This manual describes C++ programming language.

1. Lexical Conventions

1.1 Comments

The characters /* introduce a comment, which terminate with characters */.

// used for single line comments.

1.2 Identifiers

Sequence of characters used for naming variables, functions, new data types etc.

Identifiers Can include only letters, digits and '_' character.

First character cannot be a digit.

Lowercase and uppercase letters are distinct.

Examples: _start, my123, t_name, etc.

1.3 Keywords

Some special identifiers are reserved to use as part of programming language itself. These special identifiers are called keywords. They can't be used for any other purpose.

asm, auto, break, case, catch, char, class, const, continue, default, delete, do, double, else, enum, extern, float, for, friend, goto, if, inline, int, long, new, operator, private, protected, public, register, return, short, signed, sizeof, static, struct, switch, template, this, throw, try, typedef, union, unsigned, virtual, void, volatile, while.

1.4 Constants

1.4.1 Integer constant

Hexadecimal (base 16) - If the sequence of digits is preceded by '0x' or '0X' then it will be considered as hexadecimal. Hexadecimal values may use the digits from 0 to 9, as well as the letters a to f and A to F. examples:

0x11, 0Xad, 0xad3f

Octal - if the first digit is 0 and the remaining digits are in between 0 and 7, then constant will be considered as octal. Examples

034, 077, 023

Decimal – in all other cases constant will be considered as decimal. Decimal values can use digit in between 0 and 9.

345, 7890, 287

1.4.2 Character constant – a character constant is a character enclosed in single quotes, as 'S'. characters constants are taken to be int.

\\ Backslash character

\? Question mark

\n new-line

\t horizontal tab

\' single quote

\" double quote

1.4.3 Floating constants

It consists of a sequence of digits which represents the integer (or "whole") part of the number, a decimal point, and a sequence of digits which represents the fractional part. it can also be followed by e or E and an integer exponent.

Example:

4.322, 9.0, .543, 4e4(4*10000).

1.4.5 String constants

A string is a sequence of characters, digits and escape sequences surrounded by double quotes as "…". String constant is of type 'array of characters'. All string constants contain a null termination character (\0) as their last character.

Example: " Go to market." , " opposite of true is \" false \"."

2.Data Types

All variables use datatype during declaration to restrict the type of data to be stored. Data types are used to tell the variables the type of data it can store. Every data type requires a different amount of memory.

2.1 integers: integer data type is used for storing integers. Keyword used for integer data types is **int**.

-short int – requires 2 byte and ranges from -32,768 to 32,767
- unsigned short int – requires 2 byte and ranges from 0 to 65,535
- unsigned int – requires 4 byte and ranges from 0 to (2^32)-1
- int – requires 4 byte and ranges from – (2^31) to (2^31)-1
-long long int – requires 8 byte and ranges from – (2^63) to (2^63)-1
- unsigned long long int - requires 8 byte and ranges from 0 to (2^64)-1

Ex: int a=4657; long long int k = 647352754; short int s = -4938;

2.2 Characters: Character data type is used for storing characters. Keyword used for character data type is **char**. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.

- signed char – requires 1 byte and ranges from -128 to 127.
- unsigned char – requires 1 byte and ranges from 0 to 256.

Ex: char temp = 'a'; char temp2 = 65;

2.3 Boolean: Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true* or *false*. Keyword used for boolean data type is **bool**.

Ex: bool item = true;

2.4 Floating Point: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is **float**. Float variables typically requires 4 byte of memory space.

Ex: float k = 5.43; float t = 4e-4;

3. Control structures

3.1 Conditionals (if, if then else)

if – else statement is used to conditionally execute part of program, based on the truth value of given test expression.

```
If (test expression)
        Statement 1
else
        statement 2
```

if test expression evaluates to true then statement 1 is executed and statement 2 is ignored. And if test expression evaluates to false then statement 2 is executed and statement 1 is ignored.

Example:

```
if (x == 1)
        cout<< "x is 1";
else
        cout<<"x is not 1";
```

if (x==1) is evaluated to true then output will be **x is 1**, and else part will be ignored. if (x==1) evaluated to false then output will be **x is not 1**.

if can also be used without **else** like

```
if (x == 1)
        cout<< "x is 1";
```

but **else** can't be used alone without a prior **if**.

we can also use multiple if else statements to check multiple conditions like

```
if (x == 1)
        cout<< "x is 1";
else if(x==2)
        cout<< "x is  2";
else if (x == 3)
        cout <<" x is 3";
else
        cout<<" x is something else";
```

3.2 Loops

3.2.1 while

```
    while (test expression)
                Statement
```

The statement executed repeatedly so long as the value of test expression remains non-zero. The test expression is checked before each execution of the statement.

Example: following loop will print numbers from 1 to 9.

```
int x = 1;
while (x<10) {
        cout <<x;
}
```

3.2.2 do while

```
do
  statement
    while(expression)
```

The statement executed repeatedly so long as the value of expression becomes zero. The test expression is checked after each execution of the statement.

3.2.3 for

```
for (statement1; expression1; expression2)
        statement2
```

statement 1 specifies initialization for the loop, expression1 specifies a test, checked before every iteration, if this evaluates to true then statement2 is executed and if this evaluates to false then loop is exited. expression 2 specifies how progress made after each iteration.

Example: following loop will print number 1 to 9. We initialize x to 1 and then after each iteration it will increment by 1 and loop will terminate when x becomes 10.

```
int x;
for (x=1; x<10;x++)
    cout<<x;
```

All three arguments (statement1, expression1, expression2) are optional and any combination of three is valid. Different examples are shown below-

following loop will print 0 indefinitely because there is not any progress neither any condition to test after each iteration.

```
int x=0;
for (; ;)
    cout<<x;
```

following loop will print whole numbers indefinitely because only progress is made but no test condition is provided. If test condition is removed, then it will be an infinite loop. This is same as using while (1) which is always true and that's why loop never terminate.

```
int x;
for (x=0; ; x++)
    cout<<x;
```

following loop will print 0 indefinitely times because no progress is made after a iteration.

```
int x;
for (x=0; x<2;)
    cout<<x;
```

4.Variables

A variable is a container (storage area) to hold data. It is the basic unit of storage in a program. the value stored in a variable can be changed during program execution. all the variables must be declared before use.

A variable name is an arbitrarily long sequence of letters and digits. it can consist of alphabets (both upper and lower case), numbers and the underscore '_' character. However, the name must not start with a number.

A typical variable declaration is of the form:

Declaring a single variable: { Datatype variable_name; }

Ex: **int** year; **char** name; **float** Balance;

Declaring multiple variables: Type of all variables will be same. {Datatype variable1_name, variable2_name, variable3_name;}

Ex: int year, month, day, hour;

Types of variables

Local variables: A variable defined within a block or method or constructor or function is called a local variable. The scope of these variables exists only within the block in which the variable is declared. i.e., we can access these variables only within that block.

Global variables: A global variable is a variable declared in the main body of the source code, outside all function. global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration

Example: in following code ***temp*** is global variable while ***temp2*** is a local variable.

```
int temp;
int func1(….){
        int temp2;
         ….. }
```

5. Expression and Operators

5.1 Bit shifting

1. Left shift – left shift operator (<<) is used to shift its first operand's bits to left. The second operand denotes the number of bit places to shift. Bits shifted off the left side of the value are discarded; new bits added on the right side will all be 0.

Ex: 10100110 << 2 = 10011000

2. Right shift – right shift operator (>>) is used to shift its first operand's bits to right. The second operand denotes the number of bit places to shift. Bits shifted off the right side of the value are discarded; new bits added on the left side are usually 0, but if the first operand is a signed negative value, then the added bits will be either 0 *or* whatever value was previously in the leftmost bit position.

Ex: 00101111 >> 2 = 00001011

5.2 Bitwise Logical operators

1. Conjunction (&) - it examines each bit in its two operands, and when two corresponding bits are both 1, the resulting bit is 1. All other resulting bits are 0.

Ex: 1101101 & 1000111 = 1000101

2. Inclusive Disjunction (|) - - it examines each bit in its two operands, and when two corresponding bits are both 0, the resulting bit is 0. All other resulting bits are 1.

Ex: 1100101 & 1000111 = 1100111

3. Exclusive Disjunction (^) - it examines each bit in its two operands, and when two corresponding bits are different the resulting bit is 1. All other resulting bits are 0.

Ex: 1100101 & 1000110 = 1011100

4. Negation (~) – it reverses each bit in its operand.

Ex: ~11001001 = 00110110

5.3 Logical operators

Logical operators test the truth value of a pair of operands. Any nonzero expression is considered true, while an expression that evaluates to zero is considered false.

1. Conjunction (&&) - it tests if two expressions are both true. If the first expression is false, then the second expression is not evaluated.

2. Disjunction (||) – it tests if at least one of two expressions is true. If the first expression is true, then the second expression is not evaluated.

3. Negation (!) – it flip the truth value of it's operand.

5.4 Assignment operators

The standard assignment operator = simply stores the value of its right operand in the variable specified by its left operand. Left operand cannot be a value. Some compound operators-

x+= y is same as x = x+y

x-=y is same as x = x-y

x*=y is same as x = x*y

x/=y is same as x = x/y

x%=y is same as x = x%y

x<<=y is same as x = x<<y

x>>=y is same as x = x>>y

x&=y is same as x = x&y

x^=y is same as x = x^y

x|=y is same as x = x|y

5.5 Comparison Operators

Comparison operators determine how two operands relate to each other. These operators produce result either 1 or 0 meaning true or false, respectively.

1. Equal-to (==): it tests two operands for equality. If operands are equal result is 1, and 0 if they are not equal.

2. Not-equal-to (!=): it tests two operands for equality. If operands are equal result is 0, and 1 if they are not equal.

3. Greater than (>): checks if left operand is greater than right operand. If it is then result is 1, otherwise 0.

4. Greater-than-or-equal-to (>=): checks if left operand is greater than or equal to right operand. If it is then result is 1, otherwise 0.

5. Less than (<): checks if left operand is less than right operand. If it is then result is 1, otherwise 0.

6. Less-than-or-equal-to (<=): checks if left operand is less than or equal to right operand. If it is then result is 1, otherwise 0.

5.5 Expression

An expression is an order collection of operators and operands which specifies a computation. An expression can contain zero or more operators and one or more operands, operands can be constants or variables. In addition, an expression can contain function calls as well which return constant values.

Expressions may be of the following types:

Integral expressions**:** Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions.

Ex: int x = 34 + 643;

Relational expressions: Relational Expressions yield results of type bool which takes a value true or false.

Ex: (x == y); , (x >= y);

Logical expressions: Logical Expressions combine two or more relational expressions and produces bool type results.

Ex: (x == 5 && y == 6), (x >=y || y == 1)

Pointer expressions: Pointer Expressions produce address values.

Ex: *ptr1 + *ptr2 - *ptr3

Bitwise expressions: Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.

Ex: int x = 20; x >>= 1;

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions.

6. Input / Output

C++ uses a convenient abstraction called streams to perform input and output operations in sequential media such as the screen, the keyboard, or a file. A stream is an entity where a program can either insert or extract characters to/from.

Input:

cin : Standard input stream

cin is used with extraction operator, which is written as ≫.

Ex: int marks; int color; cin≫marks≫color;

Output :

cout : Standard output stream

cout is used with insertion operator, which is written as ≪.

Ex: int marks = 99; int color =55; cout≪marks<<" "≪color; Output : 99 55

7. Arrays

An array is a data structure which allows you to store one or more elements consecutively in memory. In **C++,** array elements, array elements are indexed beginning at position 0.

Declaration:

Array is declared by specifying the data type for its elements, its name, and the number of elements it stores.

Ex: int arr[10]; char my_char_arr[6];

Initializing Arrays:

Array can be initialized during declaration by listing all the value in curly braces separated by commas. If number of elements let's say k in braces are less than the size of array let's say n then First k elements of array will be initialized with corresponding value and other indices will be initialized to 0 by default.

Ex: int arr[5] = {2,3,4,6,5}; int arr2[5] = {1,2,3};

Accessing Array elements :

In C++, elements of array can be accessed by specifying array name followed by the element index, enclosed in brackets. Index should according to zero indexing.

Examples : arr[0] = 2; arr[4]=3;

Multidimensional Arrays:

You can make multidimensional arrays, or "arrays of arrays". Multidimensional arrays can be created by adding an extra set of brackets and array lengths for every additional dimension you want your array to have.

Ex: int array1[2][4] = { {1,2,3,4} , {3,5,2,3} };

int array2[3][4] = { {1,2,3,4} , {3,5,2,3}, {4,5,2,3} };

Accessing : array2[1][3] = 3;

Arrays as Strings:

An Array of characters can be used to hold a string. The array may be built of either signed or unsigned characters. During declaration by specifying size of character array size we can specify maximum number of characters that should be in string including the null character.

Ex: char color[26]; char country[10] = { 'I' , 'N' , 'D' , 'I' , 'A', '\0' };

char my_country[26] = "INDIA";

Arrays of Structures : You can create an array of a structure type just as you can any array of primitive data type.

```
struct point
    {
        Int x ,y;
      } ;
```

Ex: struct point arr[3] = { {1, 2} , {3, 7}, {5, 7} };

Accessing : arr[0].x = 1; , arr[1].y = 7;

8. Function

A function is a set of statements that take inputs, do some specific computation, and produces output. There are only two things that can be done with a function, call it or take its address. If the name of a function appears in an expression not in the function name position of call, a pointer to the function is generated. We must declare function before first use of the function.

The general form of a function is:

return_type function_name (arg1_type arg1_name, arg2_type arg2_name, ….) {

function-body}

Ex: A function that takes two integers as parameters and returns an integer.

int max (int x, int y) {….}

Calling functions – call a function by its name and providing needed parameters. General form of function call:

function-name (parameters)

Ex: int a = max (5,6);

Recursive functions

We can also call another function inside a function. These functions are known as recursive function.

Ex:

```
  int factorial (int x) {
                if (x < 1)
                 return 1;
 else
   return (x * factorial (x - 1));
}
```

9. Pointers

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address.

There are a few important operations, which we will do with the help of pointers very frequently.

● We define a pointer variable

● assign the address of a variable to a pointer and

● finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Declaration:

The obvious way to declare two pointer variables in a single declaration is :

● If the type of a variable containing a pointer to **int** is **int ***,

● and a single declaration can declare multiple variables of the same type by simply providing a comma-separated list (ptr_a, ptr_b),

● then you can declare multiple **int**-pointer variables by simply giving the **int**-pointer type (**int *)** followed by a comma-separated list of names to use for the variables (ptr_a, ptr_b).

Ex: int* ptr_a, ptr_b;

NULL Pointers: A pointer that is assigned NULL is called a null pointer. The NULL pointer is a constant with a value of zero defined in several standard libraries.

Ex: int* ptr = NULL;

Pointers have many and easy concepts and they are very important.

1. Pointer Arithmetic: There are four arithmetic operators that can be used in pointers:

++, **--**, **+**, **-**.

2. Array of Pointers: We can define arrays to hold several pointers.

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

4. Passing pointers to functions: Passing an argument by reference or by address enables the passed argument to be changed in the calling function by the called function.

5. Return pointer from functions: C++ allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

10. User Defined Data Types

The UDT (User-Defined Data Type) is a typical data type that we can derive out of any existing data type in a program. We can utilize them for extending those built-in types that are already available in a program, and then you can create various customized data types of your own.

Class: The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Syntax:

```
class className
{     Access Specifier:          //can be private, public or protected
      Data Members; //Variables to be used
      Member Functions() //Methods to access data members
 }; //Class names ends with a semicolon
```

Structure: A structure is a user defined data type in C++. A structure creates a data type that can be used to group items of possibly different types into a single type.

Syntax:

```
struct address {
        char name[50];
        char street[100];
        char city[50];
        char state[20];
        int pin;   };
```

Accessing: struct address one; one.name; one.city;