
MODULE - IV

Pointers and Virtual Functions

MODULE 4

Pointers and Virtual Functions

Module Description

The main goal of studying Pointers in C++ is to understand how to use pointers. This module explains the Pointers and arrays, also Memory management in C++.

By the end of this module, students will learn how to declare and access pointer variables, Also to write some sample programs using pointers. In addition to these skills students are also able to intelligently compile and execute the program and debug the errors.

Chapter 4.1

Pointer

Chapter 4.2

Virtual Functions

Chapter Table of Contents

Chapter 4.1

Pointer

| | |
|---|-----|
| Aim..... | 199 |
| Instructional Objectives..... | 199 |
| Learning Outcomes..... | 199 |
| 4.1.1 Introduction..... | 200 |
| 4.1.2 Pointer | 200 |
| (i) Pointer Declaration and Access..... | 200 |
| (ii) Pointer and Arrays..... | 203 |
| (iii) Pointer and Functions..... | 206 |
| (iv) Array of Pointers to Strings | 207 |
| Self-assessment Questions..... | 209 |
| 4.1.3 Memory Management | 209 |
| (i) New and Delete Operators..... | 209 |
| (ii) Pointer to Objects..... | 210 |
| (iii) Self-Containing Classes..... | 212 |
| (iv) Sorting Pointers | 213 |
| Self-assessment Questions..... | 215 |
| Summary | 216 |
| Terminal Questions..... | 216 |
| Answer Keys..... | 217 |
| Activity..... | 217 |
| Bibliography..... | 218 |
| e-References | 218 |
| External Resources | 218 |
| Video Links | 219 |



Aim

To equip the students with skills needed to write programs using pointers in C++ programming



Instructional Objectives

After completing this chapter, you should be able to:

- Explain pointers
- Describe pointer constants and pointer variables
- Explain pointers and functions
- Illustrate pointer to object by referring to members using pointers
- Describe memory management in C++



Learning Outcomes

At the end of this chapter, you are expected to:

- Discuss pointer declaration and access
- Design a program to demonstrate pointer concepts
- Write a code to demonstrate call by pointer array and array of pointers to string
- Identify new and delete operator in C++

4.1.1 Introduction

Pointers are a very important feature of C++ language which has been utilized in many lower level programming languages. These are variable that stores the location of another variable. These variables are locations in the PC's memory which can be accessed by their identifier (their name). Thus, the program does not have to think about the physical location of the information in memory; it basically utilizes the identifier at whatever point it needs to refer to the variable in the program.

The following sections of this chapter will discuss about pointers and how these pointers are declared and can be accessed in C++ programs. It will further discuss the different types of pointers and its use in C++ programming and also how pointers can be used in memory management.

4.1.2 Pointer

A pointer is a derived data type that stores the memory address of another data variable. Instead of defining actual data, a pointer variable defines where to get the value of a specific data variable.

Like C, a pointer variable can also refer to another pointer in C++. However, it often points to a data variable. Pointers provide an alternative approach to access other data objects.

(i) Pointer Declaration and Access

We can declare a pointer variable like different variables in C++. Here, the declaration depends on the data type of the variable it points to. The declaration of a pointer variable has the following structure:

```
data-type *pointer-variable;
```

Here data-type refers to one of the legitimate C++ data types, *for example*, int, char, float, etc. and the pointer-variable is the name of the pointer variable. The * symbol recognizes a pointer variable from different variables to the compiler.

A pointer can point to any data available in C++. Let us declare a pointer variable, which points to an integer variable, given below:

```
int * ptr;
```

Here, ptr is a pointer variable which points to an integer data type and contains the memory location of any integer variable. Similarly, we can declare pointer variables for other data types also.

A variable must be initialised before using it in a C++ program. We can initialise a pointer variable as follows.

```
Int *ptr, a;          //declaration
ptr = &a;             //initialization
```

The pointer variable ptr contains the location of the variable 'a'. The (*) is a deference operator as seen in C, to recover the location of a variable, we utilize 'location of operator or reference operator. The second statement assigns the location of the variable "a" to the pointer 'ptr'.

We can also declare a pointer variable to point to another pointer. Below is an example program to illustrate the use of pointers.

Program:

```
1  #include <iostream>
2  using namespace std;
3  int main ()
4  {
5      int ptr = 20;    // actual variable declaration.
6      int *p;          // pointer variable
7      p = &ptr;        // store address of var in pointer variable
8      cout << "Value of var variable: ";
9      cout << ptr << endl;
10     // print the address stored in ip pointer variable
11     cout << "Address stored in p variable: ";
12     cout << p << endl;
13     // access the value at the address available in pointer
14     cout << "Value of *p variable: ";
15     cout << *p << endl;
16     return 0;
17 }
```

Output:

```
Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Value of var variable: 20
Address stored in p variable: 0x7ffc4300c314
Value of *p variable: 20
sh-4.3$
```

The memory location is always addressed by the operating system. The output may vary depending on the system.

Pointer to Void

We can also use void pointers, known as generic pointers, which refer to variables of any data type. Before using void pointers, we must type cast the variables to the specific data types that they point to.

The void type of pointer is a special type of pointer. In C++, void represents the absence of type. Therefore, void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).

This gives void pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters. In exchange, they have a great limitation: the data pointed to them cannot be directly dereferenced (which is logical, since we have no type to dereference to) and for that reason, any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

One of its possible uses may be to pass generic parameters to a function. *For example,*

```
1  #include <iostream>
2  using namespace std;
3  void increase (void* data, int psize)
4  {
5      if ( psize == sizeof(char) )
6      { char* pchar; pchar=(char*)data; ++(*pchar); }
7      else if (psize == sizeof(int) )
8      { int* pint; pint=(int*)data; ++(*pint); }
9  }
10 int main ()
11 {
12     char a = 'x';
13     int b = 1602;
14     increase (&a,sizeof(a));
15     increase (&b,sizeof(b));
16     cout << a << ", " << b << "\n";
17     return 0;
18 }
```

Output:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
y, 1603
sh-4.3$
```

sizeof is an operator integrated in the C++ language is that it returns the size in bytes of its argument. For non-dynamic data types, this value is a constant. Therefore, *for example*, sizeof (char) is 1, because char has always a size of one byte.

The pointers which are not initialized in a program are called NULL pointers. Pointers of any data type can be assigned with one value *i.e.*, '0' called null address.

(ii) Pointer and Arrays

The array of pointers represents a group of locations. By declaring array of pointers, we can save a generous amount of memory space. We can declare an array of pointers as follows:

```
data-type *arrayname[array-size];
```

An array of pointers points to an array of data items. Every component of the pointer array focuses to an item of the data array. Data items can be accessed either directly or by dereferencing the components of pointer array. We can revamp the pointer components without touching the data items.

For example, consider these two declarations:

```
int arr[20];  
int * a;
```

Let's see an example that mixes arrays and pointers:

```
1  #include <iostream>  
2  using namespace std;  
3  int main ()  
4  {  
5      int num[5];  
6      int * ptr;  
7      ptr = num;  
8      *ptr = 100;  
9      ptr++;  
10     *ptr = 200;  
11     ptr = &num[2];  
12     *ptr = 300;  
13     ptr = num + 3;  
14     *ptr = 400;  
15     ptr = num;  
16     *(ptr+4) = 500;  
17     for (int n=0; n<5; n++)  
18         cout << num[n] << " , ";  
19     return 0;  
20 }
```

Output:

```
Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
100, 200, 300, 400, 500, sh-4.3$
```

• Pointer Constants and Pointer Variables

Pointers can be utilized to get to a variable by its location and this may include changing the value it is pointing to. Additionally, it can also be used to read the value pointed by the pointer without changing it. For this, it is sufficient with qualifying the type indicated by the pointer as `const`.

For example,

```
int a;
int b = 10;
const int * p = &b;
a = *p;           // ok: reading p
*p = a;           // error: modifying p, which is const-qualified
```

Here `p` points to a variable which is declared as `const` so that it can read the value pointed, but it cannot modify it. One of the use cases of pointers to `const` elements is as function parameters:

Program:

```
1  #include <iostream>
2  using namespace std;
3
4  void incr_all (int* start, int* stop)
5  {
6      int * curr = start;
7      while (curr != stop) {
8          ++(*curr); // increment value pointed
9          ++curr;    // increment pointer
10     }
11 }
12 void print_all (const int* start, const int* stop)
13 {
14     const int * curr = start;
15     while (curr != stop) {
16         cout << *curr << '\n';
17         ++curr;    // increment pointer
18     }
19 }
20
```

```
21 int main ()
22 {
23     int num_arr[] = {5,10,15};
24     incr_all (num_arr,num_arr+3);
25     print_all (num_arr,num_arr+3);
26     return 0;
27 }
```

Output:

```
Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
6
11
16
sh-4.3$
```

Note that `print_all` utilizes pointers that point to constant components. These pointers point to constants which cannot be altered, however they are not constant themselves: *i.e.*, the pointers can in any case be increased or can be assigned various locations, despite the fact that they can't alter variable it points to.

And this is where a second dimension to constness is added to pointers: Pointers can also be themselves `const`. And this is specified by appending `const` to the pointed type (after the asterisk):

```
int p;
int *      ptr1 = &p; // non-const pointer to non-const int
const int * ptr2 = &p; // non-const pointer to const int
int * const ptr3 = &p; // const pointer to non-const int
const int * const ptr4 = &p; // const pointer to const int
```

The syntax with `const` and pointers is certainly precarious and perceiving the cases that best suit every utilisation has a tendency to require some experience. Regardless, it is imperative to get constness with pointers (and references) right within the near future, but you should not stress a lot about knowing everything if this is the first occasion when you are presented to the mix of `const` and pointers.

To add a little bit more confusion to the syntax of `const` with pointers, the `const` qualifier can either precede or follow the pointed type, with the exact same meaning:

```
const int * ptr2 = &p; // non-const pointer to const int
int const * ptr2 = &p; // also non-const pointer to const int
```

As with the spaces surrounding the asterisk, the order of `const` in this case is simply a matter of style. This chapter uses a prefix `const`, as for historical reasons this seems to be more extended, but both are exactly equivalent. The merits of each style are still intensely debated on the internet.

(iii) Pointer and Functions

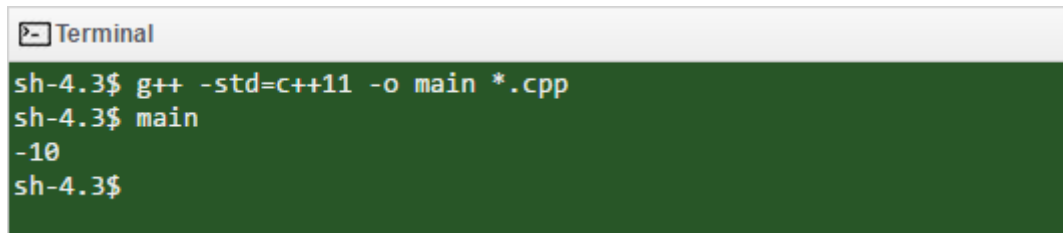
C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function.

Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

Program:

```
1  #include <iostream>
2  using namespace std;
3
4  int add (int x, int y)
5  { return (x+y); }
6
7  int sub (int x, int y)
8  { return (x-y); }
9  int op (int a, int b, int (*fun_call)(int,int))
10 {
11     int z;
12     z = (*fun_call)(a,b);
13     return (z);
14 }
15 int main ()
16 {
17     int p,q;
18     int (*minus)(int,int) = sub;
19
20     p = op (15, 15, add);
21     q = op (20, p, minus);
22     cout << q << endl;
23     return 0;
24 }
```

Output:



```
Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
-10
sh-4.3$
```

In the example above, minus is a pointer to a function that has two parameters of type int. It is directly initialized to point to the function subtraction:

```
int (* minus)(int,int) = subtraction;
```

- **Call by Pointer Arrays**

Pointers may be arrayed like any other data type. The declaration of an int pointer array of size 10 is `int * x[10];`

If we want to pass an array of pointers in to a function, we can utilize the same strategy that we use to pass different arrays *i.e.*, basically call the function with the array name without any indexes.

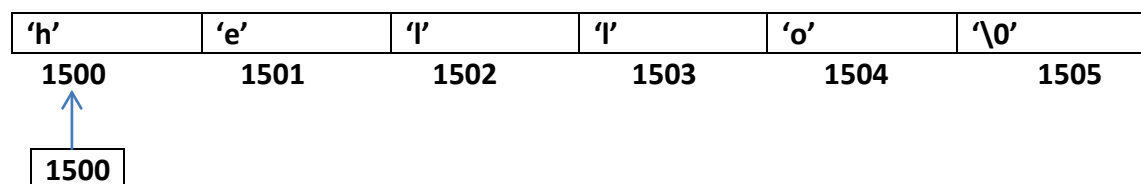
For example, a function that can receive array x looks like this

```
Void display_array(int * q[])
{
    Int t;
    For(t=0;t<10;t++)
        Printf("%d",*q[t]);
}
```

(iv) Array of Pointers to Strings

A string is a one dimensional array of characters, which start with the index 0 and ends with a null character '\0'. A pointer variable can access a string by referring to its first character. There are two ways to assign a value to a string. We can use the character array or variable of type `char *` as follows.

```
char ptr[] = "hello";
const char *ptr = "hello";
```



ptr

The declaration is done as shown above, the first declaration creates an array of four characters, which contains 'h','e','l','l','o','\0', but the second declaration generates a pointer variable, which points to the first character, *i.e.*, 'h' of the string. In C++, there are numerous

string handling functions available and are available in the header file <cstring>. The program given below illustrates how to access strings using pointers and arrays.

Program:

```
#include <iostream>
using namespace std;
const int SIZE = 4;
int main ()
{
    char *names[SIZE] =
    {
        "Ramya",
        "Rahul",
        "Nilam",
        "Sara",
    };
    for (int i = 0; i < SIZE; i++)
    {
        cout << "Value of names[" << i << "] = ";
        cout << names[i] << endl;
    }
    return 0;
}
```

Output:

Value of names[0] = Ramya

Value of names[1] = Rahul

Value of names[2] = Nilam

Value of names[3] = Sara



Self-assessment Questions

- 1) The _____ operator informs the compiler that the variable is a pointer variable.
- 2) A pointer to a pointer stores _____ of another _____ variable.
 - a) Address, Pointer
 - b) Value, Pointer
 - c) Allocates Memory, Pointer
 - d) Allocates memory, Address
- 3) The & operator retrieves the lvalue of the variable.
 - a) True
 - b) False
- 4) The operator “*” signifies a.
 - a) Referencing Operator
 - b) Dereferencing Operator
 - c) Address operator
 - d) Star Operator
- 5) While declaring pointer variables, which operator do we use?
 - a) Address
 - b) Arrow
 - c) Indirection
 - d) Dot

4.1.3 Memory Management

We utilize dynamic allocation techniques when it is not known ahead of time how much memory space is required. In spite of the fact that C++ supports these functions, it additionally characterizes two unary operators new and delete.

(i) New and Delete Operators

New and delete operators are unary administrator that plays out the undertaking of allocating and releasing the memory in a superior and simpler way. Since these operators control memory on the store, they are otherwise called store operators.

An object can be created by using new and destroyed by using delete, as and when required. An information object made inside a block with new, will stay in presence until it is unequivocally destroyed by using delete. Along these lines, the lifetime of an object is specifically under our control and is disconnected to the class structure of the program.

The new operator can be used to create objects of any data type.

The syntax is:

```
Pointer-variable = new data-type;
```

Here, pointer-variable is a pointer of type data-type. The new operator assigns adequate memory to hold a data object of type data-type and returns the location of the object. The data-type may be any valid data-type. The pointer-variable holds the location of the memory space assigned to it.

For example,

```
x = new float;  
y = new int;
```

In the above example, x is a pointer of type float and y is a pointer of type int. Here, x and y must have already been declared as pointers of appropriate types. Alternatively we can combine the declaration of pointers and their assignments as follows:

```
float *x = new float;  
Int * y = new int;
```

Subsequently, the below statements assign 1.5 to the newly created float object and 10 to the int object.

```
*x = 1.5;  
*y = 10;
```

(ii) Pointer to Objects

Pointers can be used to point any object that is created for a class.

Let us consider the following example:

```
fruit mango;
```

Here fruit is a class and mango is an object created for class of type fruit. In the same way, we can also define a pointer imango that points to the class of type fruit as given below:

```
fruit *imango;
```

These are called as object pointers which points to objects created to any class. Objects pointers are further used for creation of objects during run time of your program.

Referring to members using pointers

We can create pointers for class members such as to its variables and functions, similar for our normal functions and variables that we create for other programs.

Pointer to Data Members of class

Also we can create pointers for data member variables of a class. Consider the following syntax:

```
datatype class_name :: *pointer_name ;
```

We can use assignment statement for assigning pointer name to the class and its data member by following the below syntax:

```
pointer_name = &class_name :: datamember_name ;
```

Further, we can then combine the above two syntax and perform the same operation with just a single statement given by the following syntax:

```
datatype class_name::*pointer_name = &class_name::datamember_name ;
```

The following program illustrates the use of pointer to member variables.

```
1  class Variable
2  {
3  public:
4  int x;
5  void print() { cout << "x is=" << x; }
6  };
7
8  int main()
9  {
10 Variable v, *vp;
11 vp = &v;      // pointer to object
12
13 int Variable::*ptr=&Variable::x;  // pointer to data member 'x'
14
15 v.*ptr=25;
16 v.print();
17
18 vp->*ptr=50;
19 vp->print();
20 }
```

Output:

| | |
|---|----------------|
| 1 | x is=25x is=50 |
|---|----------------|

The concept of pointers to objects is complicated; hence it is used only for special cases of programming.

Pointer to Member Functions

We can also create pointers to member functions of a class. Consider the following syntax:

```
return_type (class_name::*ptr_name) (argument_type) =
    &class_name::function_name ;
```

The following program illustrates the use pointer to member functions.

```
class Variable
{ public:
    int x (float) { return 1; }
};
int (Variable::*fx1) (float) = &Variable::x;    // Declaration and
assignment
int (Variable::*fx2) (float);                  // Only Declaration
int main()
{
    fx2 = &Variable::x;    // Assignment inside main()
}
```

(iii) Self-Containing Classes

All member functions defined within class definition is called self-contained classes. The following & program illustrate this concept.

```
class Self
{
    private:
        double a, b, c;
        double length() const { return sqrt(a*a + b*b + c*c); }
    public:
        Self(double x = 0.0, double y = 0.0, double z = 0.0)
        {
            a = x; b = y; c = z; return;
        }
        ~Self() {return; }
};
class Self
{
    private: ... Data and member functions (from the above class)
```

```

public:
... Constructor and destructor (from the above class) ...
Self operator+ (Self const &p) const
{
    return Self(a + p.a, b + p.b, c + p.c);
}
Self operator* (double const factor) const
{
    return Self(a*factor, b*factor, c*factor);
}
};

```

(iv) Sorting Pointers

Sorting is arranging of elements in ascending or descending order. Sorting pointers is sorting of pointer data type in ascending or descending order.

The following example shows the sorting of array elements using pointers.

```

1  #include <iostream>
2  using namespace std;
3  void bubble(int*,int);//////////prototype
4
5  int main()
6  {
7      int Array[5],size=5,*pointer;
8
9      for(int num=0;num<size;num++)
10     {
11         cout<<"Enter element no : "<<num+1<<endl;
12         cin>>Array[num];
13     }
14     pointer=Array;
15     bubble(pointer,size);
16 }
17
18 void bubble(int *pointer,int size)
19 {
20     int count1,count2,swap;
21
22     for(count1=0;count1<size-1;count1++)
23     {
24         for(count2=0;count2<size-count1-1;count2++)
25         {
26
27             if(*(pointer+count2)>*(pointer+count2+1))
28             {
29                 swap=*(pointer+count2);
30                 *(pointer+count2)=*(pointer+count2+1);
31                 *(pointer+count2+1)=swap;
32             }
33         }
34     }
35 }
36

```

```
37         cout << "Array after sorting is " << endl;
38         for(count1=0; count1<size; count1++)
39         {
40             cout << *(pointer+count1) << " , ";
41         }
42     }
```

Output:

```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Enter element no : 1
100
Enter element no : 2
10
Enter element no : 3
200
Enter element no : 4
50
Enter element no : 5
20
Array after sorting is
10 , 20 , 50 , 100 , 200 , sh-4.3$
```



Did you Know?

C++ allows pointers to perform arithmetic operations like increment (++) or decrement (--) pointers and added or subtracted from a pointer and one pointer can be subtracted from another. Pointer arithmetic that moves a pointer outside of an array is undefined even without dereferencing.



Self-assessment Questions

- 6) Dynamically allocated memory can be referred using _____.
- 7) _____ function returns memory to the heap.
- a) New
 - b) Delete
 - c) Main
 - d) Library
- 8) The _____ operator calculates the size of the object.
- a) New
 - b) Delete
 - c) Dot
 - d) Arithmetic



Summary

- Pointers are nothing but memory addresses.
- A pointer is a variable that contains the address of another variable.
- The address of the memory location is a pointer constant, therefore it cannot be changed.
- Object pointers are useful in creating objects at run time.
- Object pointers can be used to access the public members of an object, along with an arrow operator.
- Pointers to objects of a base class type are compatible with pointers of objects of a derived class. Therefore, we can use a single pointer variable to point to objects of base class as well as derived classes.



Terminal Questions

1. Explain pointers, pointer constants and pointer variables.
2. Explain pointers and functions.
3. Explain array of pointers to strings.
4. Illustrate pointer to object by referring to members using pointers.
5. Describe memory management in C++.



Answer Keys

| Self-assessment Questions | |
|---------------------------|----------|
| Question No. | Answer |
| 1 | * |
| 2 | a |
| 3 | a |
| 4 | b |
| 5 | a |
| 6 | pointers |
| 7 | b |
| 8 | a |



Activity

1. Activity Type: Online

Duration: 30 Minutes

Description:

1. Design and execute a program which prompts the user to enter a string and returns the length of the longest sequence of identical consecutive characters within the string using pointers to data members and member function. *For example*, in the string "aaaAAAAAjjB"
2. ", the longest sequence of identical consecutive characters is "AAAAA".

Bibliography



e-References

- cplusplus.com,(2016). Pointers - C++ Tutorials. Retrieved 7 April, 2016. from, <http://www.cplusplus.com/doc/tutorial/pointers/>
- exforsys.com,(2016). C++ Memory Management Operators .Retrieved 7 April, 2016. from, <http://www.exforsys.com/tutorials/c-plus-plus/c-plus-plus-memory-management-operators.html>
- cprogramming.com,(2016). Pointers in C and C++ - Tutorial .Retrieved 7 April, 2016. from, <http://www.cprogramming.com/tutorial/lesson6.html>
- tutorialspoint.com,(2016). C++ Pointers. Retrieved 7 April, 2016. from, http://www.tutorialspoint.com/cplusplus/cpp_pointers.htm



External Resources

- Balaguruswamy, E. (2008). Object Oriented Programming with C++. Tata McGraw Hill Publications.
- Lippman. (2006). C++ Primer (3rd ed.). Pearson Education.
- Robert, L. (2006). Object Oriented Programming in C++ (3rd ed.). Galgotia Publications References.
- Schildt, H., & Kanetkar, Y. (2010). C++ completer (1st ed.). Tata McGraw Hill.
- Strousstrup. (2005). The C++ Programming Language (3rd ed.). Pearson Publications



Video Links

| Topic | Link |
|---------------------------------|---|
| Pointers and Arrays | https://www.youtube.com/watch?v=qRJUPN0NWRU |
| Pointers and Functions | https://www.youtube.com/watch?v=ne2VrFRUfVw |
| New and Delete Operators in C++ | https://www.youtube.com/watch?v=gVGnOsB1n_o |
| Memory Management in C++ | https://www.youtube.com/watch?v=5Hi3KN-0RdY&list=PLtNErhYMkHnFAxgFui0MWQwa3R-V1tqUB |



Notes:

