}
cout<<"\nTotal Votes: "<<voteTotal<<endl;
cout << "The Winner is:" << res[winner].name<<endl;
cin >> voteTotal;
}
```



2. STRINGS

a. Write a program to accept and display the number of letters, words and sentences of a text input.

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

int main(){
    string userInput;
    char myChar;

    int words = 1;
    int sentences = 0;
    int paragraphs = 1;
    cout << "Enter some text: ";
    getline (cin, userInput);

    for (int i = 0; i <int(userInput.length()); i++)
    {
        myChar = userInput.at(i);

        /*if (userInput[i] == isascii(NUL))    {
            words --;
            paragraphs --;
        }*/

        if (userInput[i] == ' ')
            words++;

        if (userInput[i] == '.')
            sentences++;

        if (userInput[i] == '\n' && userInput[i] == '\t')
            paragraphs++;

    }

    cout << "words: " << words << endl;
    cout << "sentences: " << sentences << endl;
    cout << "paragraphs: " << paragraphs << endl;

    int p;
    cin >> p;
    return 0;
}
```



b. Write a program to determine if a word is palindrome

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

int main()
{
    char str[25],str1[25];
    cout<<"Enter a string: ";

    gets(str);

    strcpy(str1,str);
    strrev(str);
    if(strcmp(str,str1)!=0)
    {
        cout<<"string is not palindrome";
    }
    else
    {
        cout<<"string is a palindrome";
    }

    cout<<endl;
    cin.get();
    return 0;
}
```



3. RECURSION

Write a program to convert a binary number to its decimal equivalent using a recursive function.

```
//BINARY TO DECIMAL
#include <iostream>
#include <math.h>

using namespace std;

int sum = 0;
using namespace std;
int BinaryToDecimal (int BinaryNumber, int weight);

int main(){
    int bitWeight;
    int binaryNum;
    bitWeight = 0;
    cout<<"=====CONVERTING BINARY TO DECIMAL===== "<<endl;
    cout<<"Enter the Binary Number: ";
    cin>>binaryNum;

    cout<<"\n";

    int sum = BinaryToDecimal (binaryNum, bitWeight);
    cout<<"The Decimal Equivalent of inputed Binary is: "<<sum<<endl;

    system("PAUSE");
    return 0;
}

int BinaryToDecimal (int BinaryNumber, int weight)
{
    int bit;
    if (BinaryNumber>0){
        bit = BinaryNumber % 10;
        sum = sum + bit * pow (2, weight);
        BinaryNumber = BinaryNumber /10;
        weight++;
        BinaryToDecimal (BinaryNumber, weight);
    }

    return sum;
}
}
}
```



IT 211 DATA STRUCTURES MIDTERM REQUIREMENT

```
//DECIMAL TO BINARY
#include <iostream>

using namespace std;

void binary(int);

int main(void) {
    int number;

    cout << "Please enter a positive integer: ";
    cin >> number;
    if (number < 0)
        cout << "That is not a positive integer.\n";
    else {
        cout << number << " converted to binary is: ";
        binary(number);
        cout << endl;
    }
}

void binary(int number) {
    int remainder;

    if(number <= 1) {
        cout << number;
        return;
    }

    remainder = number%2;
    binary(number >> 1);
    cout << remainder;
    cin.get();
}
```



LECTURE REQUIREMENTS

Research on the definitions, C++ implementation, and program examples of the following:

- Stack
- Lists
- Queues
- Pointers

1. Stack

LIFO stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from the end of the container.

stacks are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed/popped* from the "back" of the specific container, which is known as the *top* of the stack.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The only requirement is that it supports the following operations:

- `back()`
- `push_back()`
- `pop_back()`

Therefore, the standard container class templates vector, deque and list can be used. By default, if no container class is specified for a particular `stack` class, the standard container class template deque is used.

In their implementation in the C++ Standard Template Library, stacks take two template parameters:

```
template < class T, class Container = deque<T> > class stack;
```




IT 211 DATA STRUCTURES MIDTERM REQUIREMENT

Where the template parameters have the following meanings:

- **T:** Type of the elements.
- **Container:** Type of the underlying container object used to store and access the elements.

```
1 // stack::push/pop
2 #include <iostream>
3 #include <stack>
4 using namespace std;
5
6 int main ()
7 {
8     stack<int> mystack;
9
10    for (int i=0; i<5; ++i) mystack.push(i);
11
12    cout << "Popping out elements...";
13    while (!mystack.empty())
14    {
15        cout << " " << mystack.top();
16        mystack.pop();
17    }
18    cout << endl;
19
20    return 0;
21 }
```



2. LISTS

Lists are a kind of sequence container. As such, their elements are ordered following a linear sequence.

List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept by the association to each element of a link to the element preceding it and a link to the element following it.

This provides the following advantages to `list` containers:

- Efficient insertion and removal of elements before any other specific element in the container (constant time).
- Efficient moving elements and block of elements within the container or even between different containers (constant time).
- Iterating over the elements in forward or reverse order (linear time).

Compared to other base standard sequence containers ([vectors](#) and [deque](#)s), lists perform generally better in inserting, extracting and moving elements in any position within the container for which we already have an iterator, and therefore also in algorithms that make intensive use of these, like sorting algorithms.

The main drawback of `lists` compared to these other sequence containers is that they lack direct access to the elements by their position; For example, to access the sixth element in a `list` one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

Storage is handled automatically by the `list` object, allowing it to be expanded and contracted automatically as needed.

In their implementation in the C++ Standard Template Library lists take two template parameters:

```
template < class T, class Allocator = allocator<T> > class list;
```



IT 211 DATA STRUCTURES MIDTERM REQUIREMENT

Where the template parameters have the following meanings:

- **T:** Type of the elements.
- **Allocator:** Type of the allocator object used to define the storage allocation model. By default, the `allocator` class template for type `T` is used, which defines the simplest memory allocation model and is value-independent.

In the reference for the `list` member functions, these same names are assumed for the template parameters.

```
1 // list::pop_back
2 #include <iostream>
3 #include <list>
4 using namespace std;
5
6 int main ()
7 {
8     list<int> mylist;
9     int sum (0);
10    mylist.push_back (100);
11    mylist.push_back (200);
12    mylist.push_back (300);
13
14    while (!mylist.empty())
15    {
16        sum+=mylist.back();
17        mylist.pop_back();
18    }
19
20    cout << "The elements of mylist summed " << sum << endl;
21
22    return 0;
23 }
```



3. Queues

FIFO queue

queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

queues are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed* into the "*back*" of the specific container and *popped* from its "*front*".

The underlying container may be one of the standard container class template or some other specifically designed container class. The only requirement is that it supports the following operations:

- `front()`
- `back()`
- `push_back()`
- `pop_front()`

Therefore, the standard container class templates [deque](#) and [list](#) can be used. By default, if no container class is specified for a particular `queue` class, the standard container class template [deque](#) is used.



IT 211 DATA STRUCTURES MIDTERM REQUIREMENT

In their implementation in the C++ Standard Template Library, queues take two template parameters:

```
template < class T, class Container = deque<T> > class queue;
```

Example

```
1 // constructing queues
2 #include <iostream>
3 #include <deque>
4 #include <list>
5 #include <queue>
6 using namespace std;
7
8 int main ()
9 {
10     deque<int> mydeck (3,100); // deque with 3 elements
11     list<int> mylist (2,200); // list with 2 elements
12
13     queue<int> first; // empty queue
14     queue<int> second (mydeck); // queue initialized to copy of deque
15
16     queue<int,list<int> > third; // empty queue with list as underlying container
17     queue<int,list<int> > fourth (mylist);
18
19     cout << "size of first: " << (int) first.size() << endl;
20     cout << "size of second: " << (int) second.size() << endl;
21     cout << "size of third: " << (int) third.size() << endl;
22     cout << "size of fourth: " << (int) fourth.size() << endl;
23
24     return 0;
25 }
```



4. Pointers

A pointer is a variable that is used to store a memory address. The address is the location of the variable in the memory. Pointers help in allocating memory dynamically. Pointers improve execution time and saves space. Pointer points to a particular data type. The general form of declaring pointer is:-

```
type *variable_name;
```

type is the base type of the pointer and variable_name is the name of the variable of the pointer. For example,

```
int *x;
```

x is the variable name and it is the pointer of type integer.

Pointer Operators

There are two important pointer operators such as '*' and '&'. The '&' is a [unary operator](#). The unary operator returns the address of the memory where a variable is located. For example,

```
int x*;
```

```
int c;
```

```
x=&c;
```



variable x is the pointer of the type integer and it points to location of the variable c. When the statement

```
x=&c;
```

is executed, '&' operator returns the memory address of the variable c and as a result x will point to the memory location of variable c.

The '*' operator is called the [indirection operator](#). It returns the contents of the memory location pointed to. The indirection operator is also called deference operator. For example,

```
int x*;  
int c=100;  
int p;  
x=&c;  
p=*x;
```

variable x is the pointer of integer type. It points to the address of the location of the variable c. The pointer x will contain the contents of the memory location of variable c. It will contain value 100. When statement

```
p=*x;
```

is executed, '*' operator returns the content of the pointer x and variable p will contain value 100 as the pointer x contain value 100 at its memory location. Here is a program which illustrates the working of pointers.



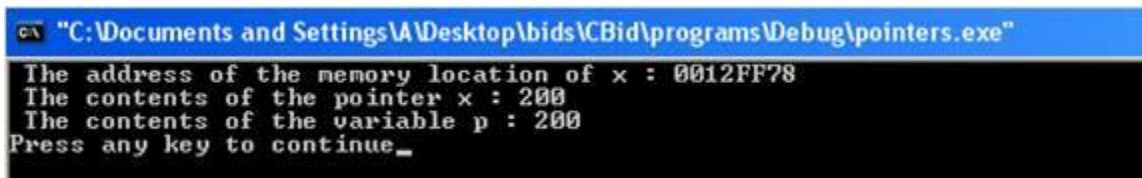
IT 211 DATA STRUCTURES MIDTERM REQUIREMENT

```
#include<iostream>

using namespace std;

int main ()
{
    int *x;
    int c=200;
    int p;
    x=&c;
    p=*x;
    cout << " The address of the memory location of x : " << x << endl;
    cout << " The contents of the pointer x : " << *x << endl;
    cout << " The contents of the variable p : " << p << endl;
    return(0);
}
```

The result of the program is:-



```
"C:\Documents and Settings\A\Desktop\ibids\CBid\programs\Debug\pointers.exe"
The address of the memory location of x : 0012FF78
The contents of the pointer x : 200
The contents of the variable p : 200
Press any key to continue_
```




IT 211 DATA STRUCTURES
MIDTERM REQUIREMENT