Arrays

CSC200
Spring 2005
Marc Schraffenberger
What is an Array?

• Used for implementing a sequence of data items of the same type.

• Every element of the sequence can be accessed separately.

• The number of elements is defined at the declaration.

• Size of the array cannot change.
Array Declaration

\[\text{type\_name\ variable\_name[\text{size}]};\]
\[\text{type\_name\ variable\_name[\text{size}]} = \{\text{value1, value2, \ldots valueN} \};\]

**Examples:**

float density[50];

int position[3] = \{20, 40, -12\};

dBodyID car[5];
Array Subscripts

- Access elements of an array with the [] operator.
- Error occurs if trying to access an element outside the range.
- First element corresponds to the number zero.

Examples:

dBodyID bike[3];  // a bike with 3 bodies.
bike[0] = dBodyCreate(world);  // OK.
bike[1] = dBodyCreate(world);  // OK.
bike[2] = dBodyCreate(world);  // OK.
bike[3] = dBodyCreate(world);  // ERROR!
int key[5] = {3, 6, 7, 1, 5};

string text = "Here is some text to encrypt";
text = stripify_string(text);

for (int i = 0; i < text.length(); i++)
    text[i] = ((text[i] - 'a' + key[i%5]) % 26) + 'a';

cout << "Encrypted: " << text << endl;
2D Arrays

- Often we want to store data with a two-dimensional layout. For example, grey-scale images, where each data value represents an intensity in the image.

- In C++ we can store 2D arrays by adding another subscript.

// 256x256 image

int image[256][256];
3D Arrays

- You can create arrays with any number of dimensions. Using 3D arrays is common for storing color images, where the 3\textsuperscript{rd} dimension stores the color of the pixel, (ie. red, green, blue).

```c
// 256x256 color image
int image[256][256][3];
```
Multi-Dimensional Array Subscripts

• To declare and initialize a multi-dimensional array in C++ is very similar to how it is done with single-dimensional arrays:

```cpp
int board[2][3] = { {0,1,0}, {0,5,6} };
```

• To access the elements in the array you need to give a separate subscript for each of the dimensions:

```cpp
board[0][0]  // equals 0
board[0][1]  // equals 1
board[1][2]  // equals ?
```
Arrays as Data Structures

- 2D arrays can be used to store graphs as an adjacency matrix.
- An adjacency matrix is an nxn matrix $M$, where $M[i,j] = 1$ if there is an edge from vertex $i$ to vertex $j$ and $M[i,j] = 0$ if there is not.
- This is the simplest way to represent a graph, however they a large amount of space.

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 0 & 0 & 0 & 1 \\
2 & 0 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 \\
4 & 0 & 1 & 0 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]
# Array Example

```cpp
#include <iostream>

using namespace std;

int main()
{

    bool graph[5][5] = {
        {0,0,0,0,1},
        {0,0,1,1,1},
        {0,1,0,0,0},
        {0,1,0,0,1},
        {1,1,0,1,0}
    };
}
```
int degree[5] = {0,0,0,0,0};

for( int vertex1 = 0; vertex1 < 5; vertex1++ )
    for( int vertex2 = 0; vertex2 < 5; vertex2++ )
        if( graph[vertex1][vertex2] == 1 )
            degree[vertex1]++;
Traversing a Graph

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph in a systematic way.
- Since any maze can be represented by a graph, where each junction is a vertex and each hallway an edge, any traversal algorithm must be powerful enough to get us out of an arbitrary maze.
Depth-First Search

- The general idea behind graph traversal is to mark each vertex when we first visit it and to keep track of what we have not yet explored.
- The DFS algorithm is as follows:

$$DFS(G,u)$$

state[u] = “discovered”

for each v in Adj[u] do

if state[v] = “undiscovered” then

$$DFS(G,v)$$
Depth-First Search in C++

```c++
int main()
{
    bool graph[8][8] = {
        {0,1,0,0,0,0,0,0},
        {1,0,1,0,1,0,0,0},
        {0,1,0,1,0,0,0,0},
        {0,0,1,0,0,0,0,0},
        {0,1,0,0,0,1,0,1},
        {0,0,0,0,1,0,1,0},
        {0,0,0,0,0,1,0,0},
        {0,0,0,0,1,0,0,0},
    };

    bool state[8] = { false, false, false, false, false, false, false, false };

    DFS( graph, state, 0 );

    return 0;
}
```
void DFS( bool graph[][8], bool state[], int vertex )
{
    state[vertex] = true;

    for( int i = 0; i < 8; i++ )
    {
        if( graph[vertex][i] == 1 && state[i] == false )
        {
            cout << vertex+1 << " -> " << i+1 << endl;
            DFS( graph, state, i );
        }
    }
}
Stacks

- A stack is used to store and retrieve data
- The two fundamental operations of a stack are
  - \textit{PUSH} – Inserts a new data item on the top of the stack
  - \textit{POP} – Removes the data item on the top of the stack
- Stacks support retrieval in last in, first out order (LIFO).
- They are simple to implement and very efficient.
- Stack can be found in many areas of computer science.
Stack Diagrams

```
5

push 12

12

5

push 44

44

12

5

pop

12

5

pop

5
```