```
/*
 * 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:

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-9223372036854775808 to 9223372036854775807
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 18446744073709551615
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	
long double	12bytes	
wchar_t	2 or 4 bytes	1 wide character

Data Type	Initializer
int	0
char	'\0'
float	0
double	0
pointer	NULL

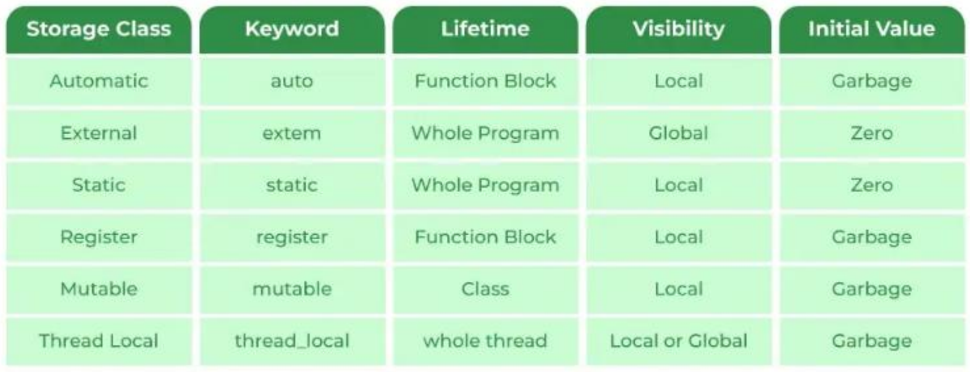
Variables:

- Definition: type variable_list = value;
- Ex: int i,j,k=10; char c,ch;

Type Qualifiers:

Sr.No	Qualifier & Meaning
1	**const** Objects of type **const** cannot be changed by your program during execution.
2	**volatile** The modifier **volatile** tells the compiler that a variable's value may be changed in ways not explicitly specified by the program.
3	**restrict** A pointer qualified by **restrict** is initially the only means by which the object it points to can be accessed. Only C99 adds a new type qualifier called restrict.

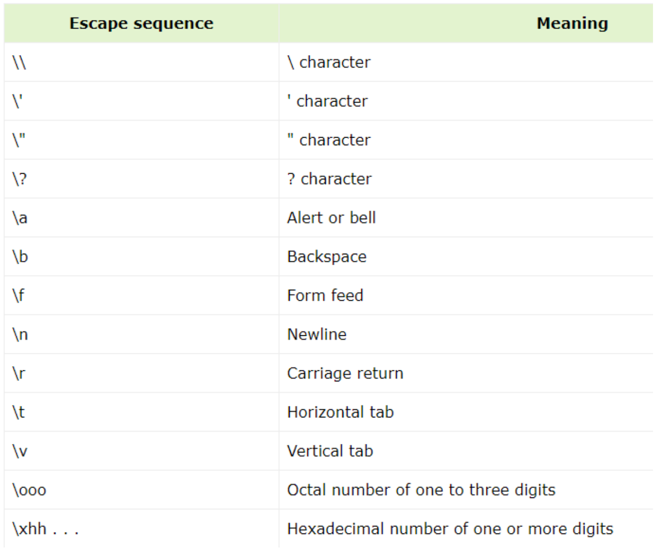
Storage Qualifiers:



Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extem	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage
Thread Local	thread_local	whole thread	Local or Global	Garbage

Input and Output:

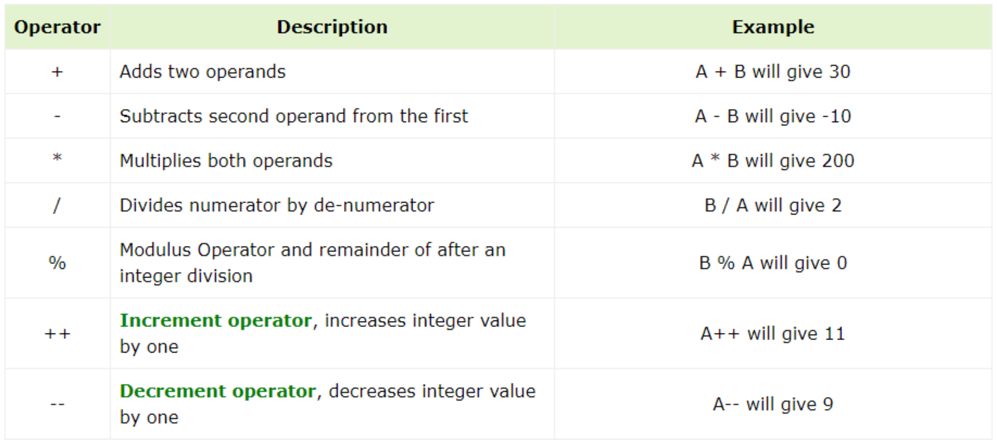
- cout << “Min int” << numeric_limits<int>::min();
- cout << “Max short int” << numeric_limits<short int>::max();
- printf(“Sum = %.7f\n”), (1.1111111+1.1111111)); // To print formatted output of float upto 7 decimal places
- cout << “int Byte:” << sizeof(int) << endl;
- printf(“%c %d %5d %.3f %s\n”, ‘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



Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

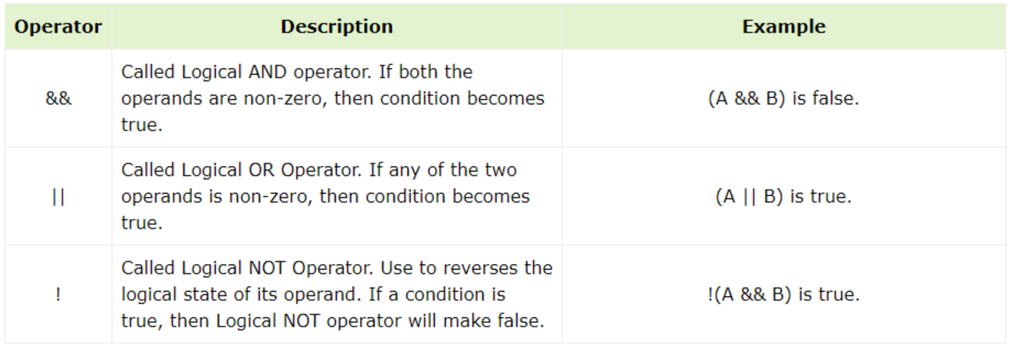
Operators:

Arithmetic Operators:



Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	**Increment operator**, increases integer value by one	A++ will give 11
--	**Decrement operator**, decreases integer value by one	A-- will give 9

Logical Operators:



Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

Relational Operators:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Bitwise Operators:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

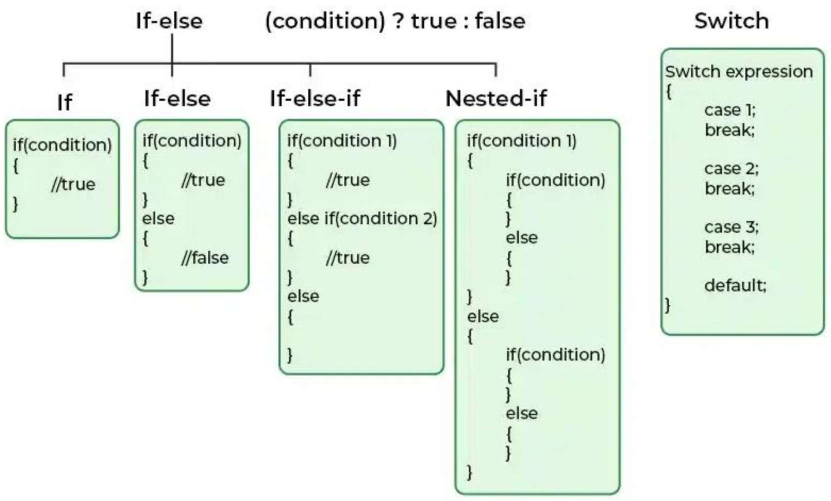
Misc Operators:

Sr.No	Operator & Description
1	**sizeof** **sizeof operator** returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	**Condition ? X : Y** **Conditional operator (?)**. If Condition is true then it returns value of X otherwise returns value of Y.
3	**,** **Comma operator** causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	**. (dot) and -> (arrow)** **Member operators** are used to reference individual members of classes, structures, and unions.
5	**Cast** **Casting operators** convert one data type to another. For example, int(2.2000) would return 2.
6	**&** **Pointer operator &** returns the address of a variable. For example &a; will give actual address of the variable.
7	***** **Pointer operator *** is pointer to a variable. For example *var; will pointer to a variable var.

Precedence, Associativity:

Precedence	Operator	Description	Associativity
1	: :	Scope resolution	Left to right
2	a++ a— type() type{} a() a[] . ->	Postfix increment and decrement Function cast Function call Subscript Member access	Left to right
3	++a - -a +a -a ! ~ (type) *a &a sizeof co_wait new new[ ] delete delete[ ]	Prefix increment and decrement Unary plus and minus Logical and bitwise NOT C-Style cast Dereference Address of Size-of Await expression Dynamic memory allocation Dynamic memory deallocation	Right to left
4	.* ->*	Pointer to member	Left to right
5	a*b a/b a%b	Multiplication, division, remainder	
6	a+b a-b	Addition , subraction	
7	<< >>	Bitwise left and right shift operators	
8	<= >	Three way comparision	
9	< <= > >=	Relational operators	
10	== !=	Equality and not equality check operators	
11	&	Bitwise AND	
12	^	Bitwise XOR	
13	|	Bitwise OR	
14	&&	Logical AND	
15	||	Logical OR	
16	a ? b:c throw co_yield = += -= *= /= %= <<= >>= &= ^= |=	Ternary conditional operator throw operator yield-expression Direct assignment Compound assignment by sum, difference Compound assignment by product,quotient,remainder Compound assignment by bitwise left and right shift Compound assignment bY bitwise AND, XOR, OR	Right to left
17	,	comma	Left to right

Conditional Statements:



```
if(condition)
{
        //true
}
```

```
if(condition)
{
        //true
}
else
{
        //false
}
```

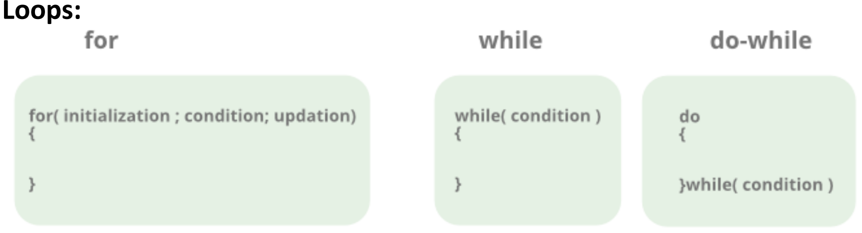
```
if(condition 1)
{
        //true
}
else if(condition 2)
{
        //true
}
else
{

}
```

```
if(condition 1)
{
        if(condition)
        {
        }
        else
        {
        }
}
else
{
        if(condition)
        {
        }
        else
        {
        }
}
```

```
Switch expression
{
        case 1;
        break;

        case 2;
        break;

        case 3;
        break;

        default;
}
```

Sr.No	Statement & Description
1	**if statement** An 'if' statement consists of a boolean expression followed by one or more statements.
2	**if...else statement** An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
3	**switch statement** A 'switch' statement allows a variable to be tested for equality against a list of values.
4	**nested if statements** You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
5	**nested switch statements** You can use one 'switch' statement inside another 'switch' statement(s).

Loops:



```
for( initialization ; condition; updation)
{

}
```

```
while( condition )
{

}
```

```
do
{

}while( condition )
```

Sr.No	Loop Type & Description
1	**while loop** Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	**for loop** Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**do...while loop** Like a 'while' statement, except that it tests the condition at the end of the loop body.
4	**nested loops** You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

Sr.No	Control Statement & Description
1	**break statement** Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**goto statement** Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

```
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;
```

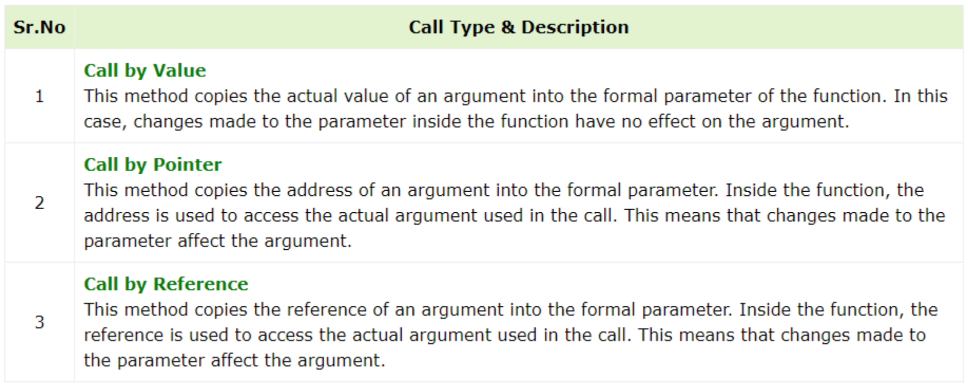
Functions:

- 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:



Sr.No	Call Type & Description
1	**Call by Value** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	**Call by Pointer** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	**Call by Reference** This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

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.

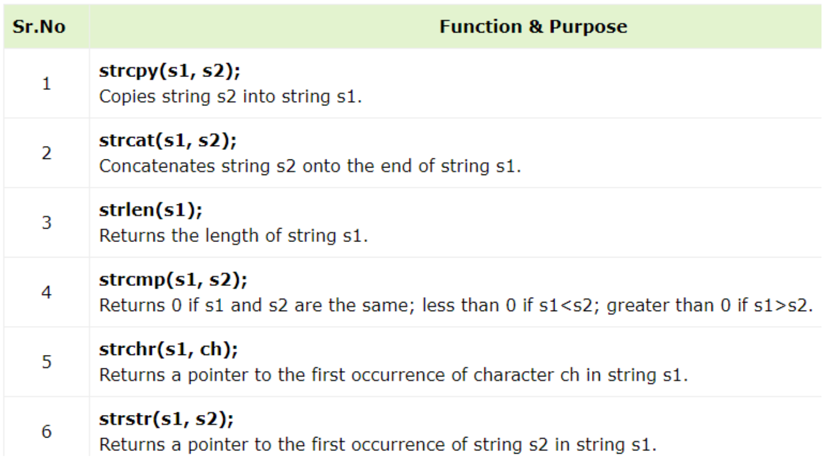
- vector::insert move version // Extends vector by inserting new element in the container.
- vector::insert initializer list version // Extends vector by inserting new element in the container.
- vector::max_size // Returns the maximum number of elements can be held by vector.
- vector::operator= copy 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



Sr.No	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

- 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
```

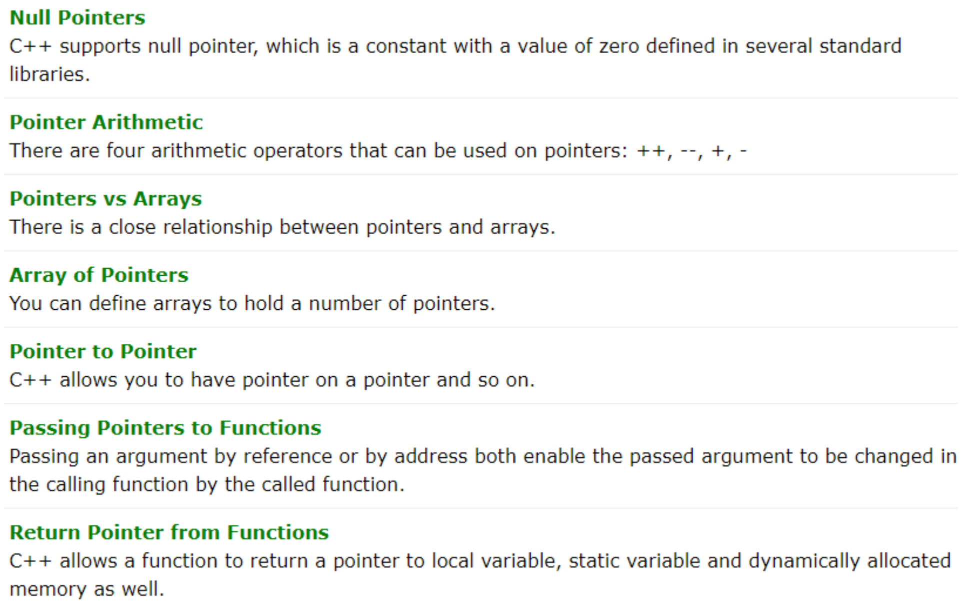
Character functions

```
  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;



Null Pointers

C++ supports null pointer, which is a constant with a value of zero defined in several standard libraries.

Pointer Arithmetic

There are four arithmetic operators that can be used on pointers: ++, --, +, -

Pointers vs Arrays

There is a close relationship between pointers and arrays.

Array of Pointers

You can define arrays to hold a number of pointers.

Pointer to Pointer

C++ allows you to have pointer on a pointer and so on.

Passing Pointers to Functions

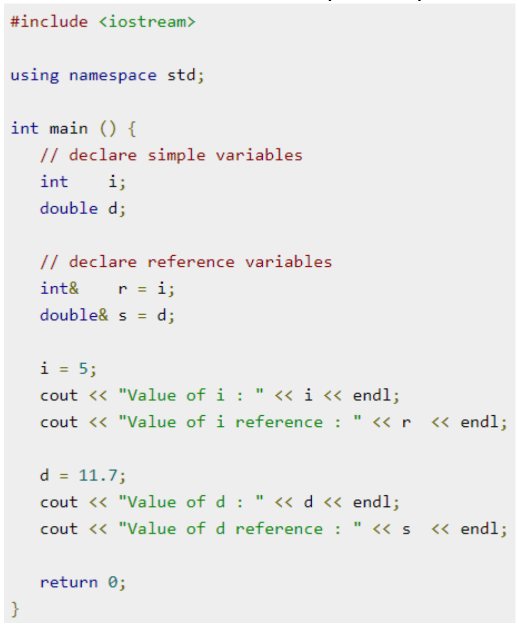
Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.

Return Pointer from Functions

C++ allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.

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.
- References are basically const pointers.



```
#include <iostream>

using namespace std;

int main () {
   // declare simple variables
   int    i;
   double d;

   // declare reference variables
   int&    r = i;
   double& s = d;

   i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;

   d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s  << endl;

   return 0;
}
```

Value of i : 5

Value of i reference : 5

```
Value of d : 11.7
Value of d reference : 11.7
```

Date and Time:

time_t time(time_t *time);
This returns the current calendar time of the system in number of seconds elapsed since January 1, 1970. If the system has no time, .1 is returned.

char *ctime(const time_t *time);
This returns a pointer to a string of the form day month year hours:minutes:seconds year\n\0.

struct tm *localtime(const time_t *time);
This returns a pointer to the **tm** structure representing local time.

clock_t clock(void);
This returns a value that approximates the amount of time the calling program has been running. A value of .1 is returned if the time is not available.

char * asctime (const struct tm * time);
This returns a pointer to a string that contains the information stored in the structure pointed to by time converted into the form: day month date hours:minutes:seconds year\n\0

struct tm *gmtime(const time_t *time);
This returns a pointer to the time in the form of a tm structure. The time is represented in Coordinated Universal Time (UTC), which is essentially Greenwich Mean Time (GMT).

time_t mktime(struct tm *time);
This returns the calendar-time equivalent of the time found in the structure pointed to by time.

double difftime (time_t time2, time_t time1);
This function calculates the difference in seconds between time1 and time2.

size_t strftime();
This function can be used to format date and time in a specific format.

Structures:

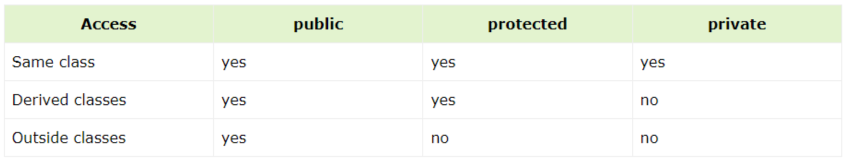
- The struct statement defines a new data type, with more than one member, for your program.
- Format:
```
struct [structure tag] {
  member definition;
  member definition;
  ...
  member definition;
} [one or more structure variables];
```
- Ex:
```
struct Books {
  char  title[50];
  char  author[50];
  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
- 

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

-

```
#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:

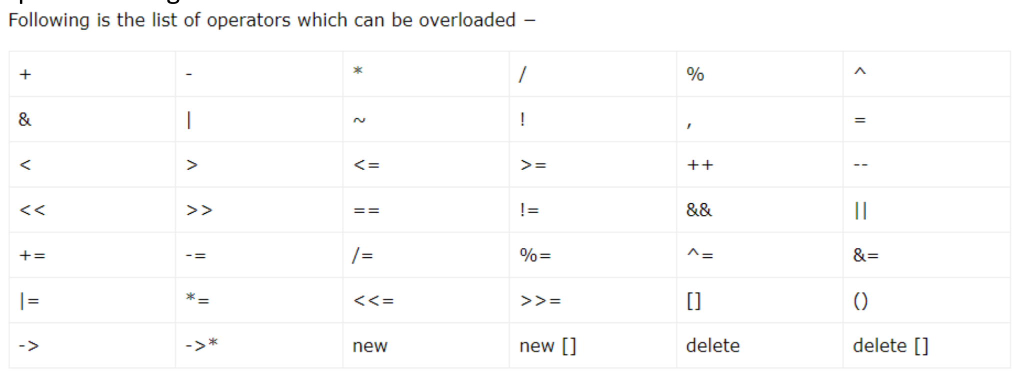
- You can have multiple definitions for the same function name in the same scope.
- The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.
- You cannot overload function declarations that differ only by return type.

```
#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;
   }
```

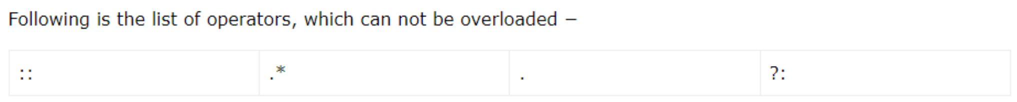
Operator Overloading:

- 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.



Following is the list of operators which can be overloaded –

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []



Following is the list of operators, which can not be overloaded –

::	.*	.	?:

-

```
#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.

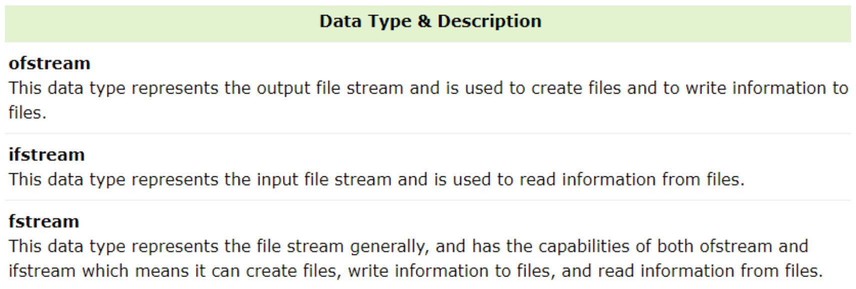
Ex:

```
#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

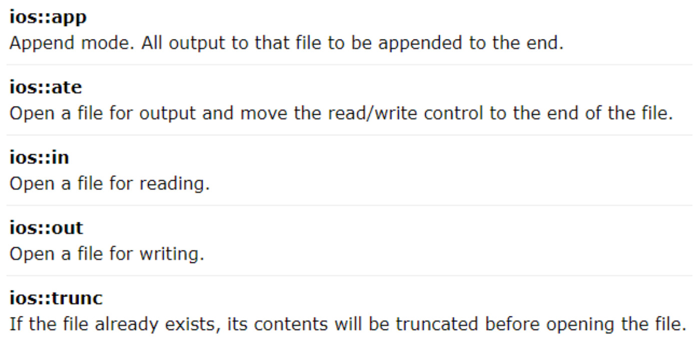


Data Type & Description
ofstream This data type represents the output file stream and is used to create files and to write information to files.
ifstream This data type represents the input file stream and is used to read information from files.
fstream This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

- 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:



ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

- 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
  }
}
```

Dynamic Memory:

- Memory in C++ program is divided into two parts:
- The stack − All variables declared inside the function will take up memory from the stack.
- The heap − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.
- You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.
- If you are not in need of dynamically allocated memory anymore, you can use delete operator, which de-allocates memory that was previously allocated by new operator.
- Syntax: new data-type; delete var;
- The malloc() function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

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

int main () {
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;         // free up the memory.

   return 0;
}
```

Ex2:

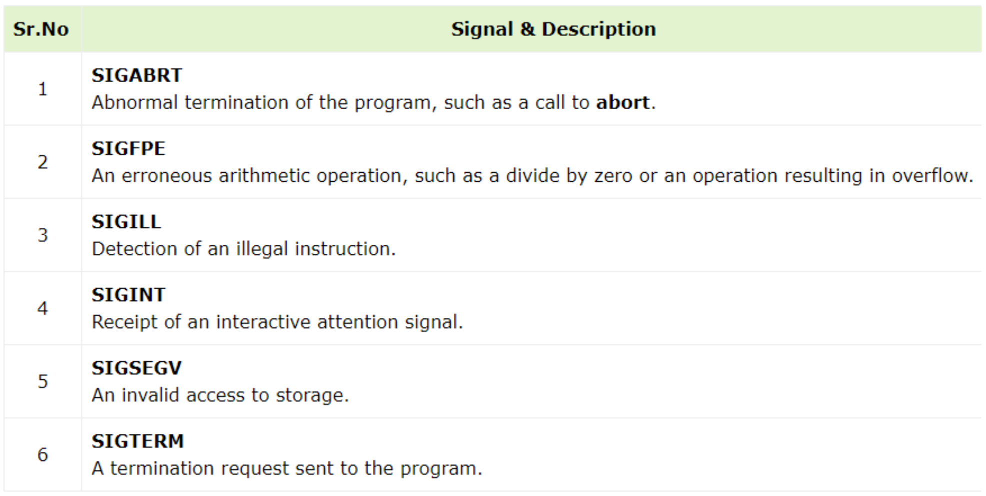
```
#include <iostream>
using namespace std;

class Box {
   public:
      Box() {
         cout << "Constructor called!" <<endl;
      }
      ~Box() {
         cout << "Destructor called!" <<endl;
      }
};
int main() {
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // Delete array

   return 0;
}
```

Namespaces:

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

// first name space
```

```
namespace first_space {
  void func() {
    cout << "Inside first_space" << endl;
  }
}

// second name space
namespace second_space {
  void func() {
    cout << "Inside second_space" << endl;
  }
}

int main () {
  // Calls function from first name space.
  first_space::func();

  // Calls function from second name space.
  second_space::func();

  return 0;
}
```

Signal Handling:



Sr.No	Signal & Description
1	**SIGABRT** Abnormal termination of the program, such as a call to **abort**.
2	**SIGFPE** An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.
3	**SIGILL** Detection of an illegal instruction.
4	**SIGINT** Receipt of an interactive attention signal.
5	**SIGSEGV** An invalid access to storage.
6	**SIGTERM** A termination request sent to the program.

Templates: