Lab 10

Objectives

After performing this lab, students will be able to

- Understand pointers and use them in programs.

Pointers

In C++, pointers are variables that store the memory addresses of other variables.

Address in C++

If we have a variable var in our program, &var will give us its address in the memory. For example,

Example 1: Printing Variable Addresses in C++

```
#include <iostream>
using namespace std;
int main()
{
   // declare variables
   int var1 = 3;
   int var2 = 24;
   int var3 = 17;

   // print address of var1
   cout << "Address of var1: "<< &var1 << endl;

   // print address of var2
   cout << "Address of var2: " << &var2 << endl;

   // print address of var3
   cout << "Address of var3: " << &var3 << endl;
}
```

Output

```
Address of var1: 0x7fff5fbff8ac
Address of var2: 0x7fff5fbff8a8
Address of var3: 0x7fff5fbff8a4
```

Here, 0x at the beginning represents the address is in the hexadecimal form.

Notice that the first address differs from the second by 4 bytes and the second address differs from the third by 4 bytes.

C++ Pointers
As mentioned above, pointers are used to store addresses rather than values.

Here is how we can declare pointers.
int *pointVar;

Here, we have declared a pointer pointVar of the int type.
We can also declare pointers in the following way.
int* pointVar; // preferred syntax

Let's take another example of declaring pointers.
int* pointVar, p;

Here, we have declared a pointer pointVar and a normal variable p.
Note: The * operator is used after the data type to declare pointers.

Assigning Addresses to Pointers

Here is how we can assign addresses to pointers:

```
int* pointVar, var;
var = 5;

// assign address of var to pointVar pointer
pointVar = &var;
```

Here, 5 is assigned to the variable var. And, the address of var is assigned to the pointVar pointer with the code pointVar = &var.

Get the Value from the Address Using Pointers
To get the value pointed by a pointer, we use the * operator. For example:

```
int* pointVar, var;
var = 5;

// assign address of var to pointVar
pointVar = &var;

// access value pointed by pointVar
cout << *pointVar << endl;   // Output: 5
```

In the above code, the address of var is assigned to pointVar. We have used the *pointVar to get the value stored in that address.
When * is used with pointers, it's called the dereference operator. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, *pointVar = var.
Note: In C++, pointVar and *pointVar is completely different. We cannot do something like *pointVar = &var;

Working of C++ Pointers:

Example 2: Working of C++ Pointers

```
#include <iostream>
using namespace std;
int main() {
   int var = 5;

   // declare pointer variable
   int* pointVar;

   // store address of var
   pointVar = &var;

   // print value of var
   cout << "var = " << var << endl;

   // print address of var
   cout << "Address of var (&var) = " << &var << endl << endl;

   // print pointer pointVar
   cout << "pointVar = " << pointVar << endl;

   // print the content of the address pointVar points to
   cout << "Content of the address pointed to by pointVar (*pointVar) = " << *pointVar << endl;

   return 0;
}
```

Output

```
var = 5
Address of var (&var) = 0x61ff08

pointVar = 0x61ff08
Content of the address pointed to by pointVar (*pointVar) = 5
```

Changing Value Pointed by Pointers
If pointVar points to the address of var, we can change the value of var by using *pointVar.

For example,

```
int var = 5;
int* pointVar;

// assign address of var
pointVar = &var;

// change value at address pointVar
*pointVar = 1;

cout << var << endl; // Output: 1
```

Here, pointVar and &var have the same address, the value of var will also be changed when *pointVar is changed.

Example 3: Changing Value Pointed by Pointers

```
#include <iostream>
using namespace std;
int main() {
   int var = 5;
   int* pointVar;

   // store address of var
   pointVar = &var;

   // print var
   cout << "var = " << var << endl;

   // print *pointVar
   cout << "*pointVar = " << *pointVar << endl
      << endl;

   cout << "Changing value of var to 7:" << endl;

   // change value of var to 7
   var = 7;

   // print var
   cout << "var = " << var << endl;

   // print *pointVar
   cout << "*pointVar = " << *pointVar << endl << endl;

   cout << "Changing value of *pointVar to 16:" << endl;
```

```
    // change value of var to 16
    *pointVar = 16;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl;
    return 0;
}
```

Output

```
var = 5
*pointVar = 5

Changing value of var to 7:
var = 7
*pointVar = 7

Changing value of *pointVar to 16:
var = 16
*pointVar = 16
```

Common mistakes when working with pointers
Suppose, we want a pointer varPoint to point to the address of var. Then,

```
int var, *varPoint;

// Wrong!
// varPoint is an address but var is not
varPoint = var;

// Wrong!
// &var is an address
// *varPoint is the value stored in &var
*varPoint = &var;

// Correct!
// varPoint is an address and so is &var
varPoint = &var;

 // Correct!
// both *varPoint and var are values
*varPoint = var;
```

C++ Pointers and Arrays

In C++, Pointers are variables that hold addresses of other variables. Not only can a pointer store the address of a single variable, it can also store the address of cells of an array.

Consider this example:

```
int *ptr;
int arr[5];

// store the address of the first
// element of arr in ptr
ptr = arr;
```

Here, ptr is a pointer variable while arr is an int array. The code ptr = arr; stores the address of the first element of the array in variable ptr.

Notice that we have used arr instead of &arr[0]. This is because both are the same. So, the code below is the same as the code above.

```
int *ptr;
int arr[5];
ptr = &arr[0];
```

The addresses for the rest of the array elements are given by &arr[1], &arr[2], &arr[3], and &arr[4].

Point to Every Array Elements

Suppose we need to point to the fourth element of the array using the same pointer ptr.

Here, if ptr points to the first element in the above example then ptr + 3 will point to the fourth element. For example,

```
int *ptr;
int arr[5];
ptr = arr;

ptr + 1 is equivalent to &arr[1];
ptr + 2 is equivalent to &arr[2];
ptr + 3 is equivalent to &arr[3];
ptr + 4 is equivalent to &arr[4];
```

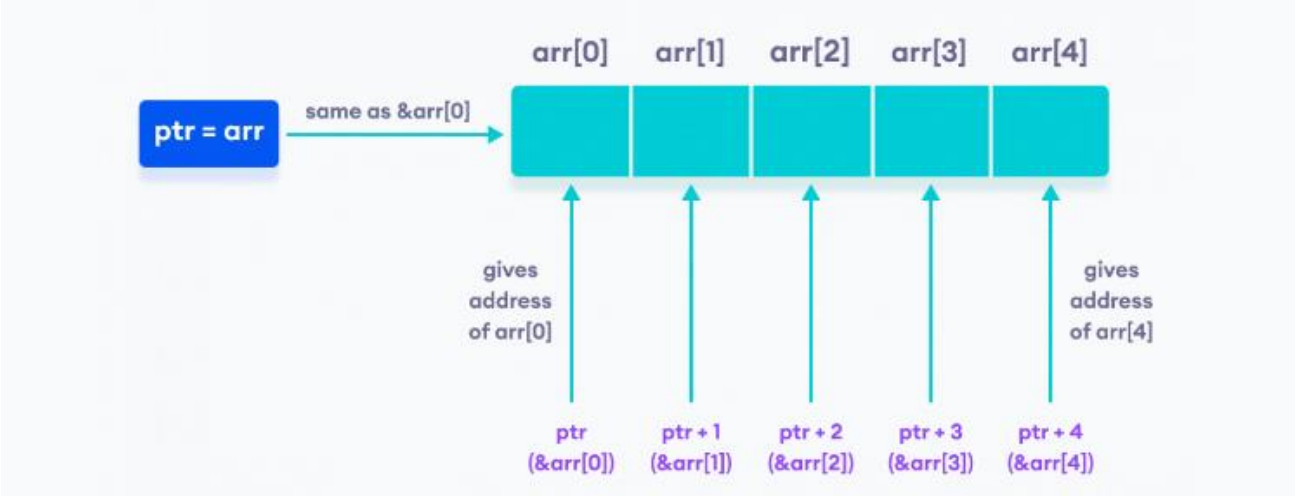
Similarly, we can access the elements using the single pointer. For example,

```
// use dereference operator
*ptr = arr[0];
*(ptr + 1) is equivalent to arr[1];
*(ptr + 2) is equivalent to arr[2];
*(ptr + 3) is equivalent to arr[3];
*(ptr + 4) is equivalent to arr[4];
```

Suppose if we have initialized ptr = &arr[2]; then

ptr - 2 is equivalent to &arr[0];
ptr - 1 is equivalent to &arr[1];
ptr + 1 is equivalent to &arr[3];
ptr + 2 is equivalent to &arr[4];

Working of C++ Pointers and Arrays



Note: The address between ptr and ptr + 1 differs by 4 bytes. It is because ptr is a pointer to an int data. And, the size of int is 4 bytes in a 64-bit operating system.

Similarly, if pointer ptr is pointing to char type data, then the address between ptr and ptr + 1 is 1 byte. It is because the size of a character is 1 byte.

Example 1: C++ Pointers and Arrays

```
// C++ Program to display address of each element of an array

#include <iostream>
using namespace std;

int main()
{
   float arr[3];

   // declare pointer variable
   float *ptr;

   cout << "Displaying address using arrays: " << endl;

   // use for loop to print addresses of all array elements
   for (int i = 0; i < 3; ++i)
   {
```

```
    cout << "&arr[" << i << "] = " << &arr[i] << endl;
  }

  // ptr = &arr[0]
  ptr = arr;

  cout<<"\nDisplaying address using pointers: "<< endl;

  // use for loop to print addresses of all array elements
  // using pointer notation
  for (int i = 0; i < 3; ++i)
  {
    cout << "ptr + " << i << " = "<< ptr + i << endl;
  }

  return 0;
}
```

Output

```
Displaying address using arrays:
&arr[0] = 0x61fef0
&arr[1] = 0x61fef4
&arr[2] = 0x61fef8

Displaying address using pointers:
ptr + 0 = 0x61fef0
ptr + 1 = 0x61fef4
ptr + 2 = 0x61fef8
```

In the above program, we first simply printed the addresses of the array elements without using the pointer variable ptr.

Then, we used the pointer ptr to point to the address of a[0], ptr + 1 to point to the address of a[1], and so on.

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why we can use pointers to access elements of arrays.

However, we should remember that pointers and arrays are not the same.

Example 2: Array name used as pointer

```
// C++ Program to insert and display data entered by using pointer notation.

#include <iostream>
using namespace std;

int main() {
   float arr[5];

  // Insert data using pointer notation
   cout << "Enter 5 numbers: ";
   for (int i = 0; i < 5; ++i) {

      // store input number in arr[i]
      cin >> *(arr + i) ;

   }

   // Display data using pointer notation
   cout << "Displaying data: " << endl;
   for (int i = 0; i < 5; ++i) {

      // display value of arr[i]
      cout << *(arr + i) << endl ;

   }

   return 0;
}
```

Output

```
Enter 5 numbers: 2.5
3.5
4.5
5
2
Displaying data:
2.5
3.5
4.5
5
2
```

Here,
We first used the pointer notation to store the numbers entered by the user into the array arr.

cin >> *(arr + i) ;
This code is equivalent to the code below:

cin >> arr[i];

Notice that we haven't declared a separate pointer variable, but rather we are using the array name arr for the pointer notation.

As we already know, the array name arr points to the first element of the array. So, we can think of arr as acting like a pointer.

Similarly, we then used for loop to display the values of arr using pointer notation.
cout << *(arr + i) << endl ;

This code is equivalent to
cout << arr[i] << endl ;

C++ Call by Reference: Using pointers
In the C++ Functions tutorial, we learned about passing arguments to a function. This method used is called passing by value because the actual value is passed.

However, there is another way of passing arguments to a function where the actual values of arguments are not passed. Instead, the reference to values is passed.

For example,

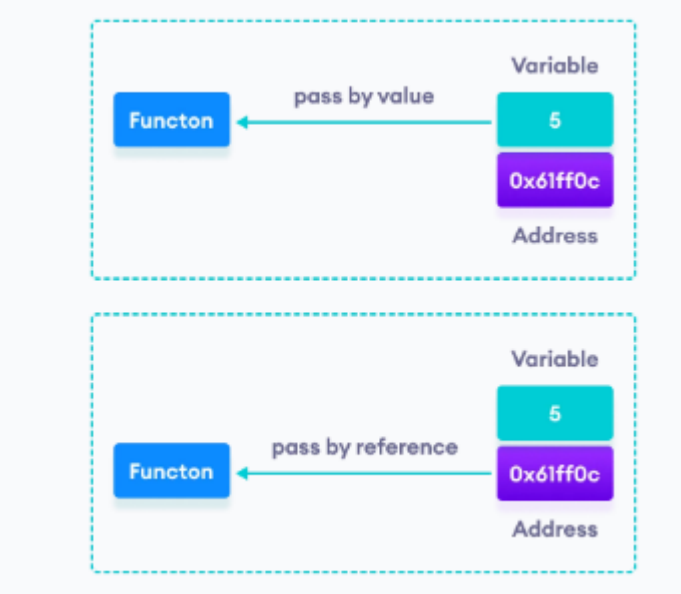
```
// function that takes value as parameter

void func1(int numVal) {
   // code
}

// function that takes reference as parameter
// notice the & before the parameter
void func2(int &numRef) {
   // code
}

int main() {
   int num = 5;

   // pass by value
   func1(num);

   // pass by reference
   func2(num);
```

```
    return 0;
}
```
Notice the & in void func2(int &numRef). This denotes that we are using the address of the variable as our parameter.

So, when we call the func2() function in main() by passing the variable num as an argument, we are actually passing the address of num variable instead of the value 5.

Passing value vs. passing reference as function argument in C++

C++ Pass by Value vs. Pass by Reference



Example 1: Passing by reference without pointers

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

// function definition to swap values
void swap(int &n1, int &n2) {
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}

int main() {

    // initialize variables
    int a = 1, b = 2;

    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
```

```
    // call function to swap numbers
    swap(a, b);

    cout << "\nAfter swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    return 0;
}
```

Output

```
Before swapping
a = 1
b = 2

After swapping
a = 2
b = 1
```

In this program, we passed the variables a and b to the swap() function. Notice the function definition,

```
void swap(int &n1, int &n2)
```

Here, we are using & to denote that the function will accept addresses as its parameters.

Hence, the compiler can identify that instead of actual values, the reference of the variables is passed to function parameters.

In the swap() function, the function parameters n1 and n2 are pointing to the same value as the variables a and b respectively. Hence the swapping takes place on actual value.

The same task can be done using the pointers. To learn about pointers, visit C++ Pointers.

Example 2: Passing by reference using pointers

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

// function prototype with pointers as parameters
void swap(int*, int*);

int main() {

    // initialize variables
    int a = 1, b = 2;

    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
```

```
    cout << "b = " << b << endl;

    // call function by passing variable addresses
    swap(&a, &b);

    cout << "\nAfter swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}

// function definition to swap numbers
void swap(int* n1, int* n2) {
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Output

```
Before swapping
a = 1
b = 2

After swapping
a = 2
b = 1
```

Here, we can see the output is the same as in the previous example. Notice the line,

```
// &a is address of a
// &b is address of b
swap(&a, &b);
```

Here, the address of the variable is passed during the function call rather than the variable.

Since the address is passed instead of the value, a dereference operator * must be used to access the value stored in that address.

```
temp = *n1;
*n1 = *n2;
*n2 = temp;
```

*n1 and *n2 gives the value stored at address n1 and n2 respectively.

Since n1 and n2 contain the addresses of a and b, anything is done to *n1 and *n2 will change the actual values of a and b.

Hence, when we print the values of a and b in the main() function, the values are changed.

C++ Memory Management: new and delete
C++ allows us to allocate the memory of a variable or an array in run time. This is known as dynamic memory allocation.

In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.

In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.

We can allocate and then deallocate memory dynamically using the new and delete operators respectively.

C++ new Operator
The new operator allocates memory to a variable. For example,

```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
// using the new keyword
pointVar = new int;

// assign value to allocated memory
*pointVar = 45;
```

Here, we have dynamically allocated memory for an int variable using the new operator.

Notice that we have used the pointer pointVar to allocate the memory dynamically. This is because the new operator returns the address of the memory location.

In the case of an array, the new operator returns the address of the first element of the array.

From the example above, we can see that the syntax for using the new operator is

```
pointerVariable = new dataType;
```

delete Operator
Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.

For this, the delete operator is used. It returns the memory to the operating system. This is known as memory deallocation.

The syntax for this operator is

```
delete pointerVariable;
```

Consider the code:

```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
// for an int variable
pointVar = new int;

// assign value to the variable memory
*pointVar = 45;

// print the value stored in memory
cout << *pointVar; // Output: 45

// deallocate the memory
delete pointVar;
```

Here, we have dynamically allocated memory for an int variable using the pointer pointVar.

After printing the contents of pointVar, we deallocated the memory using delete.

Note: If the program uses a large amount of unwanted memory using new, the system may crash because there will be no memory available for the operating system. In this case, the delete operator can help the system from crash.

Example 1: C++ Dynamic Memory Allocation

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

int main() {

  // declare an int pointer
  int* pointInt;

  // declare a float pointer
  float* pointFloat;

  // dynamically allocate memory
  pointInt = new int;
  pointFloat = new float;

  // assigning value to the memory
  *pointInt = 45;
  *pointFloat = 45.45f;

  cout << *pointInt << endl;
  cout << *pointFloat << endl;
```

```
  // deallocate the memory
  delete pointInt;
  delete pointFloat;

  return 0;
}
```

Output

```
45
45.45
```

In this program, we dynamically allocated memory to two variables of int and float types. After assigning values to them and printing them, we finally deallocate the memories using the code

```
delete pointInt;
delete pointFloat;
```

Note: Dynamic memory allocation can make memory management more efficient.

Especially for arrays, where a lot of the times we don't know the size of the array until the run time.

Example 2: C++ new and delete Operator for Arrays

```
// C++ Program to store GPA of n number of students and display it
// where n is the number of students entered by the user

#include <iostream>
using namespace std;

int main() {

  int num;
  cout << "Enter total number of students: ";
  cin >> num;
  float* ptr;

  // memory allocation of num number of floats
  ptr = new float[num];

  cout << "Enter GPA of students." << endl;
  for (int i = 0; i < num; ++i) {
    cout << "Student" << i + 1 << ": ";
    cin >> *(ptr + i);
  }

  cout << "\nDisplaying GPA of students." << endl;
  for (int i = 0; i < num; ++i) {
```

```
    cout << "Student" << i + 1 << ": " << *(ptr + i) << endl;
  }

  // ptr memory is released
  delete[] ptr;

  return 0;
}
```

Output

```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9

Displaying GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9
```

In this program, we have asked the user to enter the number of students and store it in the num variable.

Then, we have allocated the memory dynamically for the float array using new.

We enter data into the array (and later print them) using pointer notation.

After we no longer need the array, we deallocate the array memory using the code delete[] ptr;.

Notice the use of [] after delete. We use the square brackets [] in order to denote that the memory deallocation is that of an array.

Exercises

1. Initialize an integer array of 5 elements. Then, print values of all elements along with their addresses using pointers.

2. **Pointer to Array**
   Declare an array and a pointer to the array. Use the pointer to access and print elements of the array.

3. Pointer Comparison
   Create two pointers pointing to different integer variables. Compare the values they point to and print whether they are equal or not.

4. String Manipulation with Pointers
   Write a function that takes a string (array of characters) and reverses it using pointers.

5. Write a program that should have a user-defined function names bubbleSort( ), which should have one parameter of pointer to an integer array. The function should sort the values of that array in ascending order (smaller to larger). Then, in the main( ) function, create an integer array initialized with some values within the code and print the array values along with a message saying that is the original/unsorted array values. After that, pass that array to the function bubbleSort( ) that should sort its values. Finally, print the array values along with a message saying that these are the sorted values.

6. Write a C++ program that creates an integer array of size 10.
   Then, get the array values as input from the user but save them in the array using a pointer.
   Then, print those values using the pointer
   Then, ask the user to search any value and perform a search operation using the pointer

7. Create a function named swap( ) that should swap two integer values (means, exchanges the values between two variables; the value of first variable should be saved in the second one, and the value of the second variable should be saved in the first one, and so on) using pointers. The function parameters should be two pointers. Then, in the main( ) function, create (and initialize) two variables and print their values before calling the swap( ) function. After that, call the swap function. And finally, print those values after the function call to show that the values are swapped/exchanged.