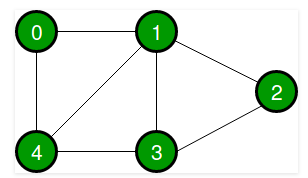
# CIS7 Unit 10 Rev1 Lab: Graphs in C++ Programming.

In this lab activity, we will use C++ programming to examine the adjacency list and matrices of 2-D graphs as explained in Chapter 16. See Chapter 16 notes for more information on graphs, vertices, and matrix.

In the below graph 1, 5 vertices are displayed: 0, 1, 2, 3, 4



Graph 1: Undirected simple graph

1. Refer to the above Graph 1 and complete the following tasks:
2. Identify the vertices and the edges.

Vertices: 0, 1, 2, 3, 4

Edges: {0, 1}, {0,4}, {1,2}, {1,3}, {1,4}, {3,4}, {2,3}

1. Create an adjacency list.

0 adjacent to 1, 4

1 adjacent to 0, 2, 3, 4

2 adjacent to 1, 3

3 adjacent to 1, 2, 4

4 adjacent to 0, 1, 3

1. Create an adjacency matrix

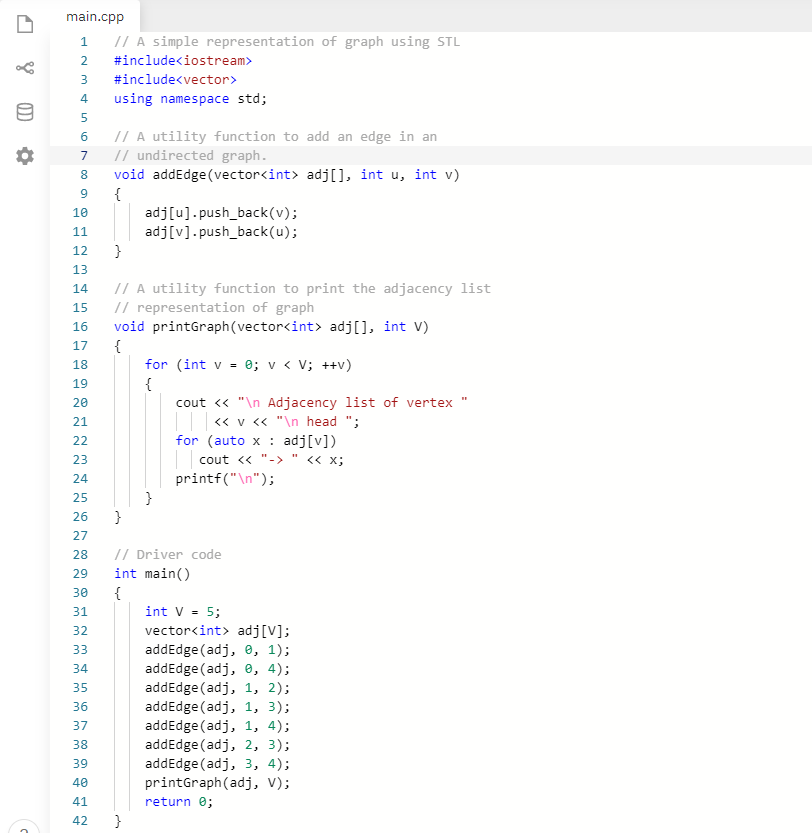
| Vertex | 0 | 1 | 2 | 3 | 4 |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

### std::vector::push\_back

***Adds a new element*** ***at the end of the vector***, ***after its current last element***. The value content is copied (or moved) to the new element.

This effectively ***increases the container size by one***, which causes an ***automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity***.

**Example 1:** C++ program to present simple graph using vector.



1. Write and run Example 1 in an IDE and examine the program output.

A screenshot of a computer

Description automatically generated

1. Verify the output of the Example 1 program. Does it match with Exercise 1? If not, explain your answer.

Yes they match.

1. Review the program code and identify the section of the code that produces the adjacency list for the graph and explain its functionality in the overall program.

The adjacency list is defined in main but filled using the addEdge function. Each time the function is called it takes in two indexes u and v. It adds the value v onto the list at index u and adds the value u onto the list at index v.

1. Given a graph, can a developer create a C++ program without evaluating the adjacency vertices?

It is possible in some instances but for anything requiring traversals it is very difficult to achieve this if not impossible as we must know the connections between vertices to be able to traverse trough the graph.

### Data Structure: struct

A **data structure** is ***a group of data elements grouped together under one name***. These data elements, known as **members**, can ***have different types and different lengths***. Data structures can be declared in C++ using the following syntax:

struct type\_name {

member\_type1 member\_name1;

member\_type2 member\_name2;

member\_type3 member\_name3;

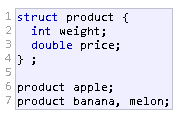
.

.

} object\_names;

Where **type\_name** is a ***name for the structure type***, **object\_name** can be ***a set of valid identifiers for objects that have the type of this structure***. Within braces {}, there is a list with the data members, each one is specified with a type and a valid identifier as its name.

Example:

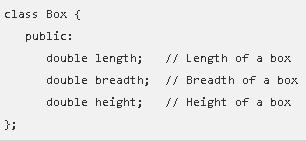


This declares a ***structure type***, called *product*, and defines it having ***two members***: ***weight and price***, each of a different fundamental type. This declaration creates a new type (product), which is then used to declare three objects (variables) of this type: apple, banana, and melon. Note how once product is declared, it is used just like any other type.

Right at the end of the **struct** definition, and before the ***ending semicolon (;)***, the optional field object\_names can be used to directly declare objects of the structure type. For example, the structure objects apple, banana, and melon can be declared at the moment the data structure type is defined.

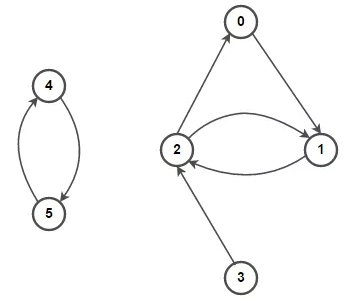
### Class in C++

A **class** is used to ***specify the form of an object and it combines data representation and methods for manipulating that data into one neat package***. The data and functions within a class are called **members of the class**. A **class** ***definition starts with the keyword class followed by the class name***; and the ***class body, enclosed by a pair of curly braces***. A class definition ***must be followed either by a semicolon or a list of declarations***. For example, we defined the Box data type using the keyword class as follows –



The keyword **public** ***determines the access attributes of the members of the class that follows it***. A ***public member can be accessed from outside the class*** ***anywhere within the scope*** ***of the class object***. You can also specify the members of a class as **private or protected**.

1. Refer to the below graph, Graph 2, and complete the following tasks:



**Graph 2: Directed simple graph**

1. Identify the vertices and edges of Graph 2.

Vertices: 0, 1, 2, 3, 4, 5

Edges: {0,1}, {1,2}, {2,0}, {2,1}, {3,2}, {4,5}, {5,4}

1. Determine the adjacency list and matrix for Graph 2.

0 -> 1

1 -> 2

2 -> 1 -> 0

3 -> 2

4 -> 5

5 -> 4

| Vertex | 0 | 1 | 2 | 3 | 4 | 5 |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 | 0 |

1. Input the example 2 C++ program in an IDE and run the program.

**Example 2**: C++ program representing adjacency list of a graph.

#include <iostream>

using namespace std;

// Data structure to store Adjacency list nodes

struct Node {

int val;

Node\* next;

};

// Data structure to store graph edges

struct Edge {

int src, dest;

};

class Graph

{

// Function to allocate new node of Adjacency List

Node\* getAdjListNode(int dest, Node\* head)

{

Node\* newNode = new Node;

newNode->val = dest;

// point new node to current head

newNode->next = head;

return newNode;

}

int N; // number of nodes in the graph

public:

// An array of pointers to Node to represent

// adjacency list

Node \*\*head;

// Constructor

Graph(Edge edges[], int n, int N)

{

// allocate memory

head = new Node\*[N]();

this->N = N;

// initialize head pointer for all vertices

for (int i = 0; i < N; i++)

head[i] = nullptr;

// add edges to the directed graph

for (unsigned i = 0; i < n; i++)

{

int src = edges[i].src;

int dest = edges[i].dest;

// insert in the beginning

Node\* newNode = getAdjListNode(dest, head[src]);

// point head pointer to new node

head[src] = newNode;

// Uncomment below lines for undirected graph

/\*

newNode = getAdjListNode(src, head[dest]);

// change head pointer to point to the new node

head[dest] = newNode;

\*/

}

}

// Destructor

~Graph() {

for (int i = 0; i < N; i++)

delete[] head[i];

delete[] head;

}

};

// print all neighboring vertices of given vertex

void printList(Node\* ptr)

{

while (ptr != nullptr)

{

cout << " -> " << ptr->val << " ";

ptr = ptr->next;

}

cout << endl;

}

// Graph Implementation in C++ without using STL

int main()

{

// array of graph edges as per above diagram.

Edge edges[] =

{

// pair (x, y) represents edge from x to y

{ 0, 1 }, { 1, 2 }, { 2, 0 }, { 2, 1 },

{ 3, 2 }, { 4, 5 }, { 5, 4 }

};

// Number of vertices in the graph

int N = 6;

// calculate number of edges

int n = sizeof(edges)/sizeof(edges[0]);

// construct graph

Graph graph(edges, n, N);

// print adjacency list representation of graph

for (int i = 0; i < N; i++)

{

// print given vertex

cout << i << " --";

// print all its neighboring vertices

printList(graph.head[i]);

}

return 0;

}

1. Review above program of Graph 2 and determine the areas of the program that produce the representation of the graph components: vertices, edges and adjacency.

Number of vertices represented by N value in Graph Class. Number of edges represented by n. Edges are represented using an array of Edge Structures. The adjacency list is stored in the Graph Class using a linked list of node pointers. When an object of the class is instantiated with the edges array, n, and N the constructor is called and builds the graph from this information.

1. Compare the program adjacency list to your result in Exercise 2 Question B.

A screenshot of a computer

Description automatically generated

They are matching.

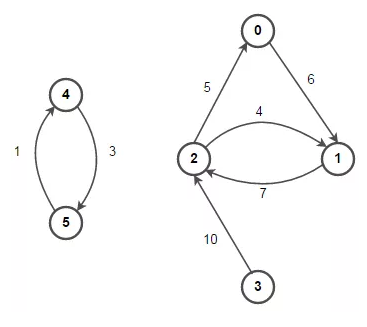
1. Can a developer create this program without pre-determining the adjacency list?

Yes the developer could dynamically build the graph using user input. However, like in the first example any kind of algorithm which requires knowledge of connections between vertices would be very difficult if not impossible.

1. Compare to a simple graph program, such as Exercise 2 program, what areas of the program produce edge directions? Explain the code functionality in the overall program.

The constructor uses information passed in the form of an edge array to add edges between source and destination vertices in the graph and determine their directions. If this were creating an undirected graph two edges would be added on from source to destination vertex and one from destination to source vertex. Then once the graph is built the program then calls the graph class’s print method which references a linked list built inside the constructor holding the adjacency information.

1. Refer to the below graph, Graph 3, and complete the following tasks:



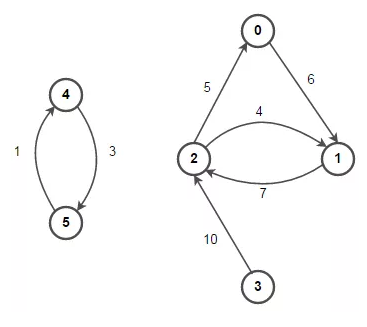
**Graph 3: Directed Weighted Graph**

1. Identify the vertices and edges in Graph 3.

Vertices: 0, 1, 2, 3, 4, 5

Edges: {0,1}, {1,2}, {2,0}, {2,1}, {3,2}, {4,5}, {5,4}

1. Derive an adjacency list and a matrix for Graph 3.



Adjacency:

0 -> 1

1 -> 2

2 -> 1 -> 0

3 -> 2

4 -> 5

5 -> 4

| Vertex | 0 | 1 | 2 | 3 | 4 | 5 |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 5 | 0 | 0 | 0 |
| 1 | 6 | 0 | 4 | 0 | 0 | 0 |
| 2 | 0 | 7 | 0 | 10 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 3 | 0 |

1. Input the following C++ program in IDE, run the program and answer the following questions.

#include <iostream>

using namespace std;

// Data structure to store Adjacency list nodes

struct Node {

int val, cost;

Node\* next;

};

// Data structure to store graph edges

struct Edge {

int src, dest, weight;

};

class Graph

{

// Function to allocate new node of Adjacency List

Node\* getAdjListNode(int value, int weight, Node\* head)

{

Node\* newNode = new Node;

newNode->val = value;

newNode->cost = weight;

// point new node to current head

newNode->next = head;

return newNode;

}

int N; // number of nodes in the graph

public:

// An array of pointers to Node to represent

// adjacency list

Node \*\*head;

// Constructor

Graph(Edge edges[], int n, int N)

{

// allocate memory

head = new Node\*[N]();

this->N = N;

// initialize head pointer for all vertices

for (int i = 0; i < N; ++i)

head[i] = nullptr;

// add edges to the directed graph

for (unsigned i = 0; i < n; i++)

{

int src = edges[i].src;

int dest = edges[i].dest;

int weight = edges[i].weight;

// insert in the beginning

Node\* newNode = getAdjListNode(dest, weight, head[src]);

// point head pointer to new node

head[src] = newNode;

// Uncomment below lines for undirected graph

/\*

newNode = getAdjListNode(src, weight, head[dest]);

// change head pointer to point to the new node

head[dest] = newNode;

\*/

}

}

// Destructor

~Graph() {

for (int i = 0; i < N; i++)

delete[] head[i];

delete[] head;

}

};

// print all neighboring vertices of given vertex

void printList(Node\* ptr, int i)

{

while (ptr != nullptr)

{

cout << "(" << i << ", " << ptr->val

<< ", " << ptr->cost << ") ";

ptr = ptr->next;

}

cout << endl;

}

// Graph Implementation in C++ without using STL

int main()

{

// array of graph edges as per above diagram.

Edge edges[] =

{

// (x, y, w) -> edge from x to y having weight w

{ 0, 1, 6 }, { 1, 2, 7 }, { 2, 0, 5 }, { 2, 1, 4 },

{ 3, 2, 10 }, { 4, 5, 1 }, { 5, 4, 3 }

};

// Number of vertices in the graph

int N = 6;

// calculate number of edges

int n = sizeof(edges)/sizeof(edges[0]);

// construct graph

Graph graph(edges, n, N);

// print adjacency list representation of graph

for (int i = 0; i < N; i++)

{

// print all neighboring vertices of vertex i

printList(graph.head[i], i);

}

return 0;

}

1. Compare the program adjacency list to your result in Exercise 3 Question B.

A screenshot of a computer

Description automatically generated

They are the same

1. Can a developer create this program without pre-determining the adjacency list or a matrix?

Yes a developer can write their program in a manner such that a graph can dynamically be built from a set of inputs without pre-determined adjacency information. However, the program would need some way to determine how the nodes and edges must be interconnected.

1. Compare to a unweighted directed graph program, such as Exercise 2 program, what areas of Exercise 3 program produce the weighted edges? Explain the code functionality in the overall program.

This program’s node structure contains an extra parameter called cost and the programs edge structure contains an extra parameter called weight. When the constructor is called with the Edge array each Node pointers cost value is set to the weight of that edge and added onto the linked list storing adjacency information.

Submit a document that contains answers and screen capture in Canvas (Unit 10 Lab).