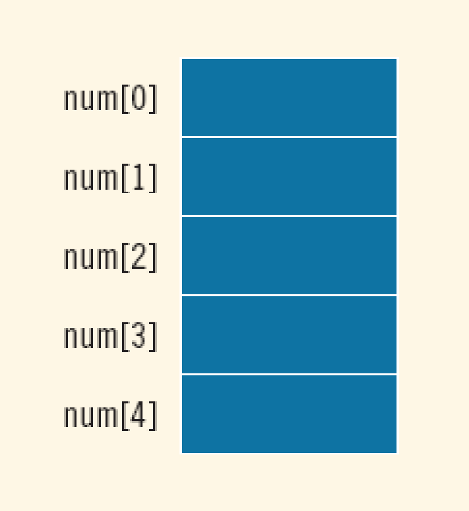
Arrays

Contents

- ○ Array overview
- ○ One-dimensional arrays
- ○ Two-dimensional arrays
- ○ Multi-dimensional arrays
- ○ C-String

Arrays

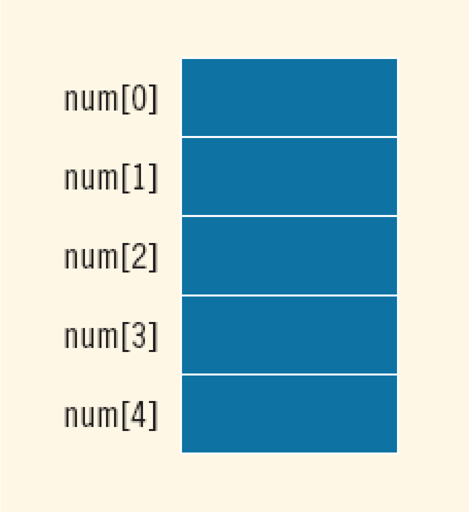
- An array is a collection of items stored at **contiguous** memory locations.
- Elements can be **accessed randomly** using **indices** of an array.
- All elements must be the same data type.

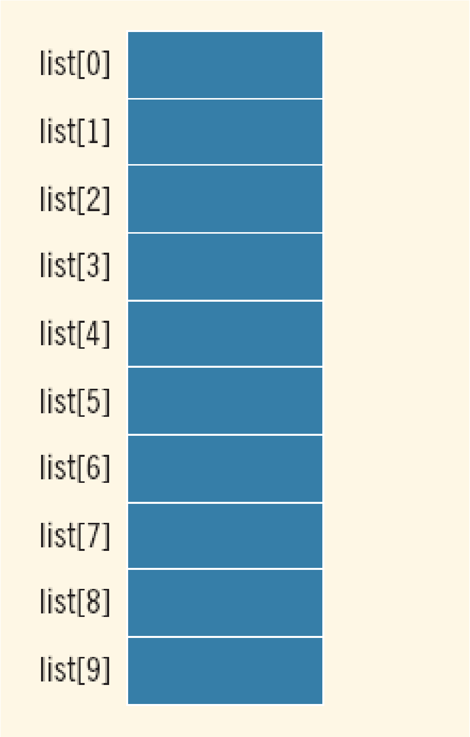


- Used to represent **many instances** in one variable.

Arrays

```
int num[5];
int list[10];
```





Arrays

- One-dimensional arrays
- Two-dimensional arrays
- Multi-dimensional arrays

Advantages

- **Random access** of elements using array index.
- **Easy access** to all the elements.
- **Traversal** through the array becomes easy using a single loop.
- Use of less line of code as it creates a single array of multiple elements.

Disadvantages

- Allows a **fixed number** of elements to be entered
  - decided at the time of declaration.

- Insertion and deletion of elements can be **costly**
  - since the elements are needed to be managed in accordance with the new memory allocation.

Some Facts

- Accessing array elements:
  - Array elements are accessed by using an integer index.
  - Array index **starts with 0** and goes till size of array minus 1. (zero-based index)
- No Index Out of bound Checking:
  - There is no index out of bounds checking (may produce unexpected output when run.)
- The elements are stored at **contiguous** memory locations

Examples

```
// This C++ program compiles fine as index out of bound
// is not checked in C.

#include <iostream>
int main()
{
      int arr[2];
      std::cout << arr[3] << " ";
      std::cout << arr[-2] << " ";
      return 0;
}
```

Examples

```
// Demonstrate that array elements are stored contiguous locations

    #include <iostream>

    int main()
    {
        int arr[5], i;
        std::cout << "Size of integer in this compiler is " << sizeof(int) << std::endl;

        for (i = 0; i < 5; i++)
    // The use of '&' before a variable name, yields address of variable.
                std::cout << "Address arr[" << i << "] is " << &arr[i] << "\n";
        return 0;
    }
```

One-dimensional Arrays

Declarations

- Syntax:

```
Data_Type ArrayName[ArraySize];
```

- Examples:

```
int numbers[10];
float grades[100];
```

Declarations

- Array declaration by specifying size

```
int arr1[10];

// With recent C/C++ versions, we can also
// declare an array of user specified size
int n = 10;
int arr2[n];
```

Declarations

- Array declaration by **initializing** elements

```
int arr[] = { 10, 20, 30, 40 };

// Compiler creates an array of size 4.
// above is same as int arr[4] = {10, 20, 30, 40}
```

Declarations

- // Array declaration by **specifying** size and **initializing** elements

```
int arr[6] = { 10, 20, 30, 40 };

// Compiler creates an array of size 6, initializes
// first 4 elements as specified by user and rest two
// elements as 0 above is same as
// int arr[] = {10, 20, 30, 40, 0, 0};
```

Some Restrictions on Array Processing

- C++ does not allow aggregate operations on an array:

```
int arr[5] = {5, 7, 9, 10, 1};
int other_arr[5];

other_arr = arr; //illegal
```

- Solution:

```
for (int i = 0; i < 5; i++)
    other_arr[i] = arr[i];
```

Some Restrictions on Array Processing

- The following is also illegal:

```
int arr[5];
std::cin >> arr; //illegal
```

- Solution:

```
for (int i = 0; i < 5; i++)
    std::cin >> arr[i];
```

Some Restrictions on Array Processing

- The following is also illegal:

```
int arr[5];
std::cout << arr; //not illegal but not desired result
```

- Solution:

```
for (int i = 0; i < 5; i++)
    std::cout << arr[i] << " ";
```

1D Arrays in Functions

1D Arrays as Function Parameters

- Arrays are passed **by reference** only.
- The symbol **&** is *NOT* used when declaring an array as a formal parameter.
- The size of the array is **usually omitted**
  - If provided, it is ignored by the compiler

```
void zeroFill(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
          arr[i] = 0;
}
```

1D Arrays as Function Parameters

- Other versions:

```
void zeroFill_01(int arr[100], int size)
{
   int i;
   for (i = 0; i < size; i++)
         arr[i] = 0;
}

void zeroFill_02(int* arr, int size)
{
   int i;
   for (i = 0; i < size; i++)
         arr[i] = 0;
}
```

Const Array Parameters

- Using the **`const`** modifier to tell that the array elements are not changed in functions.

```
void printArray(const int arr[], int size)
{
   int i;
   for (i = 0; i < size; i++)
         std::cout << "arr[" << i << "] " << arr[i] << std::endl;
}
```

Functions that Return an Array

- C++ does not allow functions to return a value of the type array.

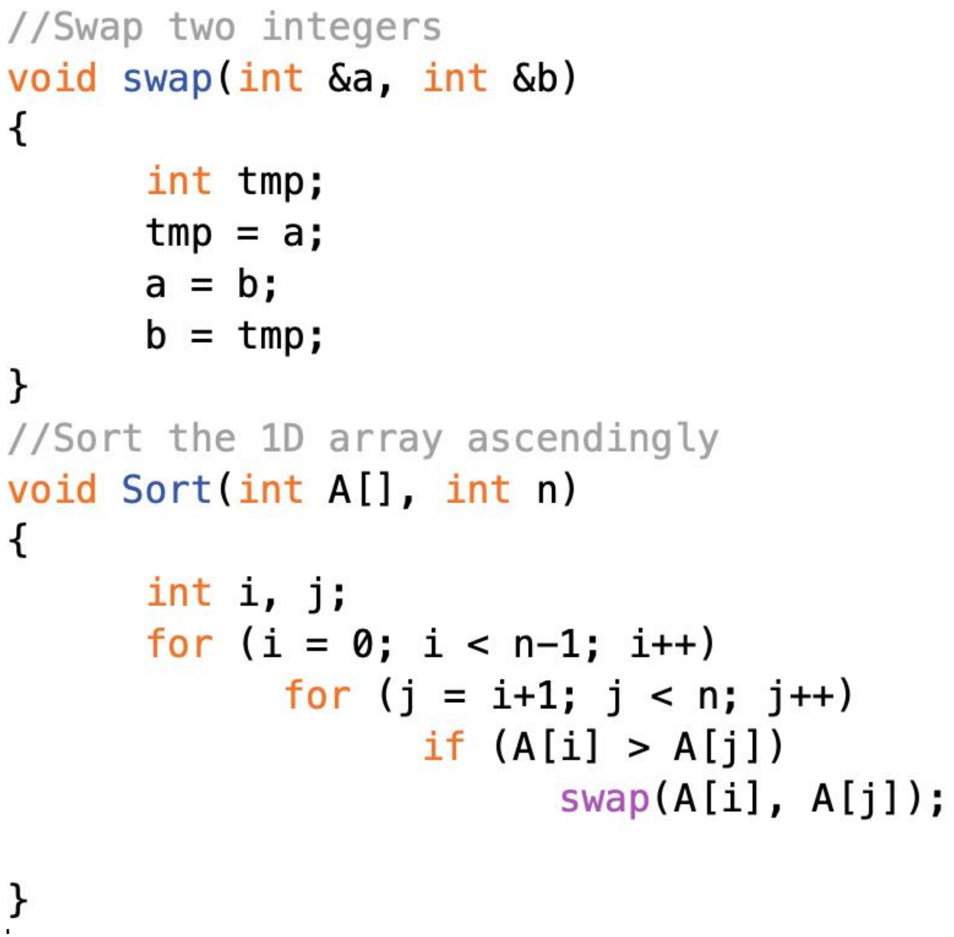
Exercises

- Write functions to find the minimum (maximum) value of an integer array with *n* elements.

- Write a function to find the first position of value *x* in an integer array with *n* elements. If *x* does not exist, the function returns -1.

Sorting

- An example of a ***Sort*** function



```
//Swap two integers
void swap(int &a, int &b)
{
      int tmp;
      tmp = a;
      a = b;
      b = tmp;
}
//Sort the 1D array ascendingly
void Sort(int A[], int n)
{
      int i, j;
      for (i = 0; i < n-1; i++)
            for (j = i+1; j < n; j++)
                  if (A[i] > A[j])
                        swap(A[i], A[j]);

}
```

Two-dimensional Arrays

Two-dimensional Arrays

- **Two-dimensional array:** collection of a fixed number of components (of the same type) arranged in two dimensions.
  - Sometimes called **matrices** or **tables**

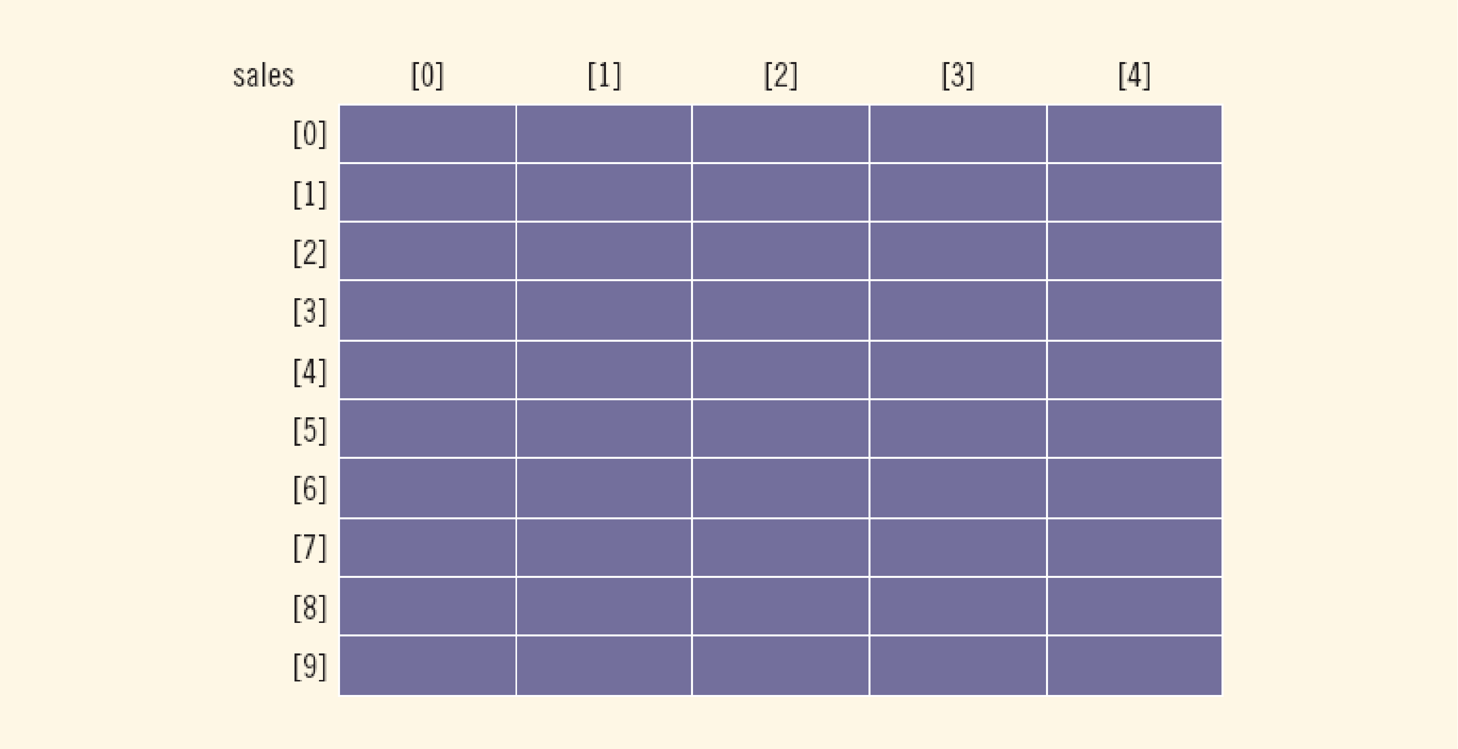
- Declaration syntax:

```
Data_Type ArrayName[ROWSIZE][COLSIZE];
```

where `ROWSIZE` and `COLSIZE` are positive integer values, and specify the **number of rows** and the **number of columns**, respectively, in the array

Two-dimensional Arrays

```
double sales[10][5];
```



Accessing Array Elements

- Syntax:

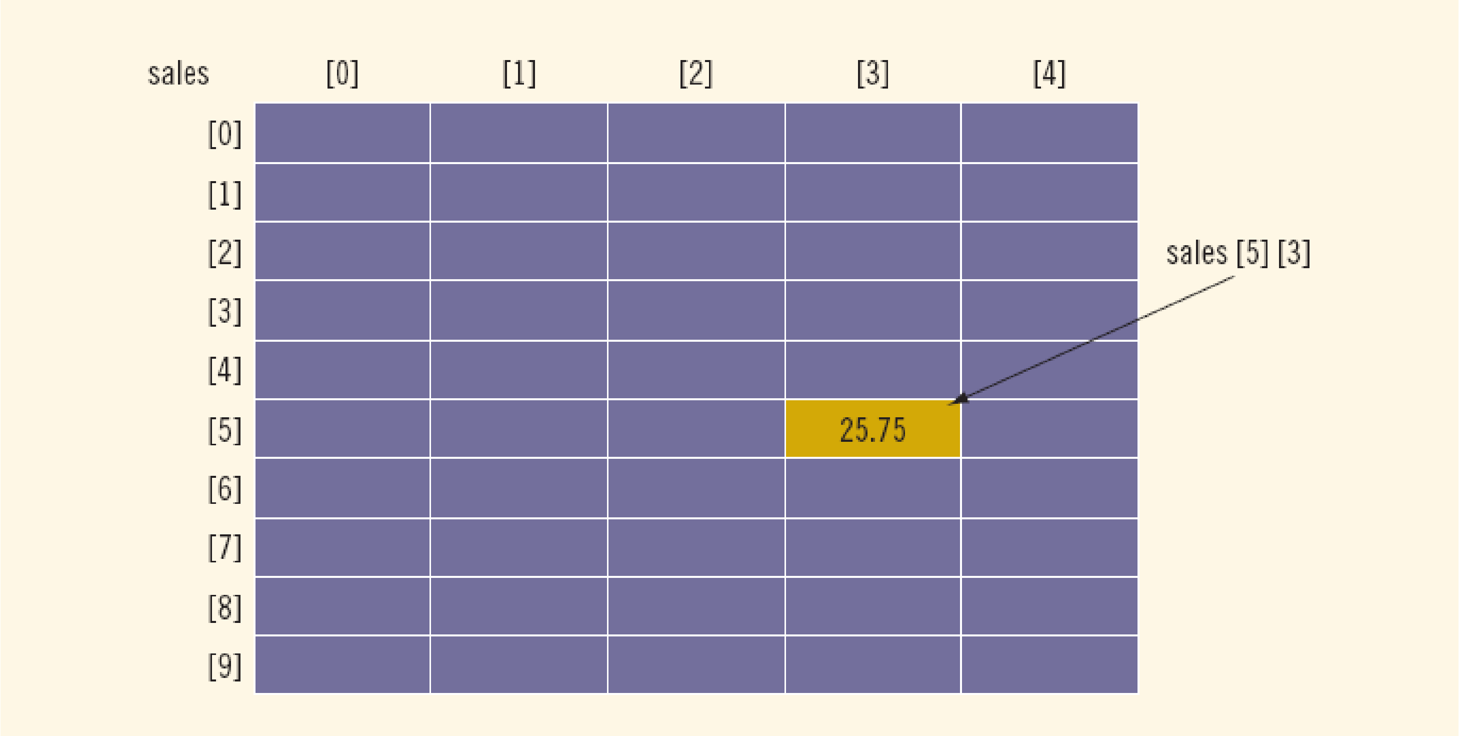
```
arrayName[rowIndex][colIndex]
```

where rowIndex and colIndex are expressions yielding nonnegative integer values, and specify the **row** and **column** position.

The element of arrayName at position (rowIndex, colIndex)

Accessing Array Elements

```
sales[5][3] = 27.75; //row index: 5, col index: 3
```



Two-dimensional Array Initialization

- Example:

```
int Matrix[3][2] = {{1, 5}, {2, 4}, {3,9}};
//3 rows, 2 cols
```

- Elements of each row are enclosed within braces and separated by commas.
- All rows are enclosed within braces.

Processing Two-dimensional Arrays

- Ways to process a two-dimensional array:
  - Process the entire array
  - Process a particular row of the array, called *row processing*
  - Process a particular column of the array, called *column processing*

- Each row and each column of a two-dimensional array is a one-dimensional array
  - To process, use algorithms similar to processing one-dimensional arrays.

Examples

```
int numRows, numCols;
numRows = 7;
numCols = 5;
int array[7][5]; //also: int array[numRows][numCols];

int row, col;
for (row = 0; row < numRows; row++)
  for (col = 0; col < numCols; col++)
      array[row][col] = (row + 1) * (col + 1);
```

Examples

```
int row, col;
for (row = 0; row < numRows; row++)
{
   for (col = 0; col < numCols; col++)
         std::cout << array[row][col] << "\t";
   std::cout << "\n";
}
```

Examples

```
int sum = 0;
int col;
int row = 3;
for (col = 0; col < numCols; col++)
   sum += array[row][col];

std::cout << "Sum of row "<< row << " is " << sum << "\n";
```

Examples

```
int sum = 0;
int col = 4;
int row;
for (row = 0; row < numRows; row++)
   sum += array[row][col];

std::cout << "Sum of column ” << col << " is "
         << sum << "\n";
```

Exercise

- Print the largest/smallest value of each column (row) in a matrix having `nRows` rows, `nCols` columns.

Two-dimensional Arrays as Function Parameters

- Two-dimensional arrays can be passed as parameters to a function
  - Pass by **reference**

- Two-dimensional arrays are stored in row order.
- When declaring a two-dimensional array as a formal parameter, **can omit size of first dimension, but not the second**.

Two-dimensional Arrays as Function Parameters

```
#define COLSIZE      100

void InitArray(int array[][COLSIZE], int numRows, int  numCols)
{
   int row, col;
   for (row = 0; row < numRows; row++)
         for (col = 0; col < numCols; col++)
               array[row][col] = (row + 1) * (col + 1);
}
```

Two-dimensional Arrays as Function Parameters

```
#define COLSIZE      100

void PrintArray(const int array[][COLSIZE], int numRows, int  numCols)
{
    int row, col;
    for (row = 0; row < numRows; row++)
    {
        for (col = 0; col < numCols; col++)
            std::cout << array[row][col] << "\t";
        std::cout << "\n";
    }
}
```

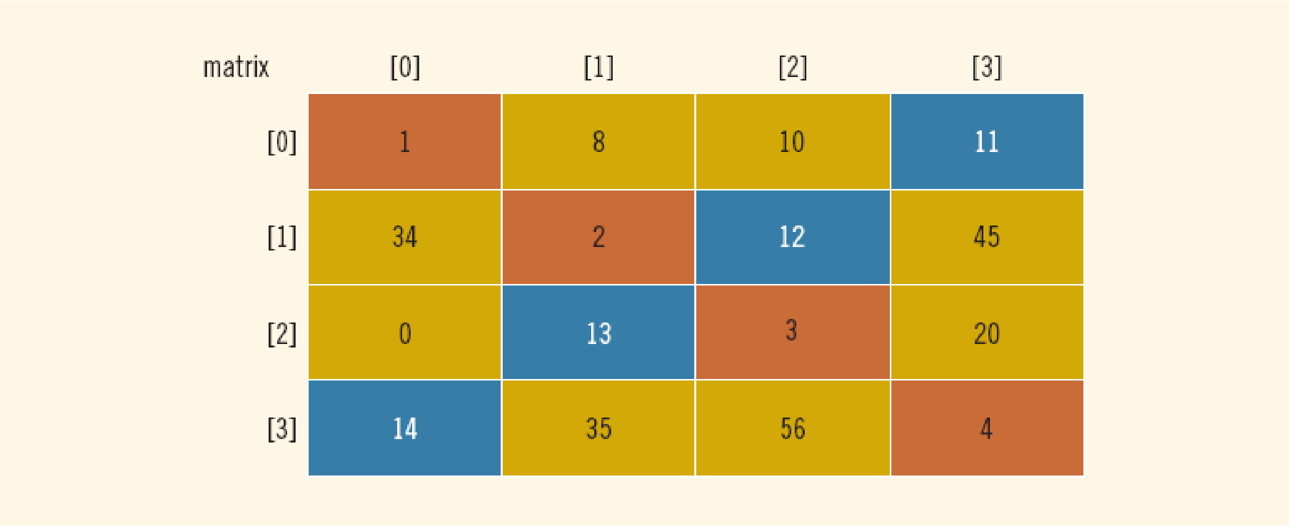
Two-dimensional Arrays as Function Parameters

```
int main()
{
    int rows, cols;
    rows = 5;
    cols = 7;

    int matrix[rows][COLSIZE];
    //or: int matrix[ROWSIZE][COLSIZE];

    InitArray(matrix, rows, cols);
    PrintArray(matrix, rows, cols);
    return 0;
}
```

Diagonal

- ○ Main diagonal

- ○ Opposite diagonal



matrix	[0]	[1]	[2]	[3]
[0]	1	8	10	11
[1]	34	2	12	45
[2]	0	13	3	20
[3]	14	35	56	4

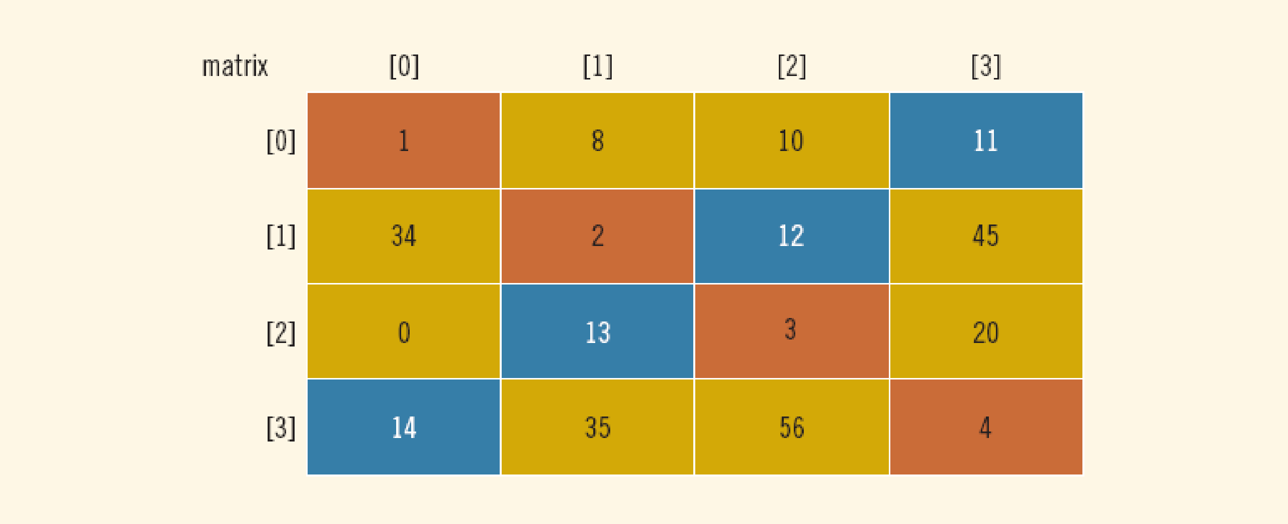
Diagonal

- Main diagonal

```
Array[i][i]
```

- Opposite diagonal

```
Array[i][Size - i - 1]
```



matrix	[0]	[1]	[2]	[3]
[0]	1	8	10	11
[1]	34	2	12	45
[2]	0	13	3	20
[3]	14	35	56	4

Exercises

- Print the values in the main diagonal of a matrix `N x N`.

- Print the values in the opposite diagonal of a matrix `N x N`.

Transpose a 2D Array

- The *transpose* of a matrix is a new matrix whose rows are the columns of the original.
  - This makes the columns of the new matrix the rows of the original.
  - The element at row `r` column `c` in the original is placed at row `c` column `r` of the transpose. The element **`a[r][c]`** of the original matrix becomes element **`a[c][r]`** in the transposed matrix.

$$\begin{pmatrix} 5 & 4 & 3 \\ 4 & 0 & 4 \\ 7 & 10 & 3 \end{pmatrix}^T = \begin{pmatrix} 5 & 4 & 7 \\ 4 & 0 & 10 \\ 3 & 4 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 5 & 4 \\ 4 & 0 \\ 7 & 10 \\ -1 & 8 \end{pmatrix}^T_{4 \times 2} = \begin{pmatrix} 5 & 4 & 7 & -1 \\ 4 & 0 & 10 & 8 \end{pmatrix}_{2 \times 4}$$

Exercises

- ○ Write user defined functions for square matrix to calculate
  - Left diagonal sum
  - Right diagonal sum

- ○ Write a program to add two array A and B of size m x n.

Exercises

- Write a function named `Upper-half` which takes a two-dimensional array A, with size N rows and N columns as argument and prints the upper half of the array.

```
2 3 1 5 0                     2 3 1 5 0
7 1 5 3 1                       1 5 3 1
2 5 7 8 1   Output will be:       7 8 1
0 1 5 0 1                           0 1
3 4 9 1 5                             5
```

Exercises

- Write a function which accepts a 2D array of integers and its size as arguments and displays the elements of middle row and the elements of middle column. Assuming the 2D Array to be a square matrix with odd dimension i.e. 3x3, 5x5, 7x7 etc...
- Example, if the array contents is

  3 5 4
  7 6 9
  2 1 8
- Output through the function should be :
  - Middle Row : 7 6 9
  - Middle column : 5 6 1

Multi-dimensional Arrays

Multi-dimensional Arrays

- **Multi-dimensional array:** collection of a fixed number of elements (called components) arranged in *n* dimensions (n >= 1)
  - Also called an *n*-dimensional array

- Declaration syntax:

```
Data_Type ArrayName[Dim1Size][Dim2Size]..[DimNSize];
```

- To access a component:

```
ArrayName[Dim1Idx][Dim2Idx]..[DimNIdx]
```

Multi-dimensional Arrays

- When declaring a multi-dimensional array as a formal parameter in a function
  - Can **omit size of first dimension** but not other dimensions
- As parameters, multi-dimensional arrays are passed by reference only
- A function cannot return a value of the type array
- There is no check if the array indices are within bounds

Character Arrays

C-strings (Character Arrays)

- Character array: an array whose components are of type `char`
- `C`-strings are null-terminated (`'\0'`) character arrays

- Example:
  - `'A'` is the character `A`
  - `"A"` is the `C`-string `A`
    - `"A"` represents two characters, `'A'` and `'\0'`

C-strings (Character Arrays)

- Consider the statement

```
char s[10];
```

- Since C-strings are null terminated and s has **10** components, the largest string that it can store has **9** characters
- If you store a string of length **7** in s
  - The first 8 components of s are used and the last two are left unused

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	i		M	o	m	!	\0	?	?

C-strings (Character Arrays)

- The statement

```
char name[16] = "John";
```

declares an array name of length 16 and stores the C-string "John" in it

- The statement

```
char name[] = "John";
```

declares an array name of length 5 and stores the C-string "John" in it

The `<cstring>` Library

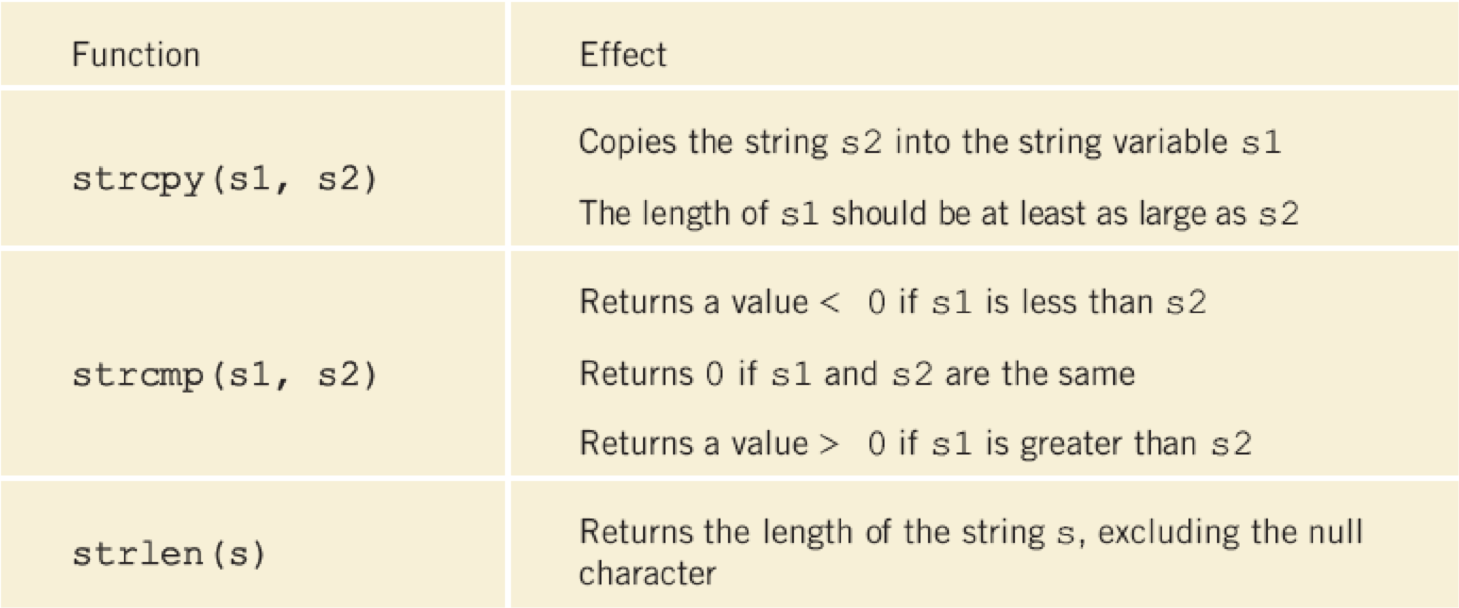
- String predefined string functions in `cstring` library.

  ```
  #include <cstring>
  ```

- Definitions in <`cstring`> are placed in global namespace.
  - Do not require `using namespace`

C-strings (Character Arrays)

- Some functions:

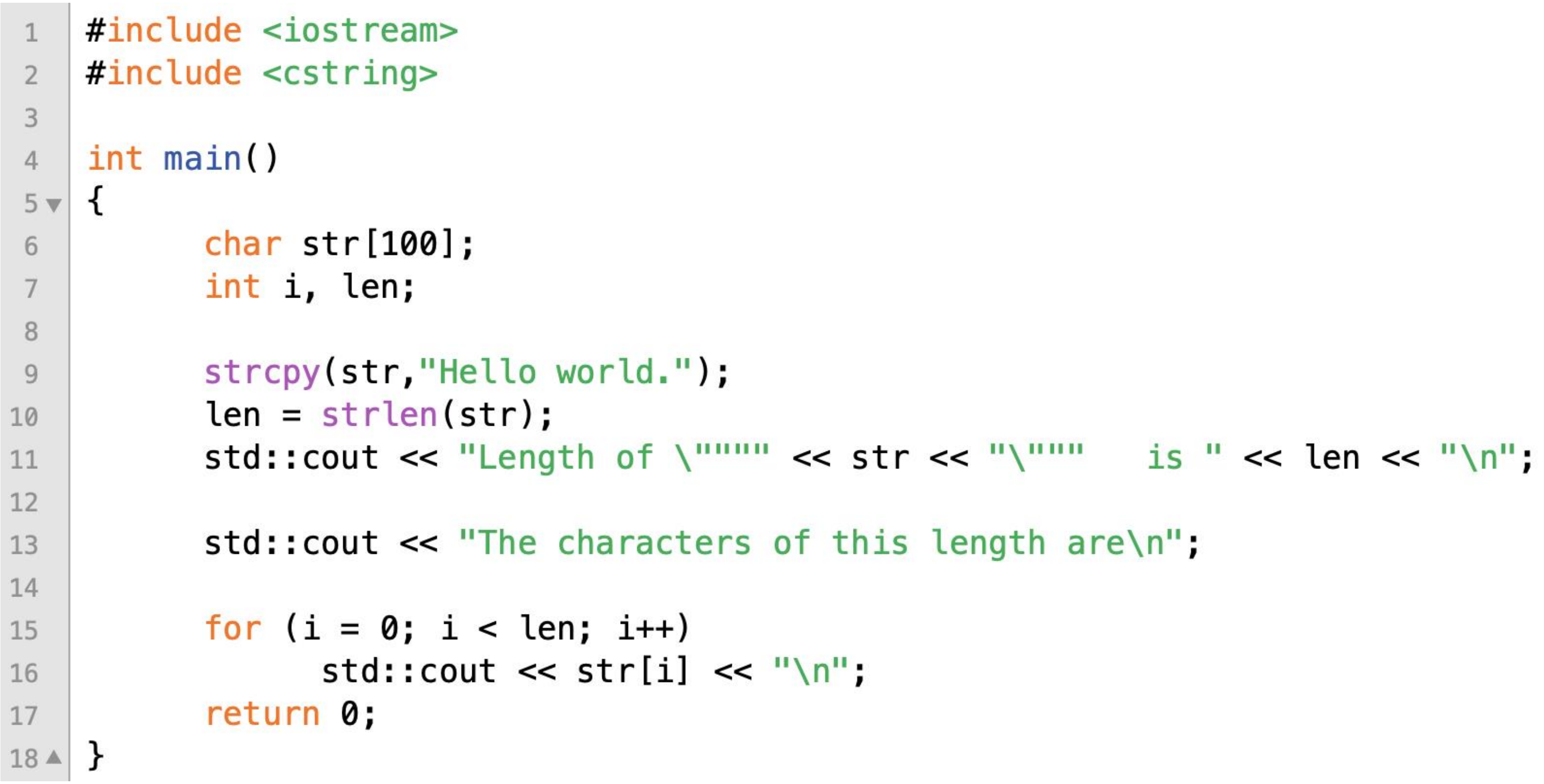


Function	Effect
`strcpy(s1, s2)`	Copies the string s2 into the string variable s1 The length of s1 should be at least as large as s2
`strcmp(s1, s2)`	Returns a value < 0 if s1 is less than s2 Returns 0 if s1 and s2 are the same Returns a value > 0 if s1 is greater than s2
`strlen(s)`	Returns the length of the string s, excluding the null character

Input Using std::cin

```
#include <iostream>
#include <cstring>
int main()
{
	char name[80];
	std::cout << "Input your name: ";
	std::cin.getline(name, 80);
	std::cout << "Your name is " << name << "\n";
	return 0;
}
```

Example



```cpp
#include <iostream>
#include <cstring>

int main()
{
      char str[100];
      int i, len;

      strcpy(str,"Hello world.");
      len = strlen(str);
      std::cout << "Length of \"""" << str << "\"""   is " << len << "\n";

      std::cout << "The characters of this length are\n";

      for (i = 0; i < len; i++)
            std::cout << str[i] << "\n";
      return 0;
}
```

String Comparison

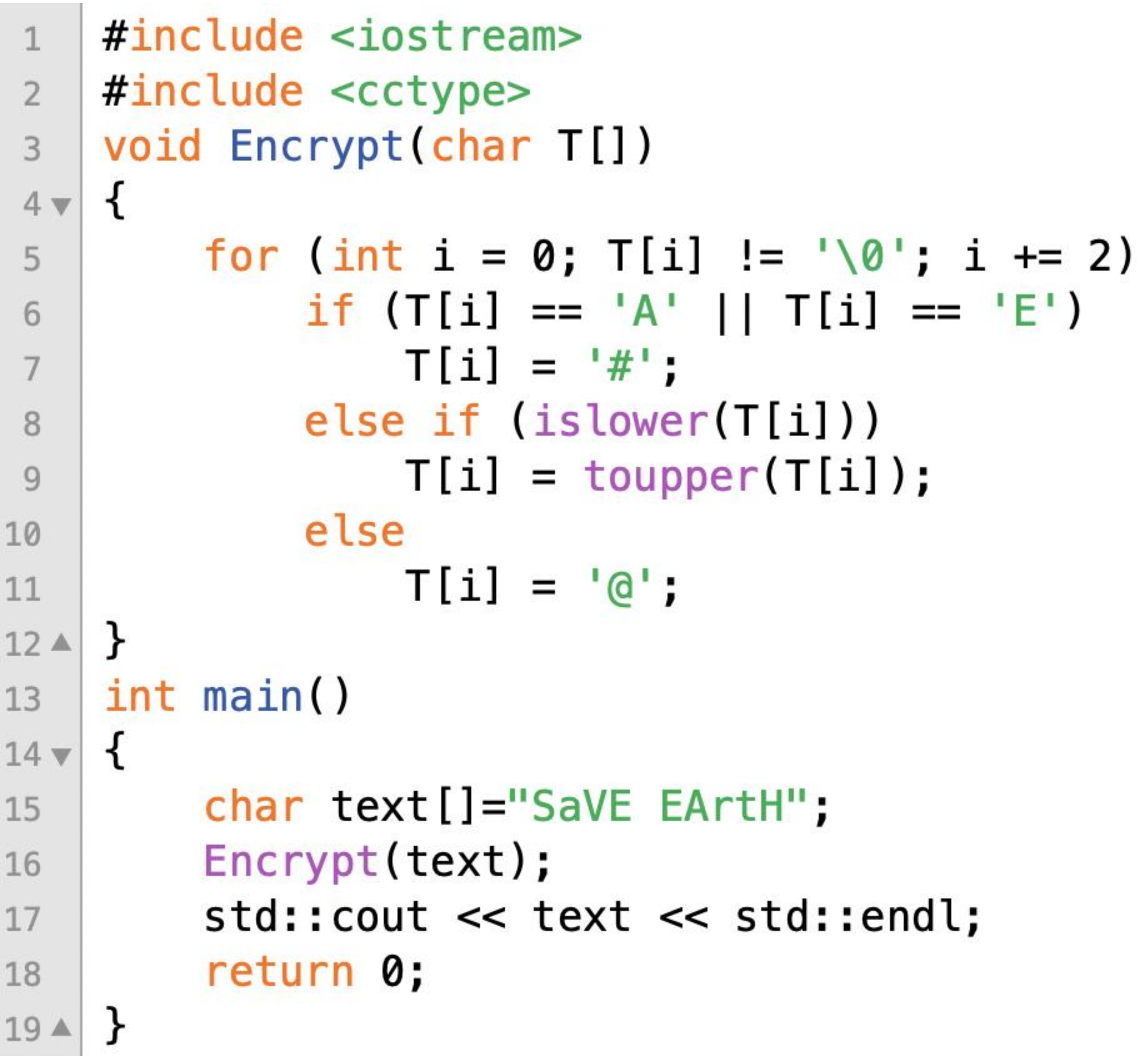
- C-strings are compared character by character using the collating sequence of the system.

- The ordered relationship is called **lexicographic order**.
- If we are using the ASCII character set
  - `"Air"` < `"Boat"`
  - `"Air"` < `"An"`
  - `"Billy"` > `"Bill"`
  - `"hello"` > `"Hello"`

Character-Manipulating Functions

- Library with header file <`cctype`>.
- Some functions (page 396):
  - `toupper`
  - `tolower`
  - `isupper`
  - `islower`
  - `isalpha`
  - `isdigit`
  - `isspace`
  - `ispunct`

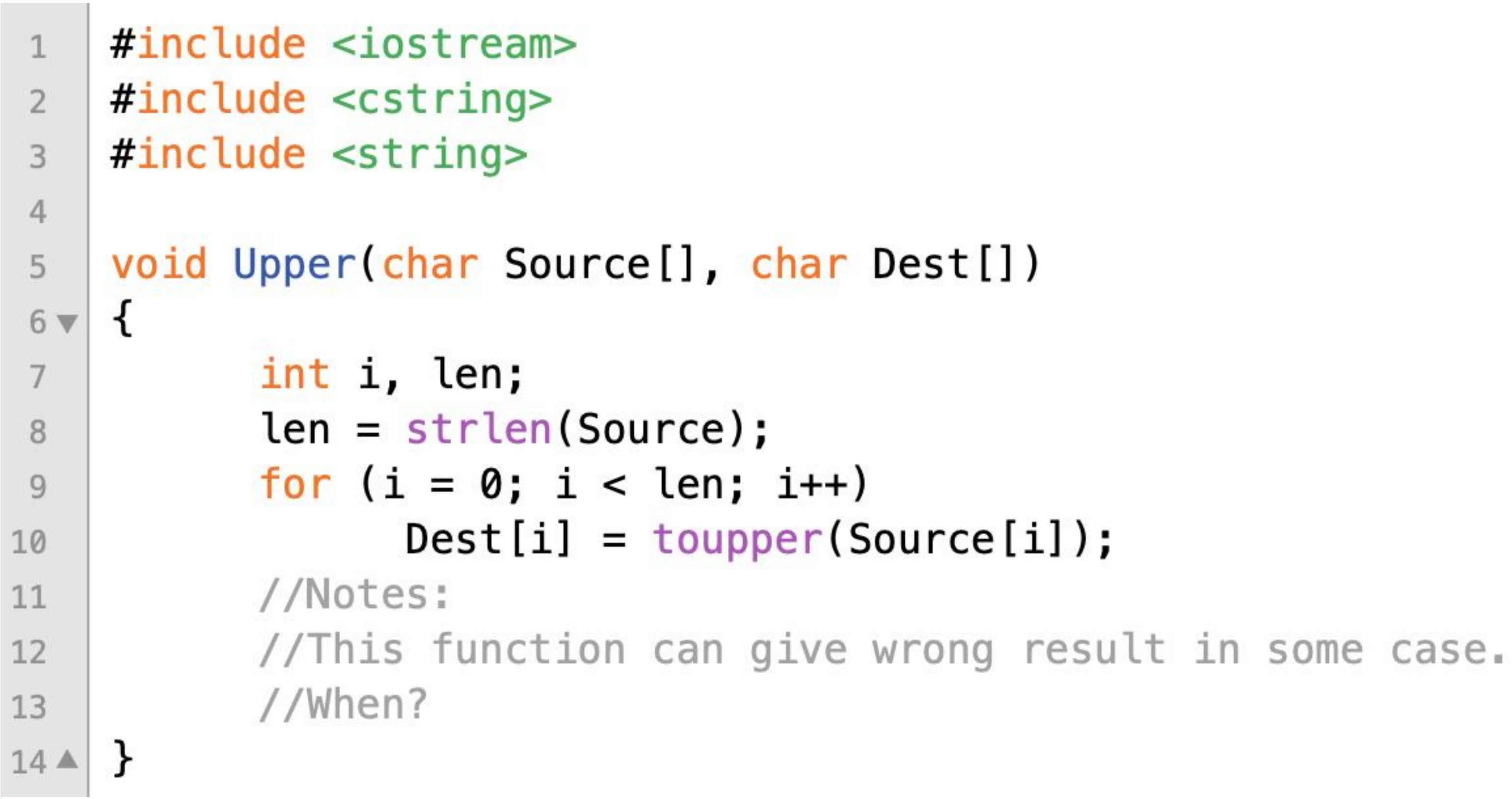
Examples



```cpp
#include <iostream>
#include <cctype>
void Encrypt(char T[])
{
    for (int i = 0; T[i] != '\0'; i += 2)
        if (T[i] == 'A' || T[i] == 'E')
            T[i] = '#';
        else if (islower(T[i]))
            T[i] = toupper(T[i]);
        else
            T[i] = '@';
}
int main()
{
    char text[]="SaVE EArtH";
    Encrypt(text);
    std::cout << text << std::endl;
    return 0;
}
```

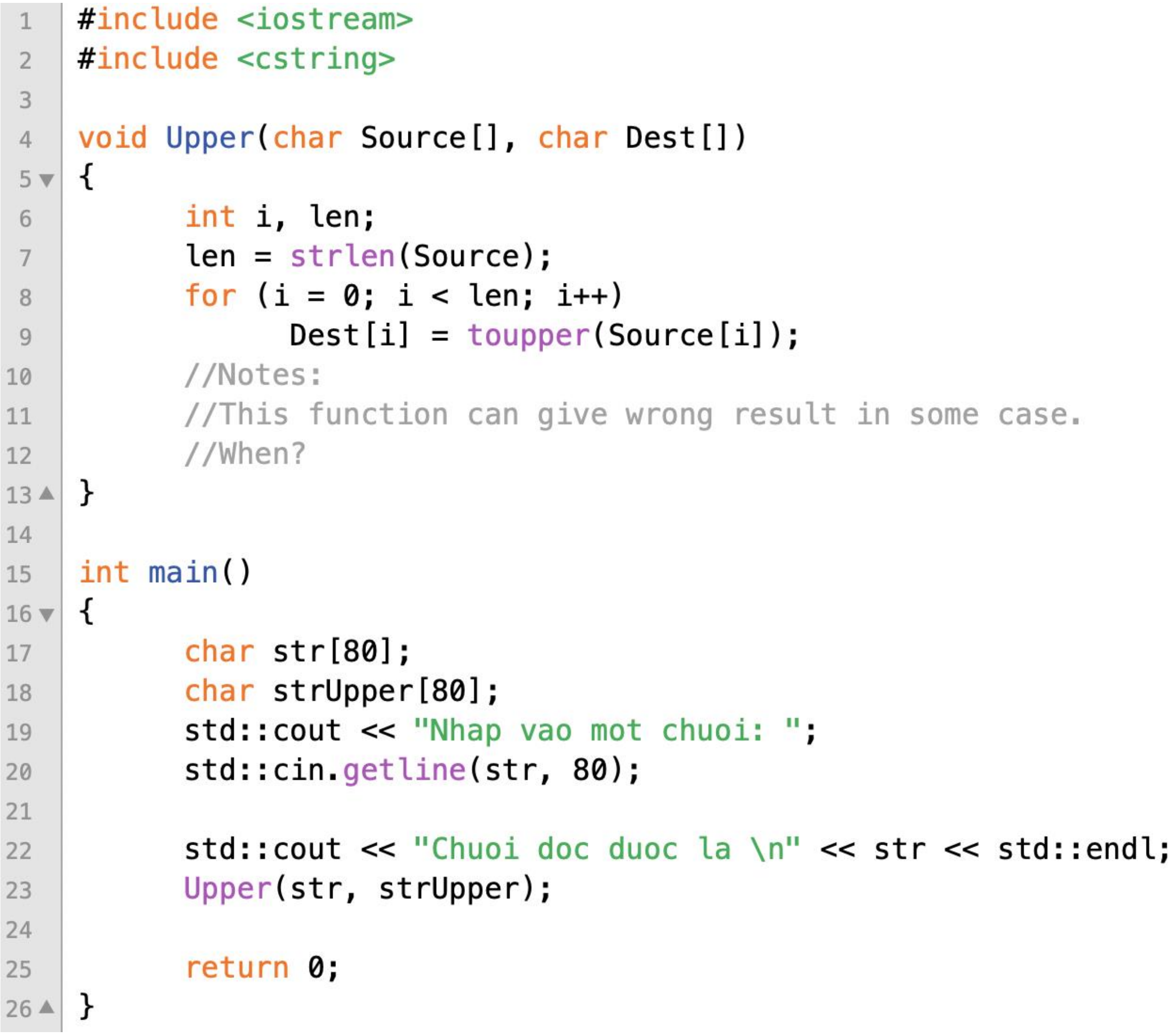
Examples

- Can this function work well?



```
#include <iostream>
#include <cstring>
#include <string>

void Upper(char Source[], char Dest[])
{
      int i, len;
      len = strlen(Source);
      for (i = 0; i < len; i++)
            Dest[i] = toupper(Source[i]);
      //Notes:
      //This function can give wrong result in some case.
      //When?
}
```

Examples



```cpp
#include <iostream>
#include <cstring>

void Upper(char Source[], char Dest[])
{
    int i, len;
    len = strlen(Source);
    for (i = 0; i < len; i++)
        Dest[i] = toupper(Source[i]);
    //Notes:
    //This function can give wrong result in some case.
    //When?
}

int main()
{
    char str[80];
    char strUpper[80];
    std::cout << "Nhap vao mot chuoi: ";
    std::cin.getline(str, 80);

    std::cout << "Chuoi doc duoc la \n" << str << std::endl;
    Upper(str, strUpper);

    return 0;
}
```

Questions and Answers