C++ Refresh

David W.

2024-06-09

Section 1: C++ Introduction

What is C++

C++ is a general purpose programming language that is free-form and compiled. It is regarded as an intermediate-level language, as it comprises both high-level and low-level language features. It provides imperative, object-oriented and generic programming features.

The overall program has a structure, but it is also important to understand the purpose of part of that structure.

101 step by step

```
#include<iostream>

using namespace std;

int main()  {
    // YOUR CODE GOES HERE
    // Please take input and print output to standard input/output (stdin/stdout)
    // E.g. 'cin' for input & 'cout' for output
    cout << "Hello, InterviewBit!" << endl;
    return 0;
}
```

preprocessor

```
#include<iostream>
```

- The hash sign (#) signifies the start of a preprocessor command.
- The #include command is a specific preprocessor command that effectively copies and pastes the entire text of the file specified between the angle brackets into the source code. In this case, the file is "iostream" which is a standard file that should come with the C++ compiler. This file name is short for "input-output streams"; in short, it contains code for displaying and getting the text from the user.
- The include statement allows a programmer to "include" this functionality in the program without having to literally cut and paste it into the source code every time.
- The iostream file is part of the C++ standard library, which provides a set of useful and commonly used functionality provided with the compiler. The "include" mechanism, however, can be used both for standard code provided by the compiler and for reusable files created by the programmer.

namespace

```
using namespace std;
```

C++ supports the concept of namespaces. A namespace - is essentially a prefix that is applied to all the names in a certain set. - like toolboxes with different useful tools.

The using command tells the compiler to allow all the names in the "std" namespace to be usable without their prefix.

The iostream file defines three names used in this program - cout, cin, and endl

Those three are all defined in the std namespace. "std" is short for "standard" since these names are defined in the standard C++ library that comes with the compiler.

Without using the std namespace, the names would have to include the prefix and be written as std::cout, std::cin, and std::endl.

main

```
int main()  {
```

The starting point of all C++ programs is the main function. This function is called by the operating system when your program is executed by the computer.

the three command in std namespace

```
cout << "Hello, InterviewBit!" << endl;
```

cin & cout - cout is character output - cin is character input

In a typical C++ program, most function calls are of the form object.function_name(argument1, argument2).

Symbols such as « can also behave like functions, as illustrated by the use of cout above. This capability is called operator overloading.

{ } A block of code is defined with the { } tokens.

semicolons Statements in C++ **must be terminated with a semicolon.**

return

The return keyword tells the program to return a value to the **function int main** that called this function, and then to continue execution in the int main function from the point at which this function was called.

The **type of the value returned by a function** must match **the type specified in the declaration** of the function.

Common data types in cpplus

built-in data types are described as follows:

- Int ("%d"): 32 Bit integer
- Long ("%ld"): 64 bit integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- Char ("%c"): Character type, can range only from -128 to 127
- Boolean (either true or false)
- Float ("%f"): 32 bit real value
- Double ("%lf"): 64 bit real value

User Defined data types are - structures (struct) - classes (class). This will be covered later

set precision for float and double

To print float and double up to specific number of decimal places use:

cout«std::fixed « std::setprecision(number of decimal places) - setprecision() is available in the **iomanip** library.

Type Modifiers

The above types can be modified using the following type modifiers: - signed - unsigned - short - long

Typedefs

Creating new names (i.e. aliases) for existing types. Following is the simple syntax to define a new type using typedef:

```
typedef int item
item number = 0 // number is a integer variable
```

As we have use typedef and created new name for int as item, item is equivalent to int now

Variables

a variable is a container (storage area) to hold data.

int val = 10; val = 15;

Exercise 1.2

Try the following example in the editorial below. You have to input 5 space-separated values: int, long, char, float and double respectively.

Print each value on a new line in the same order it is received as input. Note that the floating point value should be correct up to 3 decimal places and the double to 9 decimal places.

Example Input:

5 1234567891231 z 24.23 1214523.028352

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

int main()  {
    int int_1; long long_1; char char_1; float float_1; double double_1;
    cin >> int_1;
    cin >> long_1;
    cin >> char_1;
    cin >> float_1;
    cin >> double_1;

    cout << int_1 << endl << long_1 << endl << char_1 << endl << fixed << setprecision(3) << float_1 <<

    return 0;
}
```

Type Conversion

C++ allows us to convert data of one type to that of another. This is known as type conversion.

There are two types of type conversion in C++.

Implicit Conversion Explicit Conversion (also known as Type Casting)

Implicit Type Conversion

The type conversion that is done automatically done by the compiler is known as implicit type conversion. This type of conversion is also known as automatic conversion.

```
int a = 10;
char b = 'A';
a = a + b; // y implicitly converted to int. ASCII value of 'A' is 65
cout<<a<<endl;
// value of a is 75(10 + 65)

int num_int;
double num_double = 10.79;
// implicit conversion
// assigning a double value to an int variable
num_int = num_double;
// Value of num_int will be 10
// Here, the double value is automatically converted to int by the compiler before it is assigned to th
```

Explicit Conversion

When the user manually changes data from one type to another, this is known as explicit conversion. This type of conversion is also known as type casting.

There are two major ways in which we can use explicit conversion in C++. They are:

- C-style type casting (also known as cast notation)
- type conversion operators

C-style Type Casting

defining the required type **in front of the expression in parenthesis, i.e. (data_type)expression**. This can be also considered as forceful casting.

(data_type)expression;

```
int a = 10;
char b = 'A';
a = a + (int)b;
cout<<a<<endl;
// value of a is 75
```

Type conversion operators

C++ also has four operators for type conversion: - static_cast: static_cast(variable) - dynamic_cast - const_cast - reinterpret_cast

```
float f = 4.5;
// using cast operator
int b = static_cast<int>(f);
cout << b;
// value of b is 4
To learn more about typecasting click http://www.cplusplus.com/doc/tutorial/typecasting/
```

Exercise 1.3

You are given a character called ch, print the ASCII value of the character.

```
#include<iostream>

using namespace std;

int main()  {
    char ch;
    cin>>ch;
    //Your code goes here
    int a;
    a =  int(ch);
    cout << a << endl;
    return 0;
}
```

Math Library

The C++ math library is actually C's math library. It is easy to use and is accessed by including cmath.

```
#include <cmath>
```

Note: Trigonometric functions in cmath use RADIANS.

Exercise 1.4

You are given two float variables A and B, perform the operations defined in comments in the editor below.

```
#include<iostream>
#include<cmath>
using namespace std;
int main()  {
    float A = 12.56, B = 5.12;
    // Print the sum of cube of both A and B, and store it in float variable named "cube_val"
    float res_1 = pow(A,3) + pow(B,3);
    cout <<  res_1 << endl;

    // Print the square root of cube_val, and store it in float variable named "sq_val"
    float result_2 = sqrt(res_1);
    cout << result_2 << endl;

    // Print the sin of sq_val
    cout << sin(result_2) << endl;

    return 0;
}
```

File Handling

Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file, and to perform various operations on it.

In C++, files are mainly dealt by using three classes **fstream, ifstream, ofstream** available in the headerfile **fstream**.

- ofstream: Stream class, to write on files

- ifstream: Stream class, to read from files
- fstream: Stream class, to both read and write from/to files.

Opening a file

```
void open(const char* file_name,ios::openmode mode);
```

The first argument of the open function **const char* file_name)** defines the name, and format of the file with the address of the file. The second argument **ios::openmode mode** represents the mode in which the file has to be opened.

- in: File open for reading: the internal stream buffer supports input operations.
- out: File open for writing: the internal stream buffer supports output operations.
- binary: Operations are performed in binary mode rather than text.
- ate: The output position starts at the end of the file.
- app: All output operations happen at the end of the file, appending to its existing contents.
- trunc: Any contents that existed in the file before it is open are discarded.

Default Open Modes:

ifstream ios::in ofstream ios::out fstream ios::in | ios::out Note: You can combine two or more of these values by | them together.

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Example using ifstream & ofstream classes.

```
#include <iostream>
#include <fstream>

using namespace std;

// Driver Code
int main()
{
    ofstream fout; // Creation of ofstream class object
    string line;

    // by default ios::out mode, automatically deletes
    // the content of file. To append the content, open in ios:app
    // fout.open("sample.txt", ios::app)
    fout.open("sample.txt");

    // Execute a loop If file successfully opened
    while (fout) {
        getline(cin, line);  // Read a Line from standard input

        if (line == "-1") // Enter -1 to exit
            break;

        fout << line << endl; // Write line in file
    }
    fout.close(); // Close the File
```

```
    ifstream fin; // Creation of ifstream class object to read the file
    fin.open("sample.txt");  // by default open mode = ios::in mode

    // Execute a loop until EOF (End of File)
    while (fin) {
        getline(fin, line); // Read a Line from File
        cout << line << endl;  // Print line in Console
    }

    // Close the file
    fin.close();
    return 0;
}
```

Random Numbers

rand() - rand() function is used to generate random numbers.

```
int rand(void):
cout<<rand()<<endl; // Outputs any random number
```

returns a pseudo-random number in the range of 0 to RAND_MAX. RAND_MAX: is a constant whose default value may vary between implementations but it is granted to be at least 32767.

srand() sets the starting point for producing a series of pseudo-random integers. - If srand() is not called, the rand() seed is set as if srand(1) were called at program start.

```
void srand( unsigned seed ):
```

Note: The pseudo-random number generator should only - be seeded once, - before any calls to rand(), - and the start of the program.

It should not be - repeatedly seeded, or - reseeded every time you wish to generate a new batch of pseudo-random numbers.

Standard practice is to use the result of a call to **srand(time(0))** as the seed.

```
int main(){
    // This program will create different sequence of
    // random numbers on every program run

    // Use current time as seed for random generator
    srand(time(0));

    for(int i = 0; i<4; i++)
        cout<<rand()<<" ";
    return 0;
}
```

Section 2: Flow Control

Comparision Operation & If-Else

- comparison operators such as ** ==, !=, >, <, ** etc can be used in C++
- These operators (cause the immediate statement in which they are contained to) return a Boolean value of either true or false.

Comparision Operator for all primitive data type (int, char, float, bool, etc.): **== and !=**

Comparision Operator for numeric data types only (int, float, double etc.): **>, <, >=, <=**

Conditional statements

2 general types - if, else / else if, else - "switch. . . case" construct

Example 2.1:

Given an integer num denoting percentage of a student, calculate the grade according to the below rules:

If num >= 90, grade A. If num >= 80, grade B. If num >= 70, grade C. If num >= 60, grade D. If num >= 50, grade E. Else grade will be F. Print a string consisting of single character denoting the grade.

```
#include<iostream>

using namespace std;

int main()  {
    int num;
    cin>>num;
    // Your code goes here
    char result;
    if (num >= 90){
        result = 'A';
    //console.log("A")
    }
    else if (num >= 80){
        result = 'B';
    }
    else if (num >= 70){
        result = 'C';
    }
    else if (num >= 60){
        result = 'D';
    }
    else if (num >= 50){
        result = 'E';
    }
    else {
        result = 'F';
    }
    cout << result << endl;

    return 0;
}
```

Switch statement

```
switch (expression) {
    case constant1:
        // code to be executed if expression is equal to constant1;
        break;
```

```
    case constant2:
        // code to be executed if expression is equal to constant2;
        break;

    default:
        // code to be executed if
        // expression doesn't match any constant
}
```

If there is a match, the corresponding code after the matching label is executed until the break statement is encountered.

```
int x = 2;
switch (x)
{
    case 1:
        cout << "Choice is 1";
        break;
    case 2:
        cout << "Choice is 2";
        break;
    case 3:
        cout << "Choice is 3";
        break;
    default:
        cout << "Choice other than 1, 2 and 3";
        break;
}
```

Exercise 2.2

```
#include<iostream>

using namespace std;

int main()  {
    int weekday;
    cin>>weekday;
    // YOUR CODE GOES HERE
    string str_wkd;
    int numeric_input = int(weekday);
    switch (numeric_input){
        case 1:
            str_wkd = "Monday";
            break;
        case 2:
            str_wkd = "Tuesday";
            break;
        case 3:
            str_wkd = "Wednesday";
            break;
        case 4:
            str_wkd = "Thursday";
            break;
```

```
        case 5:
            str_wkd = "Friday";
            break;
        case 6:
            str_wkd = "Saturday";
            break;
        case 7:
            str_wkd = "Sunday";
            break;
    }
    cout << str_wkd << endl;
    return 0;
}
```

Conditional or Ternary Expression

```
( condition ) ? expressionIfTrue : expressionIfFalse;
```

First the condition is evaluated, and the side effects of this evaluation carry out their impact on the local environment.

If the result is true then only the **expressionIfTrue** is evaluated (causing side effects) and this second result is the value of the whole conditional expression, and the expressionIfFalse is not evaluated (and hence cause no side effects).

If the condition evaluates to false, then the situation is converse

A common use of the conditional expression: - assign the value x or y to a, depending on an easily decidable condition, say x > y.

```
int x = 7;
int y = 5;
int a = ( x > y ) ? x : y;
cout << a << endl;
```

Exercise 2.3

Given two integer x and y, using Conditional or Ternary expression, print “Robin” if the value of x is less than or equal to y else print “Rahul” (without quotes).

```
#include<iostream>

using namespace std;

int main()  {
    int x, y;
    cin>>x>>y;
    // YOUR CODE GOES HERE
    string ans = ( x <= y )? "Robin" : "Rahul";

    cout << ans << endl;

    return 0;
}
```

Loops

3 Types of loop: 1. for 2. while 3. do. . . while

for loop

for (initialization; condition; update) { // body of-loop } for(int i = 1; i <= 5; i++){ cout«i«" "; } 1. initialization - initializes variables and is executed only once 2. condition - if true, the body of for loop is executed, if false, the for loop is terminated 3. update - updates the value of initialized variables and again checks the condition

while loop

while (condition) { statememt(s); } int i = 1;

while(i <= 5){ cout«i«" "; i++; }

do. . . while loop

- do. . . while loop checks the conditional statement after the first run, then continuing onto another iteration.
- A do-while loop is used where your loop should execute at least one time. Ex: take an integer input from the user until the user has entered a positive number.

do { //body } while (condition);

int i = 1; do { cout « i « " "; i++; } while(i <= 5); // the contition is being checked after the first run

Exercise 2.4

```
#include<iostream>

using namespace std;

int main()  {
    int N;
    cin>>N;
    // YOUR CODE GOES HERE
    int i = 0;
    while( i <= N) {
        if(i % 2 != 0) {
            cout << i << endl;
        }
        i++;
    }
    return 0;
}
```

Jump Statements

Jump statements are used to **interrupt the normal flow** of program.

Types of Jump Statements: 1. break 2. continue 3. goto

Break Statement

- used inside loop, or switch statement.

- compiler will abort the loop and continue to execute statements followed by loop. int a = 1;

```
while(a <= 10)
{
if(a==5)
break;
a++;
}
cout « “Value of a is” «a«endl;
// Value of a is 5
```

Continue Statement

- inside loop
- skip the followling statements in the current loop, and resume the loop

```
int a = 0;
while(a < 10)
{
a++;
if(a == 5)
continue;
cout << a << " ";
}
// prints 1 2 3 4 6 7 8 9 10
```

Goto Statement

- jumps from one point **to another point within a function**. goto label_1; (a) . . . .
  . . . .
  . . . .
  label_1: (b) statement;
  . . . .
- When **goto label_1;** (a) is encountered, the control of program jumps to **label_1:** (b) and executes the code below it.

```
num = 10
if (num % 2 == 0)
// jump to even
goto even;
else
// jump to odd
goto odd;
even:
cout « num « ” is even”;
// return if even
return;
odd:
cout « num « ” is odd”;
```

Reason to Avoid goto Statement

- can write any C++ program without the use of goto statement
- makes the logic of the program complex and tangled
- a harmful construct and a bad programming practice

Exercise 2.5

You are given an integer N, print all the odd values, for all i, where 0 <= i < N. Print each values on a seperate line. Note: Use continue statement

```
#include<iostream>

using namespace std;

int main()  {
    int N;
    cin>>N;
    // YOUR CODE GOES HERE
    for(int i = 0; i <= N; i++) {
        if (i % 2 == 0) {
            continue;
        }
        cout << i << endl;
    }

    return 0;
}
```

Function Definition Basic

The general form of a function is:

```
<return_type> <function_name>([arg1_type arg1_name, ...]) { code }

int addTwoInts(int arg1, int arg2) //function takes two integers arg1,arg2
{
   int sum = arg1 + arg2;    // adding both integers together to compute sum
   return sum;    // retuns the sum of type int which is same as function type
}

int main() {
  int answer = addTwoInts(2,4);   //calling our addTwoInts function in main()
  cout << "Answer is: " << answer << endl;
  return 0;
}
```

Function Declaration Basic

Programs written in C++ are executed in logical order **from the top down**. Hence symbols, or named constructs like functions need to be declared & has already been executed before using in a statement; the compiler must be aware of the name used to describe a function before that function can be called in the code.

For main() to call functions that are defined after the main function itself in the code, we use **forward declarations**, which tells the compiler that the function 1. exists, and 2. its arguments(names are optional, but keeping the arguments needed in the declaration is useful for documentation purposes

but the definition of the function will be elsewhere.

The declaration begins much the same as the definition.

```
int fctn2(int num1, int num2) //Recap Function Definition

// Function Declaration:
int fctn2(int, int);
```

Calling Function:

Let's take a look at how functions (that have already been made) are called in the main function.

```
// Case 1: Calling a declaired function
void fctn1();    // declaring a void function
int fctn2(int, int);   //declaring int type function taking two int arguments

int main()
{
    int sum;

    fctn1();        //calling the void function
    sum = fctn2(2,3);    // calling the function fctn and saving the value returned in variable 'sum'
    cout << "The value of sum is: " << sum << endl;
    return 0;
}


// Case 2: Calling a defineded function
void fctn1()      // writing the function definition here
{
    cout << "This is function 1!" << endl;  // function only couts a string
}

int fctn2(int num1, int num2)    // writing the function definition
{
    //the value 2 has been passed as num1
    //the value 3 has been passed as num2

    return num1 + num2;      // returning the sum of num1 and num2
}
```

Function Parameters

Data is passed between functions as parameters through the call of the function. where the order of the Parameters list is determined by the function definition.

```
int fctn(int arg1, int arg2);

int main()
{
   int answer;
   int num1 = 10;
   answer = fctn(num1, 12);    // num1 and 12 are arguments passed to fctn
   cout << "Answer is: " << answer << endl;
}

int fctn(int arg1, int arg2)    // function definition
```

```
{
  cout << "num1 is assigned to arg1, value of arg1 is: "<<arg1<<endl;
  cout << "12 is assigned to arg2, value of arg2 is: "<<arg2<<endl;
  return arg1*arg2;      // mutliplying arg1 and arg2 and returning the answer
}
```

Exercise 3.1

Create a function named "compute" which takes two integer arguments A and B and returns an integer denoting A2 + B2.

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

// Your Code goes here

int compute(int A, int B)

{
     return pow(A, 2) + pow(B, 2);
}
```

Passing parameters

Parameter values can be passed to a function through two methods. - pass by value - pass by reference #### M1: Pass by value Value of the variable is passed, not the variable itself. This would be like a copy of the variable and the original is not altered in any way.

M2: Pass by reference The address of the variable is given instead, and hence allowing function call direct access to the information. Placing a **&** after the data type in the function definition (before the variable name) allows direct access (this must also be present in any forward declaration).

```
void fctn(int& arg1, int arg2){ //passing argument 1 by reference using &
    arg1 = arg2;    //we equate arg1 to arg2
    //there is no return as void function
}

int main() {
    int num1 = 4; //intializing and decalring num1
    cout << "num1 before passing to fctn is: "<<num1<<endl;
    fctn(num1,23); //passing num1 and 23 as arguments to fctn function
    cout << "Value of num1 (arg1 in function) is: "<<num1<<endl;
    return 0;
}
```

As arg1 is passed by reference, whatever changes we make to the arg1 value in fctn function will automatically be made to the argument num1 passed in main function.

Scope

- the level of access an object has.
- A function can access only global variables, and those that are passed to it through arguments.

Exercise 3.2

Create a function named "compute" which takes three arguments A, B, C passed by reference and has a return type void. Update the value of each integer to its respective cube i.e A = A3, B = B3, C = C3.

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

void compute(int &A, int &B, int &C)

{
     A = pow(A, 3);
     B = pow(B, 3);
     C = pow(C, 3);
}
/*
int main()  {
    int A, B, C;
    cin>>A>>B>>C;
    compute(A, B, C);
    cout<<A<<endl;
    cout<<B<<endl;
    cout<<C<<endl;
    return 0;
}
*/
```

Templates

- a mechanism by which C++ implements the generic concept
- were made to fulfill the need to design a generic code, that works the same way in different situations.

Before going through function templates and class templates look at two non-generic (type-sensitive) functions for adding as example:

```
#include <iostream>
using namespace std;
int add ( int x, int y ) //add two ints{
  return (x + y);
}
double add ( double x, double y )  //add two doubles{
    return (x + y);
}
int main(){
  int temp1;
  double temp2;
  temp1 = add(4,5);
  temp2 = add(4.5,5.5);
  cout << "Value of temp1 is: "<< temp1<<endl;
  cout << "Value of temp2 is: "<<temp2<<endl;
}
```

Two functions that do exactly the same thing, but **cannot be defined as a single function because they use different data types.**

1. Function Templates

Design a generic code, that works the same way in different situations for above scenario 1. declaration

```
template<class Type>
// or
template<typename Type>
// Note: class and typename have exactly the same meaning in this case
```

2. Type is our generic data type's name, and when the template is to be used, it would be the same as if Type was a typedef for your datatype,

```
#include <iostream>
using namespace std;
template<class Type>
Type add ( Type x, Type y )
{
    return (x * y);
}

int main() {
    int result = add<int> ( 2, 5 ); //calling template type function
    cout << "Result when integer values are passed is: " << result <<endl;
    double result2 = add<double>(2.5,5.5);
    cout << "Result when double values are passed is: "<<result2;
    return 0;
}
```

2. Class Templates

As another powerful feature of C++, you can also make template classes, which are classes that can have members of the generic type.

```
template<class T>
class Score
{
private:
        T scorenumber;
public:
       Score(T args){scorenumber = args;}
};

// This is how it's object will be declared:
Score<int> myscore(40);
score<double> myscore(23.9);
```

This class we defined keeps score in the variable **scorenumber which could be int, float or double.**

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

template <class T>
class myvalues {
    T myval1, myval2; //two values of type T
  public:
    myvalues (T arg1, T arg2){myval1=arg1; myval2=arg2;} //constructor
    T max (); //max function
```

```
};

template <class T>
T myvalues<T>::max()  //definition of a function with type T
{
  if(myval1 > myval2){
    return myval1;
  }else{
    return myval2;
  }
}

int main () {
  myvalues <int> obj(20, 50); //try changing the value and value types to results for different types
  cout << "Max value is: " << obj.max();
  return 0;
}
```

In the example above, this time we declared a function of type T in our class. The definition of the function was outside the class so we had to add the prefix for template class beforehand. ## Exercise 3.3 define a Template Class type function index_of_largest that will generalize the function such that it finds maximum value index for both int and double type array input values.

The function code for finding the index of the largest value in an array:

Takes two values:

Array of values Size of array Note: All values in the array will be distinct

Sample Input

int array[] = {4, 2, 10, 13, 2} double array[] = {2.4, 5.2, 3.1, 1.5}

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

// Your code goes here
template <class T2>
int index_of_largest(T2 array_numeric[], int n_A)
{
    T2 max = array_numeric[0];
    int ix_max = 0;
    for (int ix = 1; ix < n_A; ix++){
        if (array_numeric[ix] > max)
        {
            max = array_numeric[ix];
            ix_max = ix;
        }
    }
    return ix_max;
}
```

Stringstream

- a stream class to operate on strings.
- implements input/output operations on memory (string) based streams.
- helpful in different type of parsing.

The following operators/functions are commonly used here - Operator » Extracts formatted data. - « Inserts formatted data. - Method str() Gets the contents of underlying string device object. - str(string) Sets the contents of underlying string device object.

Its header file is sstream.

Stringstream common use case:

- parse comma-separated integers from a string (e.g., “23,4,56”).

```
stringstream ss("33,7,65");
char ch;
int a, b, c;
ss >> a >> ch >> b >> ch >> c; // a = 33, b = 7, c = 65
```

Here ch is **a storage area for the discarded commas.** If the » operator returns a value, that is a true value for a conditional, else failure to return a value is false.

Getline Function

- a standard library function that is used to read a string or a line from an input stream.
- a part of the header. // (or sstream?)
- extracts characters from the input stream and appends it to the string object **until the delimiting character is encountered.**
- Previously stored value in the string object str will be replaced by the input string if existed.

```
istream& getline(istream& is, string& str, char delim);
```

Parameters:

is: It is an object of istream class, and tells the function about the stream from where to read the input from. str: It is a string object, the input is stored in this object after being read from the stream. delim: It is the delimitation character which tells the function to stop reading further input after reaching this character. Return Value: This function returns the same input stream as is which is accepted as parameter. ## Example 3.4 Given a string A of comma delimited integers, print each integer in a separate line. Sample Input: A = 33,7,65 Sample Output: 33 7 65

```
/*
#include<iostream>
#include<sstream>
using namespace std;
*/

int main()  {
    string A;
    cin>>A;
    // YOUR CODE GOES HERE
    stringstream  stst(A);
    string number;
    while(getline(stst, number, ',')){
        cout << number << endl;
    }

    return 0;
}
```

Strings

Some of String's widely used features are the following:

```
string a = "abc"; // Declaration:
int len = a.size(); // Size:
// Concatenate two strings:
string a = "abc";
string b = "def";
string c = a + b; // c = "abcdef".
// Accessing ith element:
string s = "abc";
char c0 = s[0]; // c0 = 'a'
char c1 = s[1]; // c1 = 'b'
char c2 = s[2]; // c2 = 'c'

s[0] = 'z'; // s = "zbc"

## Example 3.5

You are given two strings, A and B, separated by a new line. Each string will consist of lower case Lati
Output three lines:
First line print two space-separated integers, representing the length of A and B respectively.
Second line print the string produced by concatenating A and B (A+B).
Third line print two strings separated by a space, A' and B'. A' and B' are the same as A and B, respec
Sample input:
abcd
ef
Sample Output:
4 2
abcdef
ebcd af
```

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

int main()  {
    string A, B;
    cin>>A>>B;
    // YOUR CODE GOES HERE
    int n_A = A.size();
    int n_B = B.size();
    cout << n_A << " " << n_B << endl;
    cout << A + B << endl;
    swap(A[0], B[0]);
    cout << A << " " << B << endl;
    return 0;
}
/*
int main()  {
    string A, B; cin>>A>>B; string s=A+B; char ch=A[0]; A[0]=B[0]; B[0]=ch;
    cout<<A.length()<<" "<<B.length()<<"\n";
```

```
    cout<<s<<"\n";
    cout<<A<<" "<<B;
    return 0;
}
*/
```

Pointers & Array

A pointer is - a special kind of variable that stores the address in memory of another variable - can be used to manipulate that variable.

In general a variable exist in the computer's memory, and pointers can store the location of the variable.

Pointers associate two pieces of information: - The **memory address**, which is the "value" of the pointer itself. - The **data type** of the variable pointed to, which is the kind of variable located at that address.

1. Declaring pointers int *ptr; void* vp;

In the example above ptr: a pointer to an integer. vp: a void pointer, which does not require a specific data type.

2. Assigning Addresses to Pointers int* ptr, var; var = 5; ptr = &var; // assign address of var to ptr pointer Here, 5 is assigned to the variable var. And, the address of var is assigned to the ptr pointer with the code ptr = &var.

3. Dereferencing Pointers Pointers can be dereferenced to access the value of the variable at the pointer's address.

void f(int* p){ int n = *p; // The code* "p" takes the value of the data at location stored in p }

Note: Unlike references, pointers are not guaranteed to be initialized; should only be used when they are **known to point to an existing object**.

4. Malloc() and Free() The malloc(): **allocates a block of uninitialized memory, and returns a void pointer to the first byte of the allocated memory block if the allocation succeeds**. If the size is zero, the value returned depends on the implementation of the library. It may or may not be a null pointer. void* malloc(size_t size);

The free(): deallocates a block of memory, does not change the value of the pointer, that is it still points to the same memory location.

void free(void *ptr); Example:

```
int main(){
    int *ptr;
    ptr = (int*) malloc(5*sizeof(int));

    if(!ptr){
        cout << "Memory Allocation Failed";
        return 0;
    }
    // Initialize values
    for (int i=0; i<5; i++){
        ptr[i] = i+i;
    }

    // Initialized values
```

```
    for (int i=0; i<5; i++){
        // ptr[i] and *(ptr+i) can be used interchangeably
        cout << *(ptr+i) << " ";
    }
    cout<<endl;
    // output: 0 2 4 6 8

    free(ptr);
    // If we again prints the value it will print some garbage value

    for (int i=0; i<5; i++){
        cout << *(ptr+i) << " ";
    }
    // This will not give any error but prints some garbage value
    return 0;
}
```

Common mistakes when working with pointers

Suppose we want 88a pointer varPoint** to point to the **address of var**. Then,

int var, *varPoint;

```
// Wrong! varPoint is an address but var is not
varPoint = var;

// Wrong! &var is an address, *varPoint is the value stored in &var
*varPoint = &var;

// Correct! varPoint is an address and so is &var
varPoint = &var;

// Correct! both *varPoint and var are values
*varPoint = var;
```

Exercise 4.1

You are given a function solve which has the following parameters:

int *A: an integer int* B: an integer The function is declared with a void return type, so there is no value to return. Modify the values in memory so that A contains their sum and B contains their absolute difference.

```
/*
#include<iostream>
using namespace std;
*/
void solve(int *A, int *B){
    int temp = abs(*A - *B);
    *A = *A + *B;
    *B = temp;
}

/*
int main()  {
    int A, B;
```

```
    int *pA = &A, *pB = &B;
    cin>>A>>B;
    solve(pA, pB);
    cout<<A<<endl;
    cout<<B<<endl;
    return 0;
}
*/
```

Array

- a collection of **similar data types** under the same name.

int arr[5]; //dataType arrayName[arraySize]; //int is the datatype, arr is the name, 5 is the size of array

The above statement declares a static array of 5 elements, which can be accessed individually.

Note: arr behaves like a pointer but its value cannot be changed as it references a specific region in memory.

calculate size of the arr2 using the inbuilt sizeof function

```
cout << "The size of arr2 is: "<<sizeof(arr2)/sizeof(arr2[0])<<endl;
```

Accessing Array Values

Arrays have 0 as the first index not 1.

Note: If you try to access array elements outside of its bound, let's say arr[20], the compiler may not show any error. However, this may cause unexpected output (undefined behavior).

int arr[15] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}; cout « "The value of arr[20] is:"« arr[20] «endl; // We will get a garbage value at arr[20] which is an index that is out of bounds.

Iterate over array

int main() { int arr[5] = {}; // no values stored in array by default int num = 1; for (int i=0; i<5; i++){ arr[i] = num; //setting index i of array arr equal to num num++; //incrementing num cout « "The value of arr["«i«"] is equal to:"« arr[i]«endl; } return 0; }

Exercise 4.2

Complete the given function named 'reverseArray' containing an integer array 'arr' and length of arr N. Reverse the array 'arr'.

```
/*
#include<iostream>
using namespace std;
*/

void reverseArray(int arr[], int N){
    // Your code goes here
    //int int_arr = sizeof(arr);
    //int *temp;
    //temp = (int*) malloc(int_arr * sizeof(int));
    int i, j, temp;
    j = N - 1;
    for ( i = 0; i < j; i++, j--){
        temp = arr[i];
```

```
        arr[i] = arr[j];
        arr[j] = temp;
    }

}
```

Exercise 4.3

You are given an integer N, create a 2D array named 'grid' of size N x N.

The diagonal of the grid should be filled with 0. The lower side should be filled with -1s. The upper side should be filled with 1s.

```
/*
#include<iostream>
using namespace std;
*/

int main() {
    int N;
    cin>>N;
    // YOUR CODE GOES HERE
    int grid[N][N];
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
            if(i == j) grid[i][j]=0;
            if(i>j) grid[i][j]= -1;
            if(i<j) grid[i][j]= 1;
        }
    }
    // Don't change the code below
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            cout<<grid[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```