typedef keyword in C++ is used for aliasing existing data types, user-defined data types, and pointers to a more meaningful name. Typedefs allow you to give descriptive names to standard data types, which can also help you self-document your code. Mostly typedefs are used for aliasing, only if the predefined name is too long or complex to write again and again. The unnecessary use of typedef is generally not a good practice.

Syntax:

typedef <current\_name> <new\_name>

Applications of typedef in C++

typedef in C++ can be used for aliasing predefined data types with long names.

It can be used with STL data structures like Vectors, Strings, Maps, etc.

typedef can be used with arrays as well.

We can use typedef with normal pointers as well as function pointers.

==================================================================

Example:

typedef std::vector<int> vInt;

using namespace std;

int main()

{

// Now we can make more vectors by using vInt

typedef std::vector<int> vInt;

// vec1 is a vectorof type int

vInt v;

v.push\_back(190);

v.push\_back(180);

v.push\_back(10);

v.push\_back(10);

v.push\_back(27);

for (auto X : v) {

cout << X << " ";

}

return 0;

}

=============================================================

Usage with data pointers:

Below is the syntax, example, and source code for using typedef with data pointers

Syntax:

typedef <data\_type>\* <alias\_name>

Example:

typedef int\* iPtr;

iPtr pointer1, pointer2;

=====================================================

#include <iostream>

using namespace std;

int main()

{

int a = 10;

int b = 20;

// iPtr can now be used to create new pointers of type

// int

typedef int\* iPtr;

iPtr pointer\_to\_a = &a;

iPtr pointer\_to\_b = &b;

cout << "a is: " << \*pointer\_to\_a << "\n";

cout << "b is: " << \*pointer\_to\_b << "\n";

return 0;

}

==============================================================================

Exception Handling

==>An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).

==>In C++, exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. The process of handling these exceptions is called exception handling.

Exception handling is a manner to handle the runtime error, we carry out exception handling, so, the normal flow of the program may be maintained even after runtime errors.

====================================================================================

Why do we need Exception Handling in C++?

The following are the main advantages of exception handling over traditional error handling:

1)Separation of Error Handling Code from Normal Code: There are always if-else conditions to handle errors in traditional error handling codes. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

2)Grouping of Error Types: In C++, both basic types and objects can be thrown as exceptions.

3)Functions/Methods can handle only the exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

4)The Exception Handling mechanism offers a way to transfer control from one part of a program to another, This makes it clean to separate the mistake handling code from the code written to address the real functionality of the program.thats why detecting of errors becomes very easy

5)Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

====================================================================================

==========================================================================

Syntax of try-catch in C++

try {

// Code that might throw an exception

throw SomeExceptionType("Error message");

}

catch( ExceptionName e1 ) {

// catch block catches the exception that is thrown from try block

}

1. try in C++

The try keyword represents a block of code that may throw an exception placed inside the try block. It’s followed by one or more catch blocks. If an exception occurs, try block throws that exception.

2. catch in C++

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.

3. throw in C++

An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

Note: Multiple catch statements can be used to catch different type of exceptions thrown by try block.

==========================================================================================

// C++ program to show what happens when division by zero is done and exceptions are not handled

#include <iostream>

using namespace std;

int main()

{

int numerator = 5;

int denominator = 0;

int ans = numerator/denominator;

// The result will not be displayed

cout << ans << endl;

return 0;

}

===============================================================================

// C++ program to handle divide by zero exception

#include <iostream>

using namespace std;

int main()

{

int numerator = 5;

int denominator = 0;

int result;

try

{

if(denominator == 0)

{

throw denominator;

}

result = numerator / denominator;

}

catch(int ex)

{

cout << "Exception: Divide by zero not allowed :" <<

ex << endl;

}

cout << "end";

return 0;

}

================================================================

// C++ program to input an age integer if the age is less than 18 then return NO, but if the age is greater than or equal to 18 then return Yes

#include <iostream>

using namespace std;

int main()

{

try

{

int age = 10;

if (age >= 18)

{

cout << "YES, you are old enough.";

}

else

{

throw (age);

}

}

catch (int num)

{

cout << "No, You must be at least 18 years old" <<

endl;

cout << "Age is: " << num;

}

return 0;

}

Output

No, You must be at least 18 years old

Age is: 10

C++ Standard Exceptions

==>C++ defines a set of standard exceptions defined in <exception> which can be used in the programs. These exceptions are arranged in the parent-child class hierarchy.

==>==>In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Below is the table listing the standard exceptions with description:

Exception Description

1. std::exception An exception and parent class of all the standard C++ exceptions.

2. std::bad\_alloc This exception can be thrown by new.

3. std::bad\_cast This exception can be thrown by dynamic\_cast.

4. std::bad\_typeid This can be thrown by type id.

==========================================================================================

==============================================================

Rethrowing an Exception:

In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using “throw; “.

Explain the concept of Rethrowing exceptions?

A rethrowing exception is a term used when we throw the exception again from one catch block to another. The exception is thrown towards another outside catch block.

#include <iostream>

using namespace std;

void help()

{

try {

throw 10;

}

catch (...) {

cout << "First throw called\n";

throw;

}

}

int main()

{

try {

help();

}

catch (...) {

cout << "Rethrowing throw called\n";

}

return 0;

}

Output

First throw called

Rethrowing throw called

=====================================================

Example

The following program shows try/catch blocks nesting.

CPP

// C++ program to demonstrate try/catch blocks can be nested

// in C++

#include <iostream>

using namespace std;

int main()

{

// nesting of try/catch

try {

try {

throw 20;

}

catch (int n) {

cout << "Handle Partially ";

throw; // Re-throwing an exception

}

}

catch (int n) {

cout << "Handle remaining ";

}

return 0;

}

Output

Handle Partially Handle remaining

A function can also re-throw a function using the same “throw; ” syntax. A function can handle a part and ask the caller to handle the remaining.

=================================================================

// C++ program to demonstate the use of try,catch and throw

// in exception handling.

#include <iostream>

using namespace std;

int main()

{

int x = -1;

// Some code

cout << "Before try \n";

// try block

try {

cout << "Inside try \n";

if (x < 0) {

// throwing an exception

throw x;

cout << "After throw (Never executed) \n";

}

}

// catch block

catch (int x) {

cout << "Exception Caught \n";

}

cout << "After catch (Will be executed) \n";

return 0;

}

======================================================================

Properties of Exception Handling in C++

Property 1

There is a special catch block called the ‘catch-all’ block, written as catch(…), that can be used to catch all types of exceptions.

Example

In the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(…) block will be executed.

CPP

// C++ program to demonstate the use of catch all

// in exception handling.

#include <iostream>

using namespace std;

int main()

{

// try block

try {

// throw

throw 10;

}

// catch block

catch (char\* excp) {

cout << "Caught " << excp;

}

// catch all

catch (...) {

cout << "Default Exception\n";

}

return 0;

}

Output

Default Exception

====================================================================================

#include <iostream>

using namespace std;

int main()

{

try {

throw 10;

}

catch (...) {

cout << "default exception \n";

}

catch (int param) {

cout << "integer exception \n";

}

return 0;

}

Output:

An error will be raised

Explanation: The default catch block should be the last catch block or else other catch blocks will never be reached.

======================================================================================

==========================================================

C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting exception class functionality.

Let's see the simple example of user-defined exception in which std::exception class is used to define the exception.

#include <iostream>

using namespace std;

class MyException : public exception{

public:

char \* what() const throw()

{

return "Attempted to divide by zero!\n";

}

};

int main()

{

try

{

int x, y;

cout << "Enter the two numbers : \n";

cin >> x >> y;

if (y == 0)

{

MyException z;

throw z;

}

else

{

cout << "x / y = " << x/y << endl;

}

}

catch(exception& e)

{

cout << e.what();

}

}

Note: In above example what() is a public method provided by the exception class. It is used to return the cause of an exception.

===================================================================================================

Enumeration in c++

What Is C++ Enum?

==>Enumeration (or enum) is a user defined data type in C/c++/java. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

==>Enum, which is also known as enumeration, is a user-defined data type that enables you to create a new data type that has a fixed range of possible values, and the variable can select one value from the set of values. For example, suppose you are the owner of an ice cream shop, and you sell a limited range of ice cream flavors. So you want the customers to select only from that collection of ice creams in your shop. This becomes an enumeration with ice cream as the name of enumeration and different flavors of ice creams as its elements.

==>Enumeration (Enumerated type) is a user-defined data type that can be assigned some limited values. These values are defined by the programmer at the time of declaring the enumerated type.

Syntax:

enum enumerated-type-name

{

value1, value2, value3…..valueN

};

enum Iceccream

{

vanilla,stawberry,mango

};

Each of these elements has values starting from 0, like vanilla is 0, stawberry is 1, etc. The default values mentioned can be altered during the declaration of the enum in C++.

For Example: If a gender variable is created with the value male or female. If any other value is assigned other than male or female then it is not appropriate. In this situation, one can declare the enumerated type in which only male and female values are assigned.

============================================================================

Declaration

To declare an enum variable, write the name of the enumeration along with the enum variable.

enum Iceccream

{

vanilla,stawberry,mango

};

int main()

{

Icecream i;

}

Here i is the enum variable.

===========================================================

Example:

//first\_enum is the enumerated-type-name

enum first\_enum{value1=1, value2=10, value3};

In this case,

first\_enum e;

e=value3;

cout<<e;

Output:

11

=========================================================

// C++ Program to Demonstrate the Functioning of Enumerators

// with an example of Gender

#include <bits/stdc++.h>

using namespace std;

int main()

{

// Defining enum Gender

enum Gender { Male, Female };

// Creating Gender type variable

Gender gender = Male;

switch (gender) {

case Male:

cout << "Gender is Male";

break;

case Female:

cout << "Gender is Female";

break;

default:

cout << "Value can be Male or Female";

}

return 0;

}

=====================================================================

// C++ Program to Demonstrate the Functioning of Enumerators

// with an Example of Year

#include <bits/stdc++.h>

using namespace std;

// Defining enum Year

enum year {

Jan=1,

Feb,

Mar,

Apr,

May,

Jun,

Jul,

Aug,

Sep,

Oct,

Nov,

Dec

};

// Driver Code

int main()

{

int i;

// Traversing the year enum

for (i = Jan; i <= Dec; i++)

cout << i << " ";

return 0;

}

Output:

0 1 2 3 4 5 6 7 8 9 10 11

==============================================================

Points to remember for C++ Enum

enum can be easily used in switch

enum can be traversed

The C++ enum constants are final implicitly.

If we assign a float value to a character value, then the compiler generates an error. In the same way, if we try to assign any other value to the enumerated data types, the compiler generates an error.

=======================================

Why enums are used in C++ programming?

Enums are used to give names to constants, which makes the code easier to read and maintain.

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

1)Enums or enumerations are generally used when you expect the variable to select one value from the possible set of values

2)Enums are strongly typed, which means that the compiler will catch any errors at compile time rather than at runtime. ...

3)readable: Enums provide a more readable way to define a fixed set of values. ...

4)extensible: Enums can be extended to add new values.

5)Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes

=====================================================================

===============================================================================================================

RTTI

==>In C++, RTTI (Run-time type information) is a mechanism that exposes information about an object’s data type at runtime and is available only for the classes which have at least one virtual function. It allows the type of an object to be determined during program execution.

==>There are three main C++ language elements to detect run-time type information:

The dynamic\_cast operator.

Used for conversion of polymorphic types.

The typeid operator.

Used for identifying the exact type of an object at runtime.

The type\_info class.

Used to hold the type information returned by the typeid operator.

=====================================================================================================

1)Runtime Casting

==>In C++, if a class is derived from a base class containing one or more virtual functions, a pointer to that base class type can be used to call virtual functions in the derived class object. A class containing virtual functions is sometimes called a "polymorphic class."

==>Using run-time type information, it's possible to check whether a pointer actually points to a complete object and can be safely cast to point to another object in its hierarchy. The dynamic\_cast operator performs a run-time check to ensure that the operation is safe. It's better to design your class hierarchy so that you can use virtual functions to avoid the need for downcasting. However, if you must downcast, use dynamic\_cast to ensure that the operation is safe.

==>The runtime cast, which checks that the cast is valid, is the simplest approach to ascertain the runtime type of an object using a pointer or reference. This is especially beneficial when we need to cast a pointer from a base class to a derived type. When dealing with the inheritance hierarchy of classes, the casting of an object is usually required. There are two types of casting:

Upcasting: When a pointer or a reference of a derived class object is treated as a base class pointer.

Downcasting: When a base class pointer or reference is converted to a derived class pointer.

Using ‘dynamic\_cast‘: In an inheritance hierarchy, it is used for downcasting a base class pointer to a child class. On successful casting, it returns a pointer of the converted type and, however, it fails if we try to cast an invalid type such as an object pointer that is not of the type of the desired subclass.

----------------------------------------------------------------------------------------------

For example, dynamic\_cast uses RTTI and the following program fails with the error “cannot dynamic\_cast `b’ (of type `class B\*’) to type `class D\*’ (source type is not polymorphic) ” because there is no virtual function in the base class B.

#include <iostream>

using namespace std;

// initialization of base class

class B {};

// initialization of derived class

class D : public B {};

// Driver Code

int main()

{

B\* b = new D; // Base class pointer

D\* d = dynamic\_cast<D\*>(b); // Derived class pointer

if (d != NULL)

cout << "works";

else

cout << "cannot cast B\* to D\*";

getchar(); // to get the next character

return 0;

}

====================================================================

// C++ program to demonstrate

// Run Time Type Identification successfully

// With virtual function

#include <iostream>

using namespace std;

// Initialization of base class

class B {

virtual void fun() {}

};

// Initialization of Derived class

class D : public B {

};

// Driver Code

int main()

{

B\* b = new D; // Base class pointer

D\* d = dynamic\_cast<D\*>(b); // Derived class pointer

if (d != NULL)

cout << "works";

else

cout << "cannot cast B\* to D\*";

getchar();

return 0;

}

========================================================================================

2) typeid is an operator in C++.

It is used where the dynamic type or runtime type information of an object is needed.

It is included in the <typeinfo> library. Hence inorder to use typeid, this library should be included in the program.

The typeid expression is an lvalue expression.

Syntax:

typeid(type);

OR

typeid(expression);

Parameters: typeid operator accepts a parameter, based on the syntax used in the program:

type: This parameter is passed when the runtime type information of a variable or an object is needed. In this, there is no evaluation that needs to be done inside type and simply the type information is to be known.

expression: This parameter is passed when the runtime type information of an expression is needed. In this, the expression is first evaluated. Then the type information of the final result is then provided.

Return value: This operator provides the runtime type information of the specified parameter and hence that type information is returned, as a reference to an object of class type\_info.

ex:

#include <iostream>

#include <typeinfo>

using namespace std;

int main()

{

int i, j;

char c;

// Get the type info using typeid operator

const type\_info& ti1 = typeid(i);

const type\_info& ti2 = typeid(j);

const type\_info& ti3 = typeid(c);

// Check if both types are same

if (ti1 == ti2)

cout << "i and j are of"

<< " similar type" << endl;

else

cout << "i and j are of"

<< " different type" << endl;

// Check if both types are same

if (ti2 == ti3)

cout << "j and c are of"

<< " similar type" << endl;

else

cout << "j and c are of"

<< " different type" << endl;

return 0;

}

ex 2:

#include <iostream>

#include <typeinfo>

class Base {

public:

virtual void vvfunc() {}

};

class Derived : public Base {};

using namespace std;

int main() {

Derived\* pd = new Derived;//pd=100

Base\* pb = pd;

cout << typeid( pb ).name() << endl; //prints "class Base \*"

cout << typeid( \*pb ).name() << endl; //prints "class Derived"

cout << typeid( pd ).name() << endl; //prints "class Derived \*"

cout << typeid( \*pd ).name() << endl; //prints "class Derived"

delete pd;

}

=========================================================================

If the expression is dereferencing a pointer, and that pointer's value is zero, typeid throws a bad\_typeid exception

#include <iostream>

#include <typeinfo>

class Base {

public:

virtual void vvfunc() {}

};

class Derived : public Base {};

using namespace std;

int main() {

Derived\* pd = new Derived;

Base\* pb = pb;

cout << typeid( pb ).name() << endl; //prints "class Base \*"

//cout << typeid( \*pb ).name() << endl; //prints "class Derived"

//cout << typeid( pd ).name() << endl; //prints "class Derived \*"

//cout << typeid( \*pd ).name() << endl; //prints "class Derived"

delete pd;

}

=======================================================================================

The type\_info class describes type information generated within the program by the compiler. Objects of this class effectively store a pointer to a name for the type.

=======================================================================================================

Casting Operators in C++

Casting operators are used for type casting in C++. They are used to convert one data type to another. C++ supports four types of casts:

static\_cast

dynamic\_cast

const\_cast

reinterpret\_cast

1. static\_cast

The static\_cast operator is the most commonly used casting operator in C++. It performs compile-time type conversion and is mainly used for explicit conversions that are considered safe by the compiler.

Syntax of static\_cast

static\_cast <new\_type> (expression);

where,

expression: Data to be converted.

new\_type: Desired type of expression

The static\_cast can be used to convert between related types, such as numeric types or pointers in the same inheritance hierarchy.

Example of static\_cast

C++

// C++ program to illustrate the static\_cast

#include <iostream>

#include <typeinfo>

using namespace std;

int main()

{

int num = 10;

// converting int to double

double numDouble = static\_cast<double>(num);

// printing data type

cout << typeid(num).name() << endl;

// typecasting

cout << typeid(static\_cast<double>(num)).name() << endl;

// printing double type t

cout << typeid(numDouble).name() << endl;

return 0;

}

Output

i

d

d

In this example, we have included the “typeinfo” library so that we can use typeid() function to check the data type. We have defined an integer variable ‘num’ and converted it into a double using static\_cast. After that, we print the data types of variables and pass static\_cast<double>(num) in typeid() function to check its data type. we can see the output “i, d, d” is printed where ‘i’ denotes integer and ‘d’ denotes double.

=======================================================================================

//prevents dangerous casts -more restrictive.

==>integer pointer will think,I have 4 bytes in memory and I can modified this 4 bytes in the memory.

assigned 3 as 00 00 00 11. it will lead to runtime error.

beacause it expect that if u r having int pointer ,u can modify 4 bytes.

// C++ Program to demonstrate

// static\_cast char\* to int\*

#include <iostream>

using namespace std;

// Driver code

int main()

{

int a = 10;

char c = 'a';

int\* q = (int\*)&c;//allowed

\*q1=10; // Pass at compile time, may fail at run time

int\* p = static\_cast<int\*>(&c);//compile time error

return 0;

}

Output

error: invalid 'static\_cast' from type 'int\*' to type 'char\*'

Explanation: This means that even if you think you can somehow typecast a particular object pointer into another but it’s illegal, the static\_cast will not allow you to do this.

===================================================================================

2. dynamic\_cast

The dynamic\_cast operator is mainly used to perform downcasting (converting a pointer/reference of a base class to a derived class). It ensures type safety by performing a runtime check to verify the validity of the conversion.

Syntax of dynamic\_cast

dynamic\_cast <new\_type> (expression);

If the conversion is not possible, dynamic\_cast returns a null pointer (for pointer conversions) or throws a bad\_cast exception (for reference conversions).

Example of dynamic\_cast

C++

// C++ program to illustrate the dynamic\_cast

#include <iostream>

using namespace std;

// Base Class

class Animal {

public:

virtual void speak()

{

cout << "Animal speaks." << endl;

}

};

// Derived Class

class Dog : public Animal {

public:

void speak()

{

cout << "Dog barks." << endl;

}

};

// Derived Class

class Cat : public Animal {

public:

void speak()

{

cout << "Cat meows." << endl;

}

};

int main()

{

// base class pointer to derived class object

Animal\* animalPtr = new Dog();

// downcasting

Dog\* dogPtr = dynamic\_cast<Dog\*>(animalPtr);

// checking if the typecasting is successfull

if (dogPtr) {

dogPtr->speak();

}

else {

cout << "Failed to cast to Dog." << endl;

}

// typecasting to other dervied class

Cat\* catPtr = dynamic\_cast<Cat\*>(animalPtr);

if (catPtr) {

catPtr->speak();

}

else {

cout << "Failed to cast to Cat." << endl;

}

delete animalPtr;

return 0;

}

Output

Dog barks.

Failed to cast to Cat.

Explanation: The first line of output is printed because the ‘animalPtr’ of the ‘Animal’ type is successfully cast to the ‘Dog’ type and speak() function of the Dog class is invoked but the casting of the ‘Animal’ type to ‘Cat’ type is failed because ‘animalPtr’ points to a ‘Dog’ object thus, the dynamic cast fails because the typecasting is not safe.

===========================================================================================

3. const\_cast

The const\_cast operator is used to modify the const or volatile qualifier of a variable. It allows programmers to temporarily remove the constancy of an object and make modifications. Caution must be exercised when using const\_cast, as modifying a const object can lead to undefined behavior.

Syntax for const\_cast

const\_cast <new\_type> (expression);

Example of const\_cast

C++

// C++ program to illustrate the const\_cast

#include <iostream>

using namespace std;

int main()

{

const int number = 5;

// Pointer to a const int

const int\* ptr = &number;

// int\* nonConstPtr = ptr; if we use this

// instead of without using const\_cast

// we will get error of invalid conversion

int\* nonConstPtr = const\_cast<int\*>(ptr);

\*nonConstPtr = 10;

cout << "Modified number: " << \*nonConstPtr;

return 0;

}

Output

Modified number: 10

In the above example, we have modified the value of the const type pointer by changing its qualifier from const to non-const and then printing the modified value.

================================================================================

4. reinterpret\_cast

The reinterpret\_cast operator is used to convert the pointer to any other type of pointer. It does not perform any check whether the pointer converted is of the same type or not.

Syntax of reinterpret\_cast

reinterpret\_cast <new\_type> (expression);

Example

C++

// C++ program to illustrate the reinterpret\_cast

#include <iostream>

using namespace std;

int main()

{

int number = 10;

// Store the address of number in numberPointer

int\* numberPointer = &number;

// Reinterpreting the pointer as a char pointer

char\* charPointer

= reinterpret\_cast<char\*>(numberPointer);

// Printing the memory addresses and values

cout << "Integer Address: " << numberPointer << endl;

cout << "Char Address: "

<< reinterpret\_cast<void\*>(charPointer) << endl;

return 0;

}

Output

Integer Address: 0x7fff64789f1c

Char Address: 0x7fff64789f1c

In the above example, we have defined an int variable ‘number’ and then store the address of ‘number’ in ‘numberPointer’ of the int type after that we have converted the ‘numberPointer’ of the int type into char pointer and then store it into ‘charPointer’ variable. To verify that we have printed the address of both numberPointer and charPointer. To print the address stored in ‘charPointer’ reinterpret\_cast<void\*> is used to bypass the type-checking mechanism of C++ and allow the pointer to be printed as a generic memory address without any type-specific interpretation.

====================================================================================================

String class by javatpoint =============>> https://www.programiz.com/cpp-programming/string-class/

#include <iostream>

#include<string.h>

using namespace std;

int main()

{

string name; // variable declaration.

cin>>name;

cout<<name;

return 0;

}

====================================================================================================

C++ getline()

The cin is an object which is used to take input from the user but does not allow to take the input in multiple lines. To accept the multiple lines, we use the getline() function. It is a pre-defined function defined in a <string.h> header file used to accept a line or a string from the input stream until the delimiting character is encountered.

Syntax of getline() function:

There are two ways of representing a function:

The first way of declaring is to pass three parameters.

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

is: It is an object of the istream class that defines from where to read the input stream.

str: It is a string object in which string is stored.

delim: It is the delimiting character.

#include <iostream>

#include<string.h>

using namespace std;

int main()

{

string name; //object declaration.

std::cout << "Enter your name :" << std::endl;

getline(cin,name); // implementing a getline() function

cout<<"\nHello "<<name;

return 0;}

=====================================================================

Manipulator

Manipulator Function

manipulator are used to change the output results.

1)setw-->in c when we used \t we get spaces. but my req is to perform formated output.

here setw means set width,here it will allocate specify number of spaces on the screen and it is going to print text from right to left.

printf() ,cout --->print output from left to right

setw print output from right to left

cout<<"1\n10\n100\n";

output is 1

1 0

1 0 0

result 3 0 0

our requirement is

1

1 0

1 0 0

result 111

print matter from right to left go for setw.

cout<< setw(5)<<1<<endl;//5 means no of spaces

leave 5 space

1

cout<<setw(5)<<10<<endl;

10

cout<<setw(5)<<100

100

//easy to calculate bill

====================================

setfill()-->fill space with some symbol;

cout<<setfill('\*');

cout<<setw(5)<<1<<endl

//\*\*\*\*1

setfill works with setw. when setw is der then only setfill is implemented.

===================================================

cout<<setfill('\*');

cout<<setw(5)<<1<<endl

cout<<setfill('\*');

cout<<setw(5)<<10<<endl

cout<<setfill(' ');//when we dont want \* symbol.setfill continue untill we write setfill(' ')

cout<<setw(5)<<100<<endl

output \*\*\*\*1

\*\*\*10

100

================

(imp)

setprecision()

cout<<22/7.0;

//3.142758

here we give priority 1 or 2 decimal no,truncate the remaining decimal points

for this setprecision () is used to indicate no of decimal places in floating values.

cout<<setprecision(2)<<22/7.0;//no of decimal

//3.14

setprecisiom() will not work for integer data

cout<<setprecision(2)<<124;//not work for integer will print as it is.

==================================================

setbase() -- use to set the base value.

65 is decimal no ,i want to print octal no ,..65 octal number is

cout<<setbase(8)<<65; // 65/8-->8 1 //8/8 --->1 0

//1 0 1 is the octal value for 65

cout<<setbase(16)<<65; //65/16-->4 1 //

//4 1 is hexa decimal format of 65

I want to print 101 in decimal no

cout<<setbase(10)<<0101;

// 65------->octal no is converted into decimal

cout<<setbase(10)<<0x41;//convert hexa into decimal no

//65

imp

==>to convert the number system from 1 format to another use setbase()

=======================================================

iomanip setiosflags() function in C++ with Examples

The setiosflags() method of iomanip library in C++ is used to set the ios library format flags specified as the parameter to this method.

Syntax:

setiosflags (ios\_base::format\_flag)

// C++ program to illustrate the const\_cast

#include <iostream>

using namespace std;

#include<iomanip>

int main()

{

//cout<<setiosflags(ios::showpos)<<100;//+100

//cout<<setiosflags(ios::scientific)<<234.12345<<endl;

//cout<<setiosflags(ios::fixed)<<2.341234e+002;

//cout<<setiosflags(ios::showpoint)<<22/5.1;

}

==========================================================================

C++ File Handling

File handling in C++ is a mechanism to create and perform read/write operations on a file.

We can access various file handling methods in C++ by importing the <fstream> class.

#include <fstream>

<fstream> includes two classes for file handling:

ifstream - to read from a file.

ofstream - to create/open and write to a file.

Opening and Closing a File

In order to work with files, we first need to open them. In C++, we can open a file using the ofstream and ifstream classes.

For example, here's how we can open a file using ofstream:

std::ofstream my\_file("example.txt");

Here,

my\_file - the name of the object of the ofstream class.

example.txt - the name and extension of the file we want to open.

Note: We can also use the open() function to open a file. For example,

std::ofstream my\_file.open("example.txt");

Closing a File

Once we're done working with a file, we need to close it using the close() function.

my\_file.close();

================================================================================

Example 1: Opening and Closing a File

#include <iostream>

#include <fstream>

using namespace std;

int main() {

// opening a text file for writing

ofstream my\_file("example.txt");

// close the file

my\_file.close();

}

//This code will open and close the file example.txt.

===============================================================================

Check the File for Errors

In file handling, it's important to ensure the file was opened without any error before we can perform any further operations on it.There are SOME common ways to check files for errors:

1. By Checking the File Object

ofstream my\_file("example.txt");

// check if the file has been opened properly

if (!my\_file) {

// print error message

cout << "Error opening the file." << endl;

// terminate the main() function

return 1;

}

Notice the condition in the if statement:

if (!my\_file) {...}

This method checks if the file is in an error state by evaluating the file object itself.

If the file has been opened successfully, the condition evaluates to true.

If there's an error, it evaluates to false, and you can handle the error accordingly.

==========================================================================================================

2. Using the is\_open() Function

The is\_open() function returns

true - if the file was opened successfully.

false - if the file failed to open or if it is in a state of error.

For example,

ofstream my\_file("example.txt");

if (!my\_file.is\_open()) {

cout << "Error opening the file." << endl;

return 1;

}

===========================================================================================

Read From a File

Reading from text files is done by opening the file using the ifstream class. For example,

ifstream my\_file("example.txt");

Then, we need to read the file line-by-line. To do this, we need to loop through each line of the file until all the lines are read, i.e., until we reach the end of the file.

We use the eof() function for this purpose, which returns

true - if the file pointer points to the end of the file

false - if the file pointer doesn't point to the end of the file

For example,

// variable to store file content

string line;

// loop until the end of the file

while (!my\_file.eof()) {

// store the current line of the file

// in the "line" variable

getline(my\_file, line);

// print the line variable

cout << line << endl;

}

Here, the while loop will run until the end of the file. In each iteration of the loop,

getline(my\_file, line); reads the current line of the file and stores it in the line variable.

Then, it prints the line variable.

==============================================================================

Example 2: Read From a File

#include <iostream>

#include <fstream>

using namespace std;

int main() {

// open a text file for reading

ifstream my\_file("example.txt");

// check the file for errors

if(!my\_file) {

cout << "Error: Unable to open the file." << endl;

return 1;

}

// store the contents of the file in "line" string

string line;

// loop until the end of the text file

while (!my\_file.eof()) {

// store the current line of the file

// in the "line" variable

getline(my\_file, line);

// print the line variable

cout << line << endl;

}

// close the file

my\_file.close();

return 0;

}

========================================================

WRITING TO A FILE

We use the ofstream class to write to a file. For example,

ofstream my\_file("example.txt");

We can then write to the file by using the insertion operator << with the ofstream object my\_file. For example,

#include <iostream>

#include <fstream>

using namespace std;

int main() {

// open a text file for writing

ofstream my\_file("example.txt");

// check the file for errors

if(!my\_file) {

cout << "Error: Unable to open the file." << endl;

return 1;

}

// write multiple lines to the file

my\_file << "LINENO1" << endl;

my\_file << "LINENO2222222222222222222222222222222222" << endl;

my\_file << "Line 33333333333333333333333333333" << endl;

// close the file

my\_file.close();

return 0;

}

==============================================================================

Append to a Text File

To add/append to the existing content of a file, you need to open the file in append mode.

In C++, you can achieve this by using the ios::app flag when opening the file:

ofstream my\_file("example.txt", ios::app);

Now, let's add some more text to the existing content of example.txt:

#include <iostream>

#include <fstream>

using namespace std;

int main() {

// open a text file for appending

ofstream my\_file("example.txt", ios::app);

// if the file doesn't open successfully, print an error message

if(!my\_file) {

cout << "Failed to open the file for appending." << endl;

return 1;

}

// append multiple lines to the file

my\_file << "Line 4" << endl;

my\_file << "Line 5" << endl;

my\_file << "Line 6" << endl;

// close the file

my\_file.close();

return 0;

}

=========================================================================

File Handling With fstream

Instead of using ifstream to read from a file and ofstream to write to the file, we can simply use the fstream class for all file operations.

The constructor for fstream allows you to specify the file name and the mode for file operations.

Mode AND Description

ios::in

Opens the file to read (default for ifstream).

ios::out

Opens the file to write (default for ofstream).

ios::app

Opens the file and appends new content to it at the end.

if (my\_file) {

my\_file << "This is a test line." << endl;

my\_file.close();

}

else {

cout << "Unable to open file for writing." << endl;

return 1;

}

// 2. read from the same file

string line;

my\_file.open("example.txt", ios::in);

if (my\_file) {

while (!my\_file.eof()) {

getline(my\_file, line);

cout << "Read from file: " << line << endl;

}

my\_file.close();

}

else {

cout << "Unable to open file for reading." << endl;

return 1;

}

// 3. append data to the end of the file

my\_file.open("example.txt", ios::app);

if (my\_file) {

my\_file << "This is another test line, appended to the file." << endl;

my\_file.close();

}

else {

cout << "Unable to open file for appending." << endl;

return 1;

}

return 0;

}

==========================================================================================

c++ 11 features:

1)Uniform Initialization in C++

Uniform initialization is a feature in C++ 11 that allows the usage of a consistent syntax to initialize variables and objects ranging from primitive type to aggregates. In other words, it introduces brace-initialization that uses braces ({}) to enclose initializer values. The syntax is as follows:

Following are some of the examples of the different ways of initializing different types:

// uninitialized built-in type

int i;

// initialized built-in type

int j=10;

// initialized built-in type

int k(10);

// Aggregate initialization

int a[]={1, 2, 3, 4}

// default constructor

X x1;

// Parameterized constructor

X x2(1);

// Parameterized constructor with single argument

X x3=3;

// copy-constructor

X (x4)x3;

If initialized using brace initialization, the above code can be re-written as:

int i{}; // initialized built-in type, equals to int i{0};

int j{10}; // initialized built-in type

int a[]{1, 2, 3, 4} // Aggregate initialization

X x1{}; // default constructor

X x2{1}; // Parameterized constructor;

X x4{x3}; // copy-constructor

======================================================================================

2) auto in C++

==>The auto keyword in C++ specifies that the type of the variable that is being declared will be automatically deducted from its initializer. In the case of functions, if their return type is auto then that will be evaluated by return type expression at runtime. Good use of auto is to avoid long initializations when creating iterators for containers.

==>Note: The variable declared with auto keyword should be initialized at the time of its declaration only or else there will be a compile-time error.

// C++ program to demonstrate working of auto

// and type inference

#include <bits/stdc++.h>

using namespace std;

int main()

{

// auto a; this line will give error

// because 'a' is not initialized at

// the time of declaration

// a=33;

// see here x ,y,ptr are

// initialised at the time of

// declaration hence there is

// no error in them

auto x = 4;

auto y = 3.37;

auto z = 3.37f;

auto c = 'a';

auto ptr = &x;

auto pptr = &ptr; //pointer to a pointer

cout << typeid(x).name() << endl

<< typeid(y).name() << endl

<< typeid(z).name() << endl

<< typeid(c).name() << endl

<< typeid(ptr).name() << endl

<< typeid(pptr).name() << endl;

return 0;

}

Output

i

d

f

c

Pi

PPi

Note: We have used typeid for getting the type of the variables.

Here, typeid is an operator that is used where the dynamic type of an object needs to be known.

typeid(x).name() returns the data type of x, for example, it returns:

‘i’ for integers, ‘d’ for doubles,

‘f’ for float, ‘c’ for char,

Pi’ for the pointer to the integer,

‘Pd’ for the pointer to double,’

Pf’ for the pointer to float,

‘Pc’ for the pointer to char,

‘PPi’ for the pointer to pointer to integer.

===============================================================================

STL:

C++ Standard Template Library (STL)

==>The C++ Standard Template Library (STL) is a set of template classes and functions that provides the implementation of common data structures and algorithms such as lists, stacks, arrays, sorting, searching, etc. It also provides the iterators and functors which makes it easier to work with algorithms and containers.

==>STL was originally designed by Alexander Stepanov and was later accepted as the part of C++ standard in C++ 98. It is a generalized library so we can use it with almost every data type without repeating the implementation code.

Components of STL

The components of STL are the features provided by Standard Template Library (STL) in C++ that can be classified into 4 types:

Containers

Algorithms

Iterators

==>These components are designed to be efficient, flexible, and reusable, making them an integral part of modern C++ programming.

==========================================================================================

Components:

Containers are the data structures used to store objects and data according to the requirement. Each container is implemented as a template class that also contains the methods to perform basic operations on it.

1)Arrays: The STL array is an implementation of a compile time non-resizable array. It contains various method for common array operations.

==>The array is a collection of homogeneous objects and this array container is defined for constant size arrays or (static size). This container wraps around fixed-size arrays.

In order to utilize arrays, we need to include the array header:

// CPP program to demonstrate working of array

#include <algorithm>

#include <array>

#include <iostream>

#include <iterator>

#include <string>

using namespace std;

int main() {

// construction uses aggregate initialization

// double-braces required

array<int, 5> ar1{{3, 4, 5, 1, 2}};

array<int, 5> ar2 = {1, 2, 3, 4, 5};

cout << "Sizes of arrays are" << endl;

cout << ar1.size() << endl;

cout << ar2.size() << endl;

cout << "\nInitial ar1 : ";

for (auto i : ar1)

cout << i << ' ';

// container operations are supported

sort(ar1.begin(), ar1.end());

cout << "\nsorted ar1 : ";

for (auto i : ar1)

cout << i << ' ';

return 0;

}

Vector:

==>Vectors are the same as dynamic arrays with the ability to resize themselves automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.

==>std::vector in C++ is the class template that contains the vector container and its member functions. It is defined inside the <vector> header file.

Commonly Used Member Functions

Some commonly used member functions of std::vector class are written below:

Iterators

begin() – Returns an iterator pointing to the first element in the vector

end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector

rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

cbegin() – Returns a constant iterator pointing to the first element in the vector.

cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.

// C++ program to illustrate the

// iterators in vector

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> g1;

for (int i = 1; i <= 5; i++)

g1.push\_back(i);

cout << "Output of begin and end: ";

for (auto i = g1.begin(); i != g1.end(); ++i)

cout << \*i << " ";

cout << "\nOutput of cbegin and cend: ";

for (auto i = g1.cbegin(); i != g1.cend(); ++i)

cout << \*i << " ";

cout << "\nOutput of rbegin and rend: ";

for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)

cout << \*ir << " ";

}

Output of begin and end: 1 2 3 4 5

Output of cbegin and cend: 1 2 3 4 5

Output of rbegin and rend: 5 4 3 2 1

Deque:

==>Double-ended queues are sequence containers with the feature of expansion and contraction on both ends.

==>Double Ended Queues are basically an implementation of the data structure double-ended queue. A queue data structure allows insertion only at the end and deletion from the front. This is like a queue in real life, wherein people are removed from the front and added at the back. Double-ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.

#include <deque>

#include <iostream>

using namespace std;

int main()

{

deque<int> gquiz;

gquiz.push\_back(10);

gquiz.push\_front(20);

gquiz.push\_back(30);

gquiz.push\_front(15);

cout << "The deque gquiz is : ";

for (auto it = gquiz.begin(); it != gquiz.end(); ++it)

cout << '\t' << \*it;

cout << '\n';

}

15 20 10 30

Sets:

Sets are a type of associative container in which each element has to be unique because the value of the element identifies it. The values are stored in a specific sorted order i.e. either ascending or descending.

The std::set class is the part of C++ Standard Template Library (STL) and it is defined inside the <set> header file.

// C++ Program to Demonstrate

// the basic working of STL

#include <iostream>

#include <set>

int main()

{

std::set<char> a;

a.insert('G');

a.insert('F');

a.insert('G');

for (auto str : a) {

std::cout << str << ' ';

}

std::cout << '\n';

return 0;

}

=============================================================

Maps store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

// C++ program to illustrate the begin and end iterator

#include <iostream>

#include <map>

#include <string>

using namespace std;

int main()

{

// Create a map of strings to integers

map<string, int> mp;

// Insert some values into the map

mp["one"] = 1;

mp["two"] = 2;

mp["three"] = 3;

// Get an iterator pointing to the first element in the

// map

map<string, int>::iterator it = mp.begin();

// Iterate through the map and print the elements

while (it != mp.end()) {

cout << "Key: " << it->first

<< ", Value: " << it->second << endl;

++it;

}

return 0;

}

=====================================================================

Algorithms

STL algorithms offer a wide range of functions to perform common operations on data (mainly containers). These functions implement the most efficient version of the algorithm for tasks such as sorting, searching, modifying and manipulating data in containers, etc. All STL algorithms are defined inside the <algorithm> and <numeric> header file.

Iterators

Iterators are the pointer like objects that are used to point to the memory addresses of STL containers. They are one of the most important components that contributes the most in connecting the STL algorithms with the containers. Iterators are defined inside the <iterator> header file.

===========================================================================================================