/\*

\* C++ Programming Notes

\* Ravi Kumar Reddy K

\* github.com/ravikumark815

\*/

**Preset:**

* Invented by Bjarne Stroustrup in 1979
* Middle Level Language
* Versions: C++ 14, C++11, C++99

**Hello World:**

#include <iostream>

using namespace std;

int imGlobal = 0;

const double PI = 3.141;

int main(int argc, char\*\*argv) {

cout << "Hello World\n";

return 0;

}

* Namespaces
* main: Start executing from here
* Cout allows us to output information to console
* “<<” Stream insertion operator: Takes string on the right to cout stream
* “endl” Issue newline and force write to console
* argc: No of arguments passed to main
* argv: Array of pointers to strings in the arg vector
* int: Return an integer when done executing
* imGlobal: Global variable and accessible everywhere else.
* const double PI: Global variable whose value cannot be changed anywhere else

**Comments:**

/\*

Multi

Line

Comment

\*/

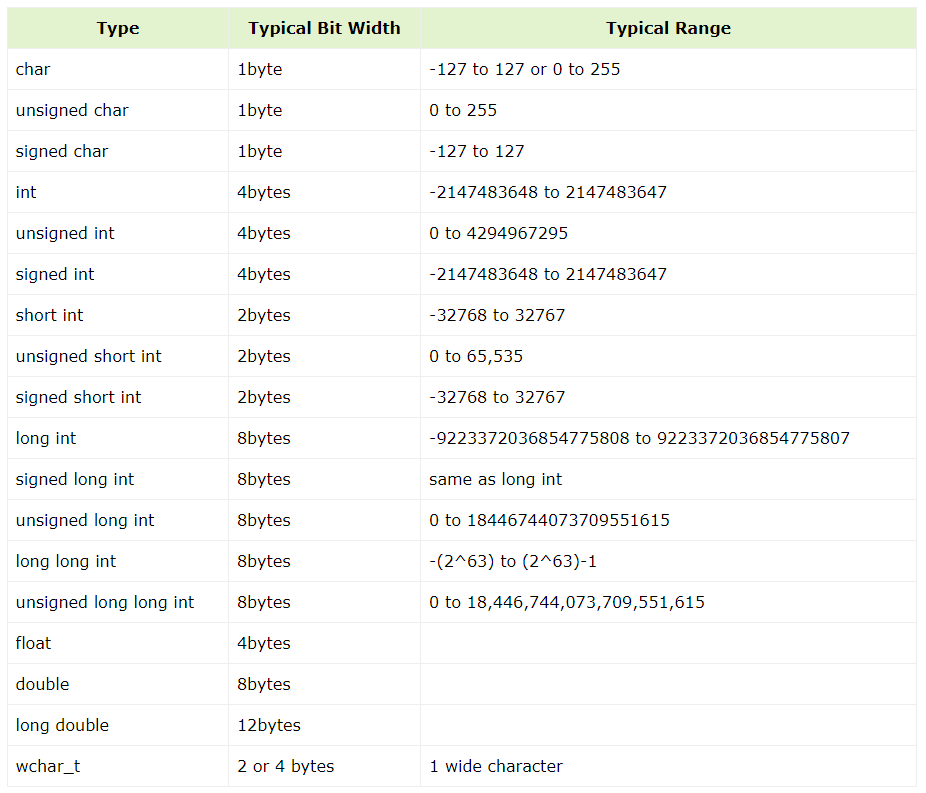
// Single Line Comment

**Common Header files:**

* #include <cstdlib> // Sorting, Searching, import c libraries, rand, memmgmt, and general-purpose functions
* #include <iostream> // Read and Write data
* #include <string> // Work with strings
* #include <limits> // Min and max values
* #include <vector> // Work with vectors
* #include <sstream> // Work with string streams
* #include <numeric> // Work with sequences of values
* #include <ctime> // Work with time
* #include <cmath> //Common math functions

**Data Types:**

A screenshot of a computer

Description automatically generated

**Variables:**

* Definition: type variable\_list = value;
* Ex: int i,j,k=10; char c,ch;

**Type Qualifiers:**

**A screenshot of a computer

Description automatically generated**

**Storage Qualifiers:**

**A table with green text

Description automatically generated**

**Input and Output:**

* cout << “Min int” << numeric\_limits<int>::min();
* cout << “Max short int” << numeric\_limits<short int>::max();
* printf(“Sum = %.7f\n”), (1.1111111+1.1111111)); // To print formatted output of float upto 7 decimal places
* cout << “int Byte:” << sizeof(int) << endl;
* printf(“%c %d %5d %.3f %s\n”, ‘A’, 10, 5, 3.1234, “Hi); // O/p: A 10 5 3.123 Hi //Right justify
* cin >> num\_str; //to take in input for num1
* int num1 = stoi(num\_str) //To convert num1 from string to int;
* bool res=true; cout.setf(ios::boolalpha); cout << res << endl; // To print booleans

A screenshot of a computer

Description automatically generated

**Operators:**

**Arithmetic Operators:**

**A screenshot of a computer

Description automatically generated**

**Logical Operators:**

**A screenshot of a computer

Description automatically generated**

**A white paper with black text

Description automatically generatedRelational Operators:**

**Bitwise Operators:**

**A screenshot of a computer program

Description automatically generated**

**Assignment Operators:**

**A screenshot of a computer

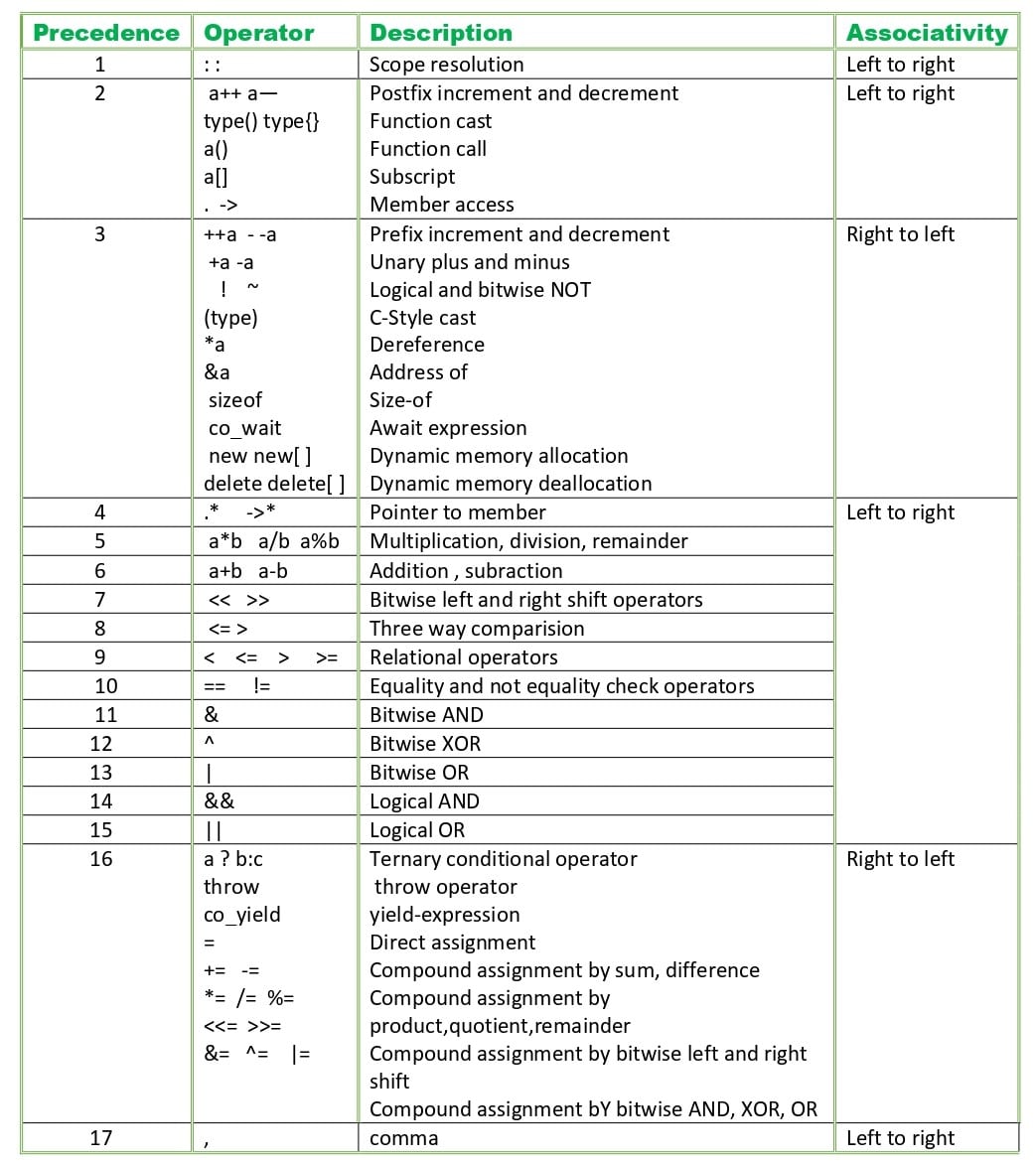
Description automatically generated**

**Misc Operators:**

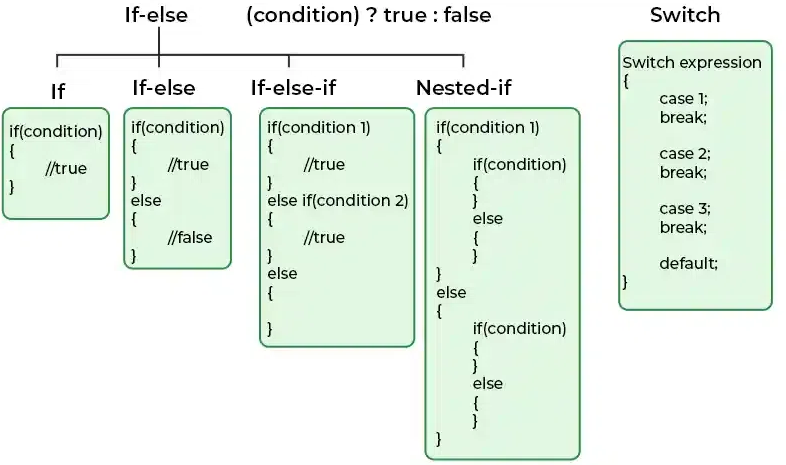
**A screenshot of a computer program

Description automatically generated**

**Precedence, Associativity:**



**Conditional Statements:**



A screenshot of a computer

Description automatically generated

**Loops:**

A white square with black text

Description automatically generated A white square with black text

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

while (i <= 20){

// If a value is even don't print it

if((i % 2) == 0){

i += 1;

// Continue skips the rest of the code

// and jumps back to the beginning

// of the loop

continue;

}

// Break stops execution of the loop and jumps

// to the line after the loops closing }

if(i == 15) break;

cout << i << "\n";

// Increment i so the loop eventually ends

i += 1;

}

// An abbreviated for loop

int arr3[] = {1,2,3};

for(auto x: arr3) cout << x << endl;

// Do while loops are guaranteed to execute at

// least once

// We'll create a secret number guessing game

// We need to seed the random number generator

// time() returns the number of seconds

// since 1, 1, 1970

// Include <ctime>

srand(time(NULL));

// Generate a random number up to 10

int secretNum = rand() % 11;

int guess = 0;

do{

cout << "Guess the Number : ";

cin >> guess;

if(guess > secretNum) cout << "To Big\n";

if(guess < secretNum) cout << "To Small\n";

} while(secretNum != guess);

cout << "You guessed it" << endl;

**Functions:**

* Return Type − A function may return a value. The return\_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword void.
* Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* Function Body − The function body contains a collection of statements that define what the function does.
* Syntax:

return\_type function\_name( parameter list ) {

body of the function

}

**Calling a Function:**

**A screenshot of a computer

Description automatically generated**

**Math Functions:**

cout << "abs(-10) = " << abs(-10) << endl;

cout << "max(5, 4) = " << max(5, 4) << endl;

cout << "min(5, 4) = " << min(5, 4) << endl;

cout << "fmax(5.3, 4.3) = " << fmax(5.3, 4.3) << endl;

cout << "fmin(5.3, 4.3) = " << fmin(5.3, 4.3) << endl;

cout << "ceil(10.45) = " << ceil(10.45) << endl;

cout << "floor(10.45) = " << floor(10.45) << endl;

cout << "round(10.45) = " << round(10.45) << endl;

cout << "pow(2,3) = " << pow(2,3) << endl;

cout << "sqrt(100) = " << sqrt(100) << endl;

cout << "cbrt(1000) = " << cbrt(1000) << endl;

// e ^ x

cout << "exp(1) = " << exp(1) << endl;

// 2 ^ x

cout << "exp2(1) = " << exp2(1) << endl;

// e \* e \* e ~= 20 so log(20.079) ~= 3

cout << "log(20.079) = " << log(20.079) << endl;

// 2 \* 2 \* 2 = 8

cout << "log2(8) = " << log2(8) << endl;

// Hypotenuse : SQRT(A^2 + B^2)

cout << "hypot(2,3) = " << hypot(2,3) << endl;

// Also sin, cos, tan, asin, acos, atan, atan2,

// sinh, cosh, tanh, asinh, acosh, atanh

**Arrays:**

* Syntax: type arrayName [size];
* Size once defined cannot be changed.
* Ex:

void main(int argc, char\*\*argv) {

int array1 [10] = {1}; // Size

int array2 [] = {1,2,3}; // Size for this would automatically be 3

int array3 [5] = {8,9}; //

cout << “First val: ” << array1[0] << endl;

array1[0] = 7;

int array4[2][3][3] = { {{1,2}, {3,4}}, {{5,6}, {7,8}} }; // Multidimensional arrays

cout << array4[0][1][1] <<endl //prints 4

return 0;

}

**Vectors:**

* Vectors are used when you don't know how big the array should be
* Syntax: template < class T, class Alloc = allocator<T> > class vector;
* Ex: vector<int> vNums(2);

// Add values

vNums[0] = 1;

vNums[1] = 2;

// Add another to the end

vNums.push\_back(3);

// Get vector size

cout << "Vector Size : " << vNums.size() << endl;

* vector::assign fill version // Assign new values to the vector elements by replacing old ones.
* vector::assign range version // Assign new values to the vector elements by replacing old ones.
* vector::assign initializer list version // Assign new values to the vector elements by replacing old ones.
* vector::at // Returns reference to the element present at location n in the vector.
* vector::back // Returns a reference to the last element of the vector.
* vector::begin // Return a random access iterator pointing to the first element of the vector.
* vector::capacity // Returns the size of allocate storage, expressed in terms of elements.
* vector::cbegin // Returns a constant random access iterator which points to the beginning of the vector.
* vector::cend // Returns a constant random access iterator which points to the beginning of the vector.
* vector::clear // Destroys the vector by removing all elements from the vector and sets size of vector to zero.
* vector::crbegin // Returns a constant reverse iterator which points to the reverser beginning of the container.
* vector::crend // Returns a constant reverse iterator which points to the reverse end of the vector.
* vector::data // Returns a pointer to the first element of the vector container.
* vector::emplace // Extends container by inserting new element at position.
* vector::emplace\_back // Inserts new element at the end of vector.
* vector::empty // Tests whether vector is empty or not.
* vector::end // Returns an iterator which points to past-the-end element in the vector container.
* vector::erase position version // Removes single element from the the vector.
* vector::erase range version // Removes single element from the the vector.
* vector::front // Returns a reference to the first element of the vector.
* vector::get\_allocator // Returns an allocator associated with vector.
* vector::insert single element version // Extends iterator by inserting new element at position.
* vector::insert fill version // Extends vector by inserting new element in the container.
* vector::insert range version // Extends vector by inserting new element in the container.
* vector::insert move version // Extends vector by inserting new element in the container.
* vector::insert initializer list version // Extends vector by inserting new element in the container.
* vector::max\_size // Returns the maximum number of elements can be held by vector.
* vector::operator= copy version // Assign new contents to the vector by replacing old ones and modifies size if necessary.
* vector::operator= move version // Assign new contents to the vector by replacing old ones and modifies size if necessary.
* vector::operator = initializer list version // Assign new contents to the vector by replacing old ones and modifies size if necessary.
* vector::operator[] // Returns a reference to the element present at location n.
* vector::pop\_back // Removes last element from vector and reduces size of vector by one.
* vector::push\_back // Inserts new element at the end of vector and increases size of vector by one.
* vector::rbegin // Returns a reverse iterator which points to the last element of the vector.
* vector::rend // Returns a reverse iterator which points to the reverse end of the vector.
* vector::reserve // Requests to reserve vector capacity be at least enough to contain n elements.
* vector::resize // Changes the size of vector.
* vector::shrink\_to\_fit // Requests the container to reduce it's capacity to fit its size.
* vector::size // Returns the number of elements present in the vector.
* vector::swap // Exchanges the content of vector with contents of vector x

**String Streams:**

// A stringstream object receives strings separated

// by a space and then spits them out 1 by 1

vector<string> words;

stringstream ss("Some Random Words");

string word;

// A while loop will execute as long as there are

// more words

while(getline(ss, word, ' ')){

words.push\_back(word);

}

// Cycle through each index in the vector using

// a for loop

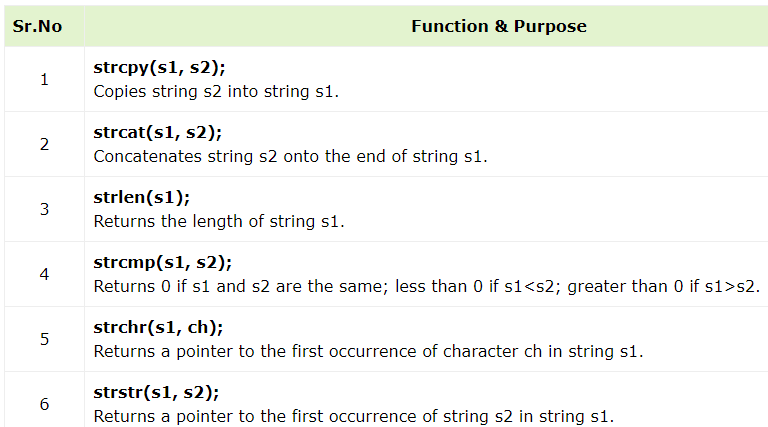
for(int i = 0; i < words.size(); ++i){

cout << words[i] << endl;

}

**Strings:**

* A C++ string is a series of characters that can be changed



* string str1 = "I'm a string";

// Get the 1st character

cout << "1st : " << str1[0] << endl;

// Get the last character

cout << "Last : " << str1.back() << endl;

// Get the string length

cout << "Length : " << str1.length() << endl;

// Copy a string to another

string str2 = str1;

// Copy a string after the 1st 4 characters

string str3(str2, 4);

// Combine strings

string str4 = str1 + " and your not";

// Append to the end of a string

str4.append("!");

// Erase characters from a string from 1 index to another

str4.erase(12, str4.length() - 1);

cout << "New String : " << str4 << endl;

// find() returns index where pattern is found or npos (End of String)

if(str4.find("string") != string::npos)

cout << "String Index : " << str4.find("string") << endl;

// O/p: String Index: 6

// substr(x, y) returns a substring starting at index x with a length of y

cout << "Substring : " << str4.substr(6,6) << endl;

//O/p: Substring: string

// Convert int to string

string strNum = to\_string(1+2);

cout << "I'm a String : " << strNum << "\n";

//O/p: I’m a String: 3

**Character functions**

char letterZ = 'z';

char num5 = '5';

char aSpace = ' ';

cout << "Is z a letter or number " <<

isalnum(letterZ) << endl;

cout << "Is z a letter " <<

isalpha(letterZ) << endl;

cout << "Is 3 a number " <<

isdigit(num5) << endl;

cout << "Is space a space " <<

isspace(aSpace) << endl;

**Pointers:**

type \*var-name;

A screenshot of a computer

Description automatically generated

**References:**

* A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
* You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
* Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
* A reference must be initialized when it is created. Pointers can be initialized at any time.
* Declaration: int& r = i;
* References as Parameters: C++ supports passing references as function parameter more safely than parameters.
* Reference as Return Value: You can return reference from a C++ function like any other data type.

A screenshot of a computer code

Description automatically generated

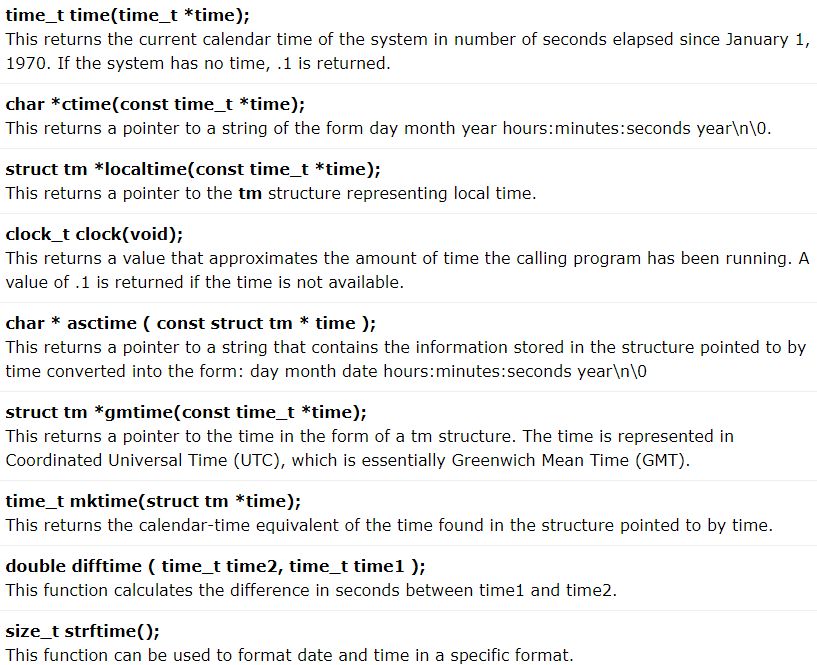
Value of i : 5

Value of i reference : 5

Value of d : 11.7

Value of d reference : 11.7

**Date and Time:**

****

**Structures:**

* The struct statement defines a new data type, with more than one member, for your program.
* Format:

struct [structure tag] {

member definition;

member definition;

...

member definition;

} [one or more structure variables];

* Ex:

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

} book;

* Pointers: struct Books \*struct\_pointer = &Book1;
* typedef:

typedef struct {

char title[50];

char author[50];

char subject[100];

int book\_id;

} Books;

Books Book1, Book2;

**Classes:**

* Enhances C programming with object orientation; classes form the backbone for object-oriented programming.
* Comprises data and functions, termed as class members.
* Class Member Functions: Functions defined or prototyped within a class.
* Class Access Modifiers: Specifying access levels (public, private, protected).
* Constructor & Destructor: Special functions for object creation and deletion.
* Copy Constructor: Initializes an object with another of the same class.
* Friend Functions: Accesses private/protected class members.
* Inline Functions: Compiler attempts to replace function calls with function body.
* 'this' Pointer: Points to the object itself within a class.
* Pointer to C++ Classes: Similar to pointers in structures.
* Static Members: Data or function members declared as static.
* **Ex:**

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main() {

Box Box1; // Declare Box1 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Volume of Box1 : " << volume <<endl;

return 0;

}

**Inheritance:**

* Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.
* Syntax: class derived-class: access-specifier base-class
* Multiple inheritance: class derived-class: access baseA, access baseB....

Ex: class Rectangle: public Shape, public PaintCost

* A screenshot of a computer

  Description automatically generated

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

**Overloading:**

**Function Overloading:**

* You can have multiple definitions for the same function name in the same scope.
* The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.
* You cannot overload function declarations that differ only by return type.

#include <iostream>

using namespace std;

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

pd.print(5);

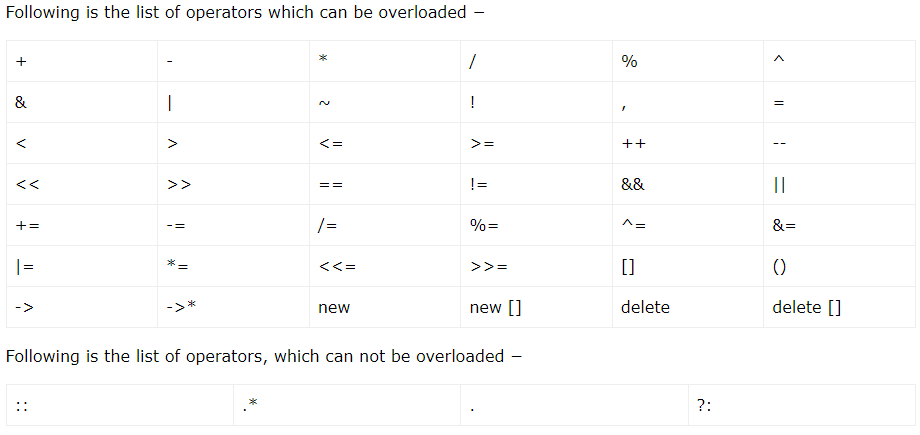
pd.print(500.263);

pd.print("Hello C++");

return 0;

}

Operator Overloading:

* You can redefine or overload most of the built-in operators available in C++
* Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined.
* 

#include <iostream>

using namespace std;

class Box {

public:

double getVolume(void) {

return length \* breadth \* height;

}

void setLength( double len ) {

length = len;

}

void setBreadth( double bre ) {

breadth = bre;

}

void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

Box operator+(const Box& b) {

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

Box3 = Box1 + Box2;

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

**Polymorphism:**

* **Virtual Function:** A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.
* What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding.
* **Pure virtual function:** A pure virtual function (or abstract function) in C++ is a virtual function for which we don’t have an implementation, we only declare it. A pure virtual function is declared by assigning 0 in the declaration.
* A virtual function is a member function of base class which can be redefined by derived class. A **pure virtual function** is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.

Ex:

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0){

width = a;

height = b;

}

**virtual int area() {**

cout << "Parent class area :" << width \* height << endl;

return width \* height;

}

};

class Rectangle: public Shape {

public:

Rectangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Rectangle class area :" << width \* height << endl;

return (width \* height);

}

};

class Triangle: public Shape {

public:

Triangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Triangle class area :" << (width \* height)/2 << endl;

return (width \* height / 2);

}

};

// Main function for the program

int main() {

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

shape = &rec;

shape->area();

shape = &tri;

shape->area();

return 0;

}

// Defining the area function in the base shape class with virtual prevents it from overriding the area functions in derived classes when their objects call area.

Files and Streams:

* Header files required: iostream and fstream
* A screenshot of a computer

  Description automatically generated
* File position pointers:

// position to the nth byte of fileObject (assumes ios::beg)

fileObject.seekg( n );

// position n bytes forward in fileObject

fileObject.seekg( n, ios::cur );

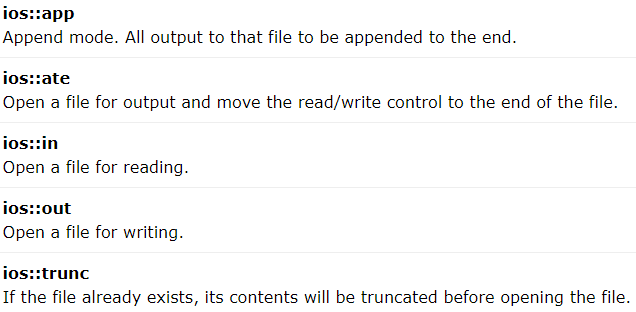
// position n bytes back from end of fileObject

fileObject.seekg( n, ios::end );

// position at end of fileObject

fileObject.seekg( 0, ios::end );

* File Modes:



* void open(const char \*filename, ios::openmode mode);
* Ex:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

// Creation of ofstream class object

ofstream fout;

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) {

// Read a Line from standard input

getline(cin, line);

// Press -1 to exit

if (line == "-1")

break;

// Write line in file

fout << line << endl;

}

// Close the File

fout.close();

// Creation of ifstream class object to read the file

ifstream fin;

// by default open mode = ios::in mode

fin.open("sample.txt");

// Execute a loop until EOF (End of File)

while (getline(fin, line)) {

cout << line << endl;

}

fin.close();

return 0;

}

**Exception handling:**

* throw − A program throws an exception when a problem shows up. This is done using a throw keyword.
* catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
* try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
* Syntax:

try {

// protected code

} catch( ExceptionName e1 ) {

// catch block

} catch( ExceptionName e2 ) {

// catch block

} catch( ExceptionName eN ) {

// catch block

}

Ex:

#include <iostream>

#include <exception>

using namespace std;

struct MyException : public exception {

const char \* what () const throw () {

return "C++ Exception";

}

};

int main() {

try {

throw MyException();

} catch(MyException& e) {

std::cout << "MyException caught" << std::endl;

std::cout << e.what() << std::endl;

} catch(std::exception& e) {

//Other errors

}

}