Pointers with Functions and Arrays

Pointers and Functions

In the functions module we learned about passing arguments to functions. The arguments can be passed to functions in two ways;

1. Passing argument by value
2. Passing arguments by reference

Passing arguments by value

By default the arguments in C++ are passed by value. In the previous module all the examples are passed by value only. When the arguments are passed by value, the values of arguments are copied into the function parameters. In passed by value method, any changes made to formal arguments do not modify the actual arguments. The pass by value is suitable many cases so it has some advantages and disadvantages.

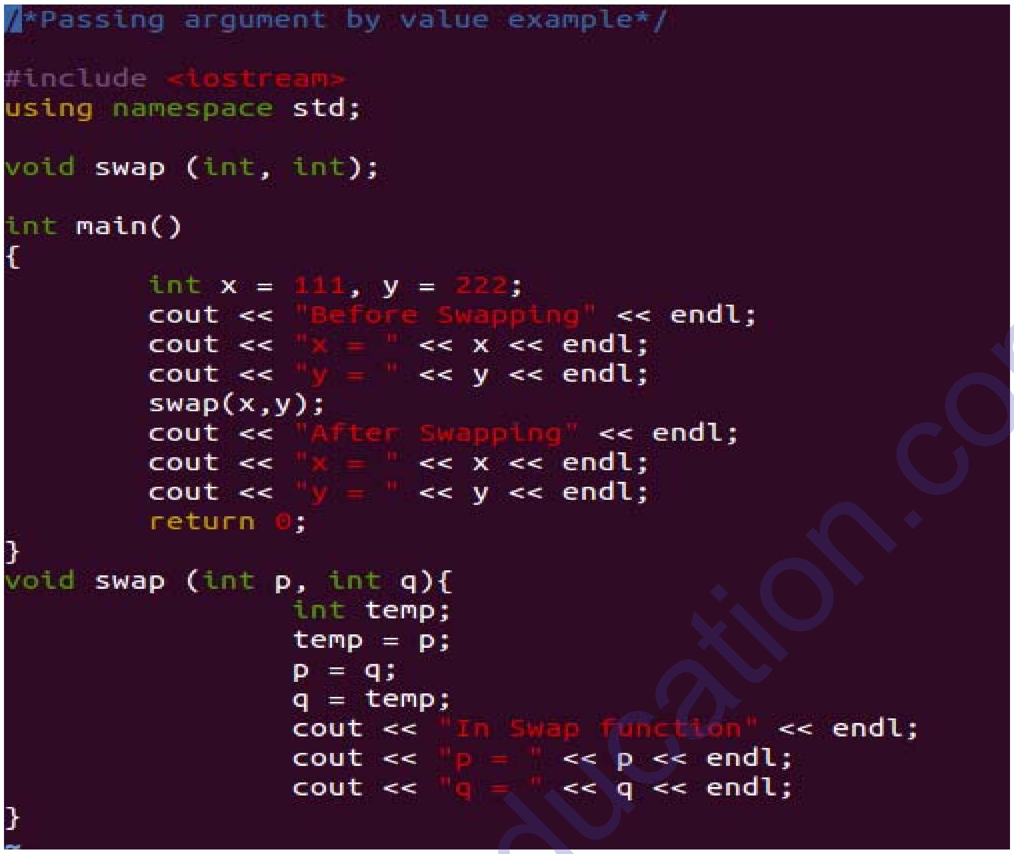
Advantages of pass by value:

- Functions arguments passed by value can be variables, expressions, enumerators, structures and classes.
- Functions arguments are never changed by the called function, which prevents the side effects.

Disadvantages of pass by value:

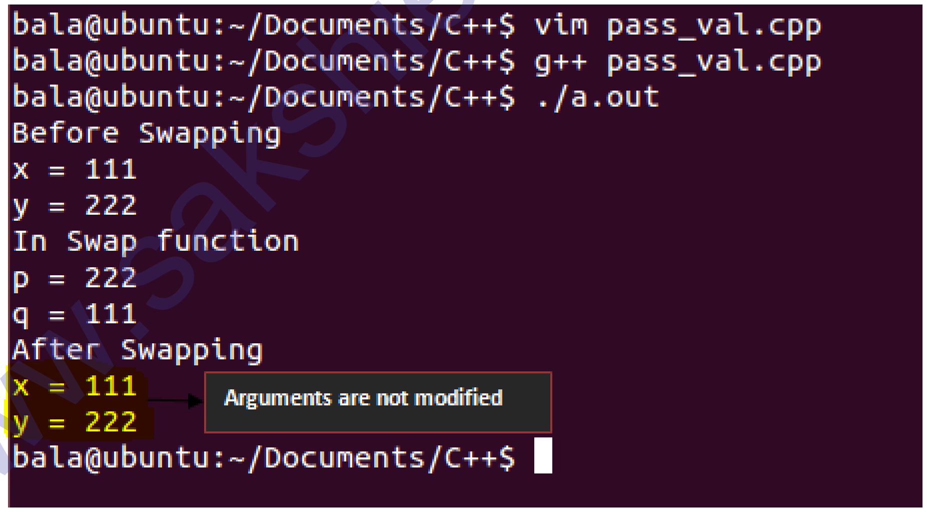
- Copying structures and classes can result a significant performance penalty, especially if the function is called many times.

Let us consider an example which shows the variables in modified in the called function will not modify variables in the calling function. In the below example we are calling swap function from main () function and in the swap function we are swapping the arguments and displaying on the console.



```
/*Passing argument by value example*/

#include <iostream>
using namespace std;

void swap (int, int);

int main()
{
        int x = 111, y = 222;
        cout << "Before Swapping" << endl;
        cout << "x = " << x << endl;
        cout << "y = " << y << endl;
        swap(x,y);
        cout << "After Swapping" << endl;
        cout << "x = " << x << endl;
        cout << "y = " << y << endl;
        return 0;
}
void swap (int p, int q){
                int temp;
                temp = p;
                p = q;
                q = temp;
                cout << "In Swap function" << endl;
                cout << "p = " << p << endl;
                cout << "q = " << q << endl;
}
```

Figure1: Example program for Pass by value

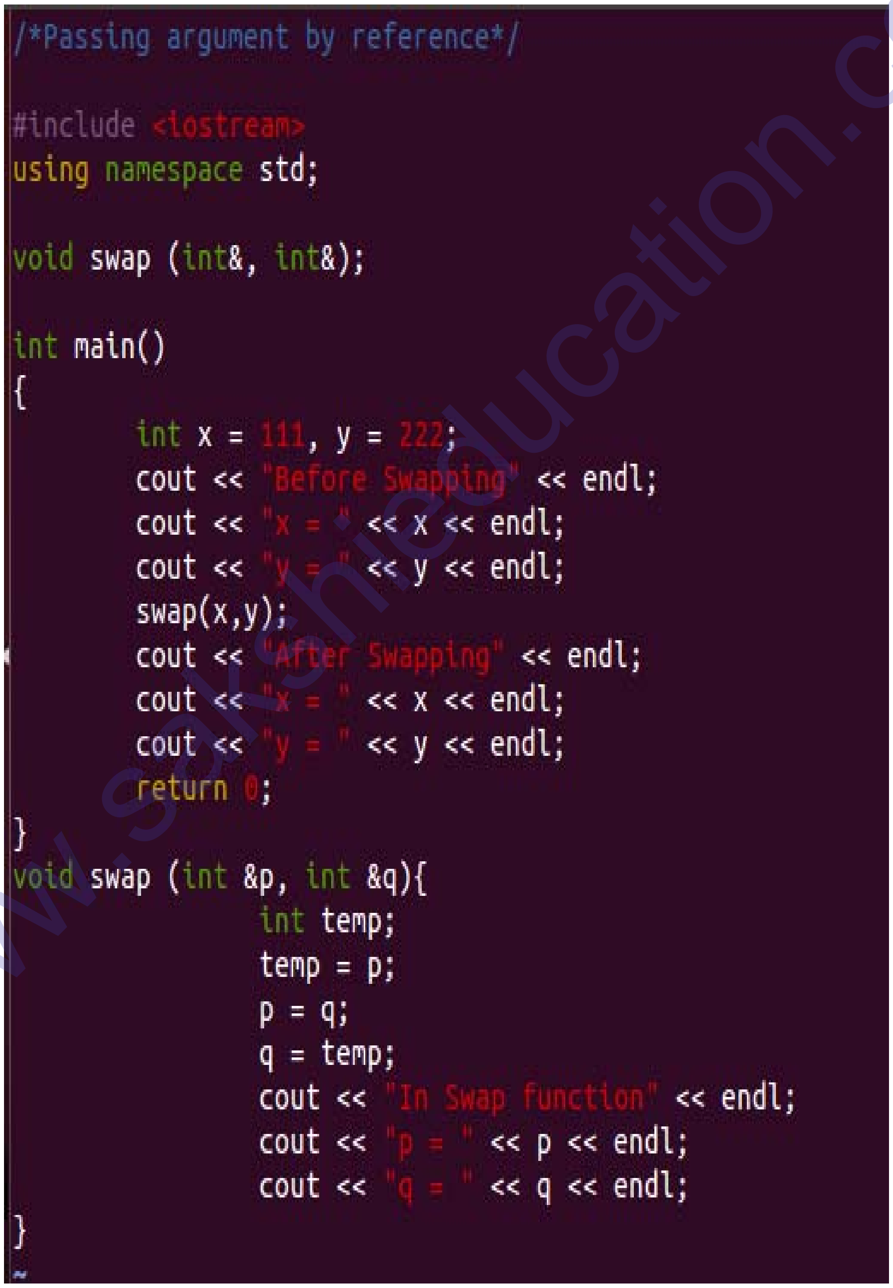


```
bala@ubuntu:~/Documents/C++$ vim pass_val.cpp
bala@ubuntu:~/Documents/C++$ g++ pass_val.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
Before Swapping
x = 111
y = 222
In Swap function
p = 222
q = 111
After Swapping
x = 111
y = 222
bala@ubuntu:~/Documents/C++$
```

Figure2: Result of the program

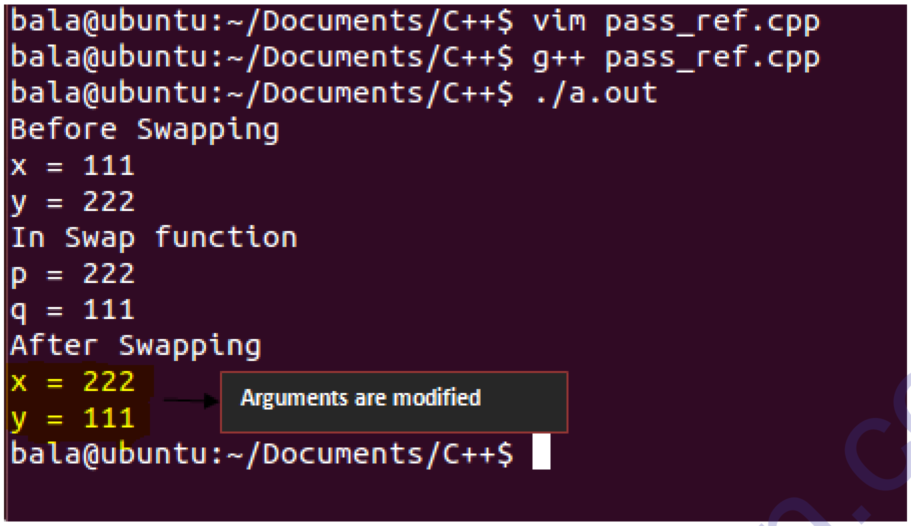
Passing arguments by reference

The pass by value has the following limitations which will affect the performance. When passing a large structure or class to a function using pass by value will make a copy of the argument into the function parameter and it has the only way to return a value back to the calling function is using functions return value. These problems will be overcome in pass by reference, in which the actual value is not passed instead of this the reference of arguments passed to the function. In the example program two integer variables are declared and those integer values are passed to swap function by reference. When the function is called the variables are reference to the arguments 'a' and 'b' in the main function. Here the reference to the variable is treated exactly same as the variable itself, any changes made to the reference variables that will modify the actual arguments.



```
/*Passing argument by reference*/

#include <iostream>
using namespace std;

void swap (int&, int&);

int main()
{
        int x = 111, y = 222;
        cout << "Before Swapping" << endl;
        cout << "x = " << x << endl;
        cout << "y = " << y << endl;
        swap(x,y);
        cout << "After Swapping" << endl;
        cout << "x = " << x << endl;
        cout << "y = " << y << endl;
        return 0;
}
void swap (int &p, int &q){
                int temp;
                temp = p;
                p = q;
                q = temp;
                cout << "In Swap function" << endl;
                cout << "p = " << p << endl;
                cout << "q = " << q << endl;
}
~
```

Figure3: Example program for pass by reference



```
bala@ubuntu:~/Documents/C++$ vim pass_ref.cpp
bala@ubuntu:~/Documents/C++$ g++ pass_ref.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
Before Swapping
x = 111
y = 222
In Swap function
p = 222
q = 111
After Swapping
x = 222
y = 111
bala@ubuntu:~/Documents/C++$
```

Figure4: Result of the program

Advantages of pass by reference:

- The pass by reference allows a function to change the value of the argument. This will be useful sometime but not good. We can use const reference to guarantee the function won't change the arguments if needed.
- It is fast because the copy of argument is not made even when used with large structures and classes.
- References must be initialized, so there is no problem about null values.
- References can be used to return multiple values from a function.

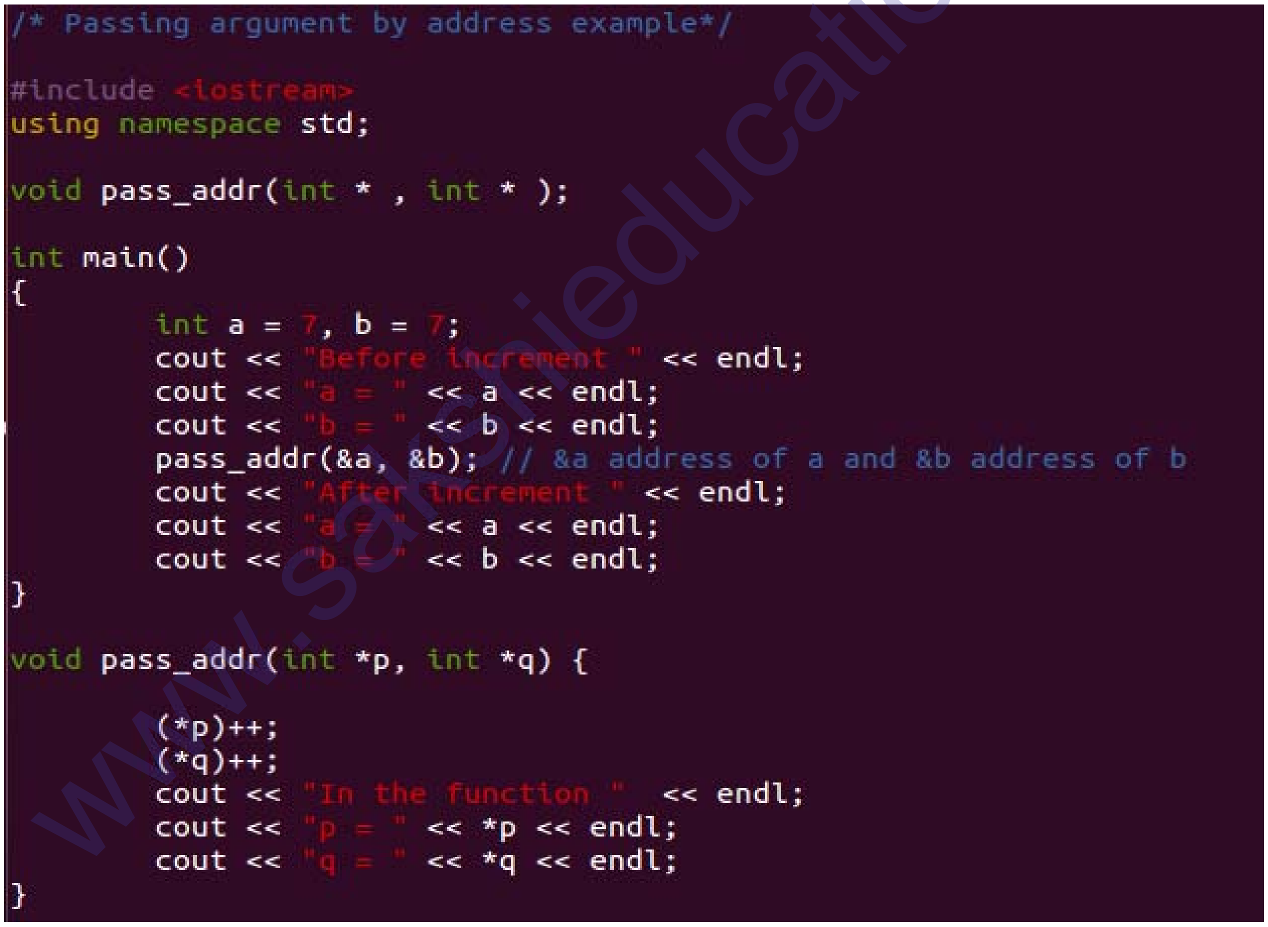
Disadvantages of pass by reference:

- It is impossible to tell from the function call whether the argument may modify or not. We can only know that an argument is passed by value or passed by reference by seeing at function declaration.

There is another method to pass arguments to functions and that is passing an argument by address. In this method the address of variable is passed during the function call rather than variable itself, because the argument is an address, so the function parameter must be pointer. The function can then dereference the pointer to access the value being pointed. Let us consider an example program which increments integer values by passing address of variables 'a' and 'b'. Here passing the address of variables 'a' and address of variable 'b' in the function call, so the receiving formal arguments in the function definition should be declared of pointer type.

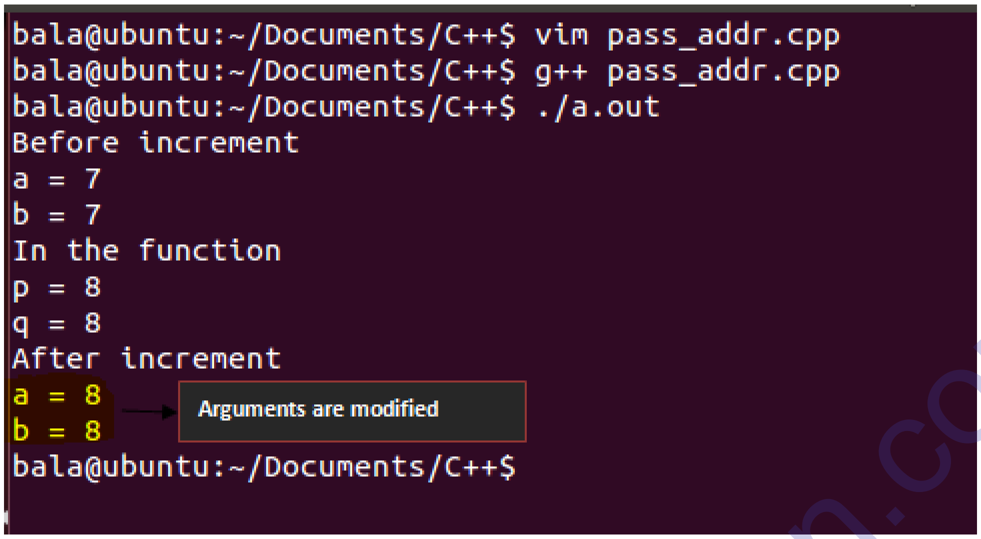
Whenever the function ‘ref ()’ is called the two pointers to integer variable ‘p’ and ‘q’ are created and they are initialized with the address of variables ‘a’ and ‘b’. By dereferencing the pointers variables ‘p’ and ‘q’, we will able to access the variables ‘a’ and ‘b’. Before the ‘ref ()’ function the values of ‘a’ and ‘b’ are ‘7’.

The values of actual arguments are copied into pointer variables ‘p’ and ‘q’ and here the actual arguments are addresses of variables ‘a’ and ‘b’. Since ‘p’ contain address of variable ‘a’ and ‘q’ contain the address of variable ‘b’ inside the ‘ref ()’ function. In the function the variables ‘p’ and ‘q’ are incremented and the values are displayed on the console and come back into main function. Here we can observe that the values of variables ‘a’ and ‘b’ are changed. Finally we can say the by the passing arguments through reference and by address it will modify the actual arguments also.



```
/* Passing argument by address example*/

#include <iostream>
using namespace std;

void pass_addr(int * , int * );

int main()
{
        int a = 7, b = 7;
        cout << "Before increment " << endl;
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
        pass_addr(&a, &b); // &a address of a and &b address of b
        cout << "After increment " << endl;
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
}

void pass_addr(int *p, int *q) {

        (*p)++;
        (*q)++;
        cout << "In the function "  << endl;
        cout << "p = " << *p << endl;
        cout << "q = " << *q << endl;
}
```

Figure5: Example program for pass by address



```
bala@ubuntu:~/Documents/C++$ vim pass_addr.cpp
bala@ubuntu:~/Documents/C++$ g++ pass_addr.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
Before increment
a = 7
b = 7
In the function
p = 8
q = 8
After increment
a = 8
b = 8
bala@ubuntu:~/Documents/C++$
```

Figure6: Result of the program

Advantages of pass by address:

- The pass by address allows a function to change the value of the argument.
- It is fast because the copy of argument is not made even when used with large structures and classes.
- It can be used to return multiple values from a function.

Disadvantages of pass by address:

- Pointer arguments must be normal variables because the literals and expressions do not have addresses.
- Accessing arguments passed by address is slower than the accessing arguments passed by value because the dereferencing pointer is slower than accessing a value directly.
- All values must be checked to see whether they are null, because trying to dereference a null value crash the program.

➔ Use 'passing arguments by value' for fundamental data types and enumerators.

➔ Use 'passing arguments by reference' when you need to modify an argument and passing classes and structures.

➔ Use 'passing arguments by addresses' for passing arrays.

The returning values from a called function to calling function by value and reference or address works almost same way as passing arguments to a function. Each of these methods has their own advantages and disadvantages. The more common problem is returning local variables declared in the functions, because the local variables declared in a function will go out of scope when the function returns.

Returning by value

Return by value is the simplest return type to use. When the values are returned by the value, a copy of that value is returned to the calling function. Using return by value we can return only variables, literals and expressions. The advantage of return by value is that we can return variables or expressions that involved local variables declared within the function without having any problem about scope of the variable, because the variables are evaluated before the function goes out of the scope and a copy of the value is returned to the calling function. So there is no problem with scope of variables at the end of the function. The problem with return by value is slow for structures and classes.

Return by reference

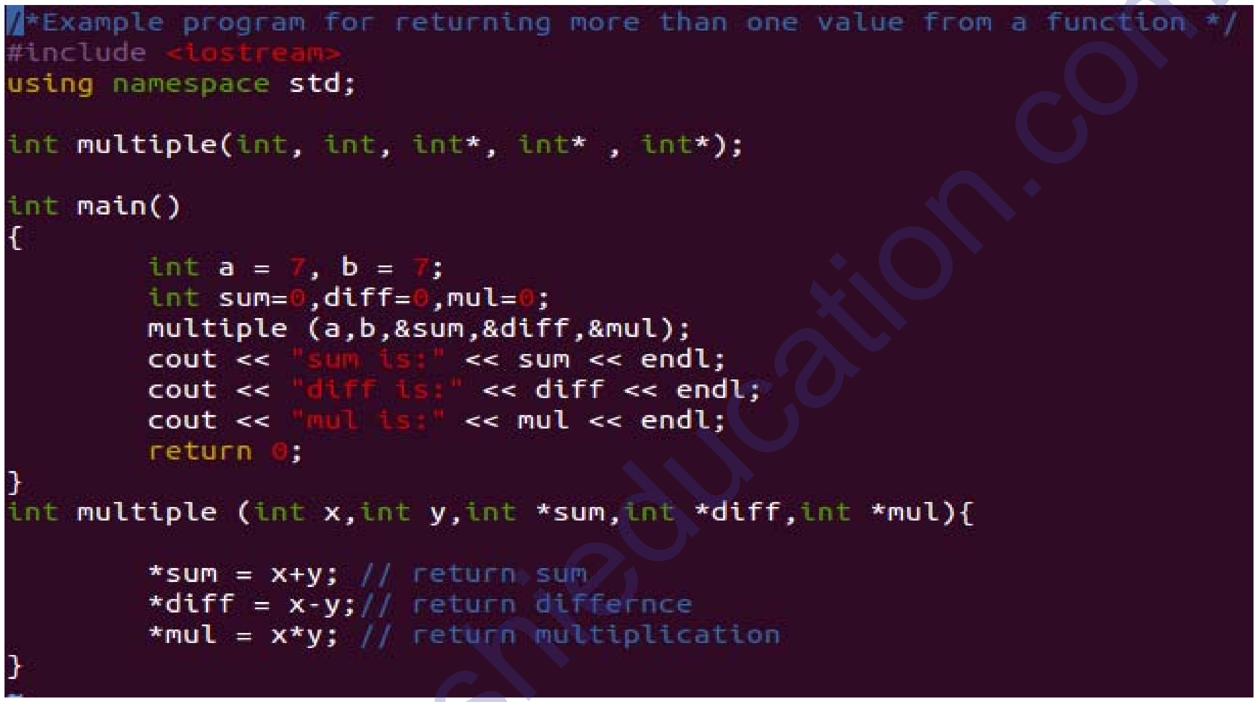
The values returned by reference must be variables. When a variable is returned by reference, a reference to the variable is passed back to the calling function. The calling function then uses this reference to modify the variable, when it needed. Return by reference is also fast and it is mostly useful when returning structures and classes. Return by reference is not used for literals and expressions because the literals and expressions do not have addresses.

Return by address

Returning by address involves returning the address of a variable to the calling function. Return by address can only return the address of the variable, so it cannot be used for expressions and literals, because return by address copies an address from the function to the calling function. When using return by reference or address makes sure that you are not returning a reference to or an address of a variable that will go out of the scope when the function returns.

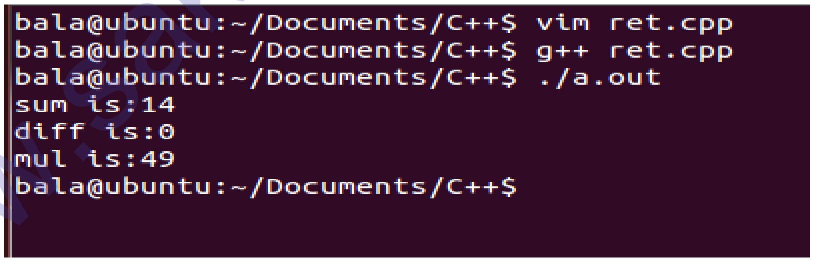
Returning more than one value from a function

We can return only one value from a function using return statement. By using pass by reference we can overcome this problem and return more than one value. Let us take one example to better understand this concept. Let us take a function which perform addition, subtraction and multiplication on two integer variables and return sum, difference and multiplication to the calling function. In the following program the variables ‘a’ and ‘b’ are passed by value and the variables ‘sum’, ‘diff’ and ‘mul’ are passed by reference . The function ‘multiple’ knows the address of variables ‘sum’, ‘diff’ and ‘mul’, so it can access these variables indirectly using pointers and assign the appropriate values to them.



```
/*Example program for returning more than one value from a function */
#include <iostream>
using namespace std;

int multiple(int, int, int*, int* , int*);

int main()
{
        int a = 7, b = 7;
        int sum=0,diff=0,mul=0;
        multiple (a,b,&sum,&diff,&mul);
        cout << "sum is:" << sum << endl;
        cout << "diff is:" << diff << endl;
        cout << "mul is:" << mul << endl;
        return 0;
}
int multiple (int x,int y,int *sum,int *diff,int *mul){

        *sum = x+y; // return sum
        *diff = x-y;// return differnce
        *mul = x*y; // return multiplication
}
```

Figure7: Example program for returning more than one value from a function

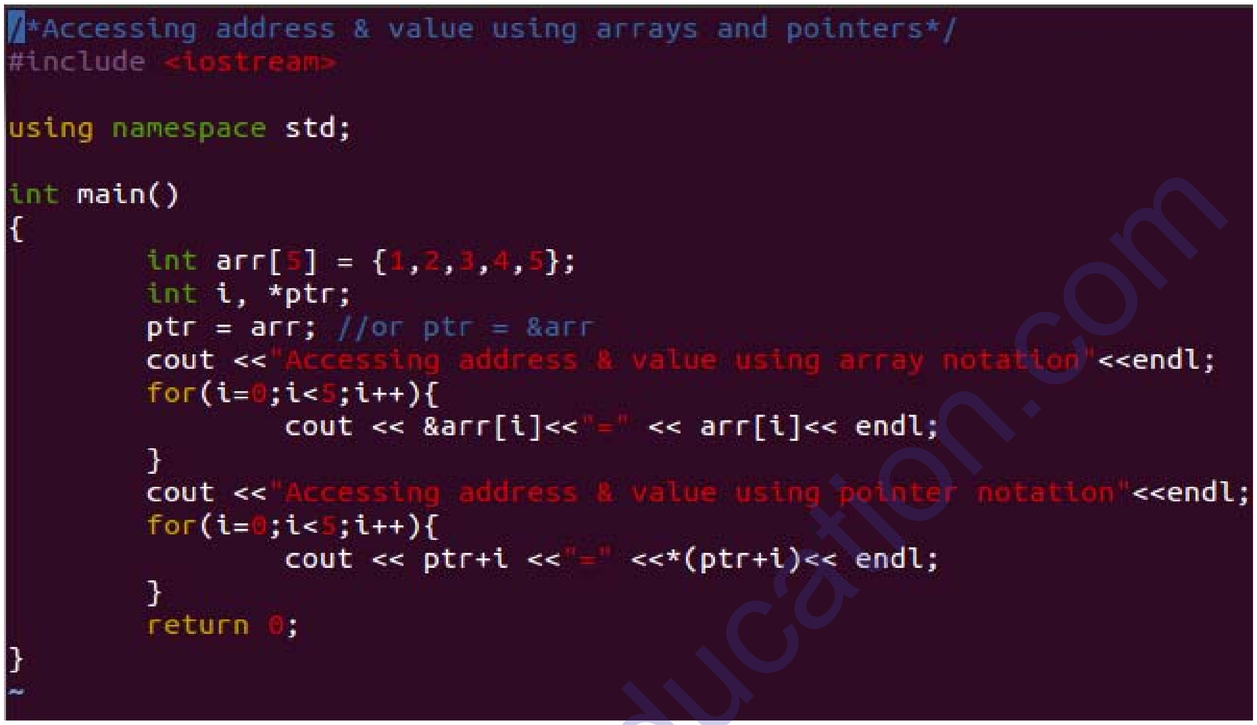


```
bala@ubuntu:~/Documents/C++$ vim ret.cpp
bala@ubuntu:~/Documents/C++$ g++ ret.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
sum is:14
diff is:0
mul is:49
bala@ubuntu:~/Documents/C++$
```

Figure8: Result of the program

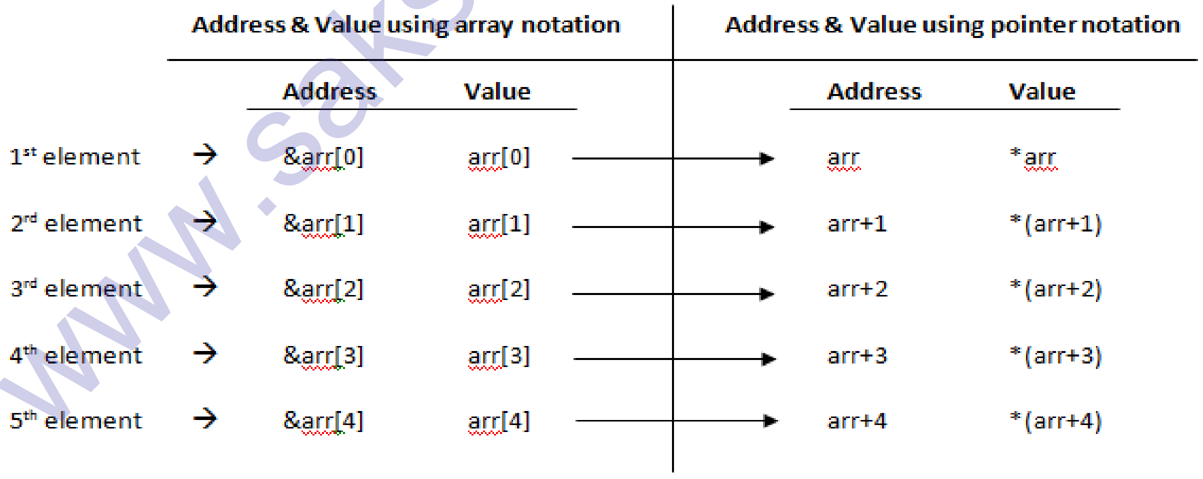
The relation between pointer and an array

The pointers and arrays are closely related. We can access array elements using pointer expressions. By default the compiler converts the array subscript notation to pointer notation to access the array elements. Let us consider an example program which displays the address and value of each element in an array using array notation and pointer notation. In the below example program we defined integer array of five elements.



```
/*Accessing address & value using arrays and pointers*/
#include <iostream>

using namespace std;

int main()
{
        int arr[5] = {1,2,3,4,5};
        int i, *ptr;
        ptr = arr; //or ptr = &arr
        cout <<"Accessing address & value using array notation"<<endl;
        for(i=0;i<5;i++){
                cout << &arr[i]<<"=" << arr[i]<< endl;
        }
        cout <<"Accessing address & value using pointer notation"<<endl;
        for(i=0;i<5;i++){
                cout << ptr+i <<"=" <<*(ptr+i)<< endl;
        }
        return 0;
}
~
```

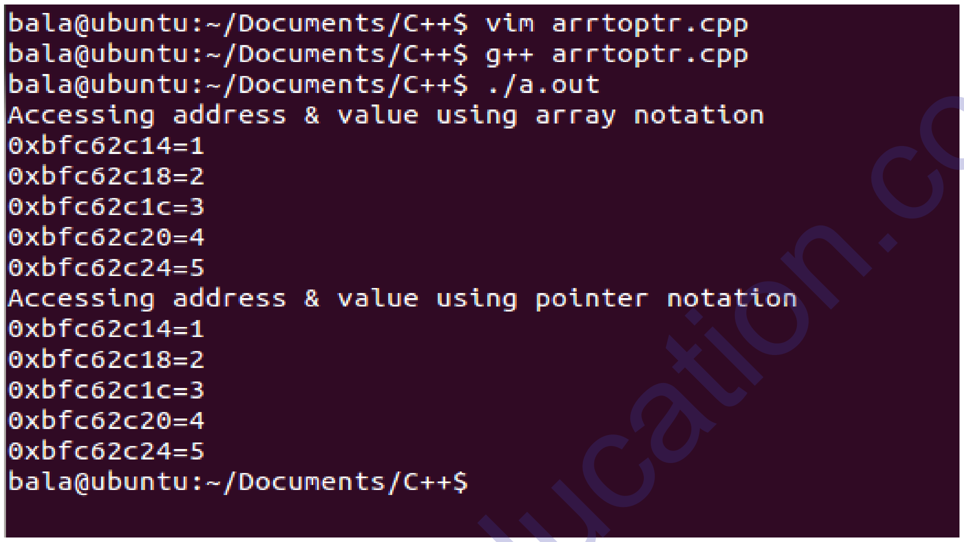
Figure9: Example program for accessing address and value using array notation and pointer notation



		Address & Value using array notation			Address & Value using pointer notation	
		Address	Value		Address	Value
1st element	→	&arr[0]	arr[0]	→	arr	*arr
2rd element	→	&arr[1]	arr[1]	→	arr+1	*(arr+1)
3rd element	→	&arr[2]	arr[2]	→	arr+2	*(arr+2)
4th element	→	&arr[3]	arr[3]	→	arr+3	*(arr+3)
5th element	→	&arr[4]	arr[4]	→	arr+4	*(arr+4)

Figure10: Comparison between array notation and pointer notation

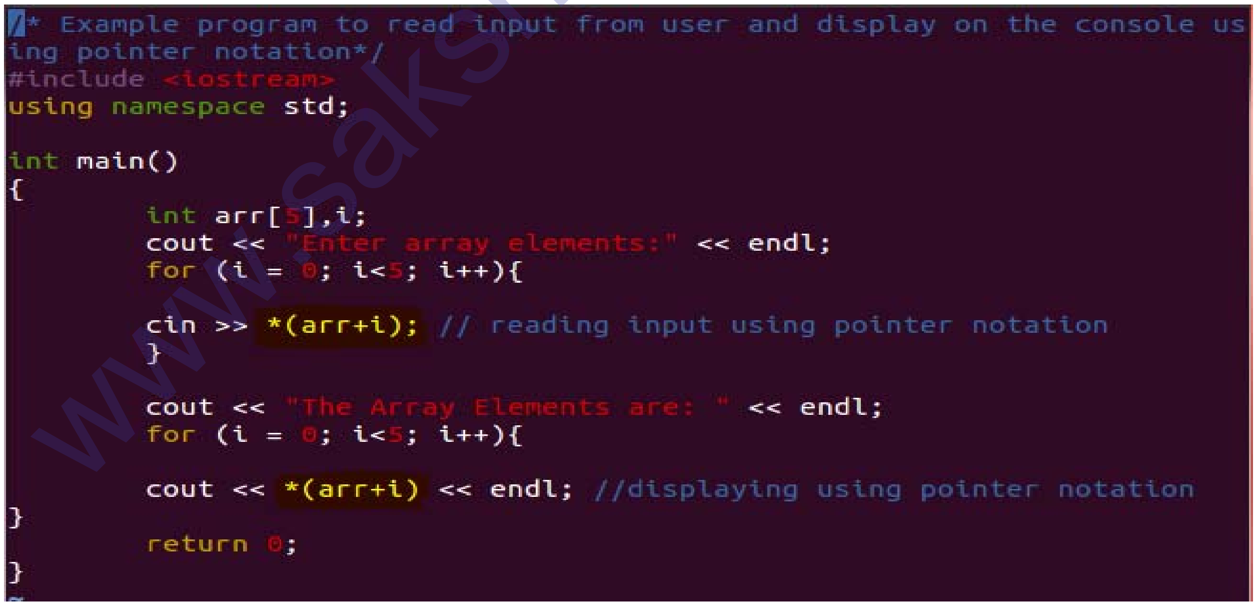
Already we know that the name of the array holds the address of the first element in the array, so the address of array can be known by ‘arr’ or ‘&arr[0]’ and this two points to the same memory address location. The name of the array is a constant pointer and according to pointer arithmetic when an integer is added to a pointer then the address of the next element is of same base type, so ‘arr+1’ will give address of the next element.



```
bala@ubuntu:~/Documents/C++$ vim arrtoptr.cpp
bala@ubuntu:~/Documents/C++$ g++ arrtoptr.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
Accessing address & value using array notation
0xbfc62c14=1
0xbfc62c18=2
0xbfc62c1c=3
0xbfc62c20=4
0xbfc62c24=5
Accessing address & value using pointer notation
0xbfc62c14=1
0xbfc62c18=2
0xbfc62c1c=3
0xbfc62c20=4
0xbfc62c24=5
bala@ubuntu:~/Documents/C++$
```

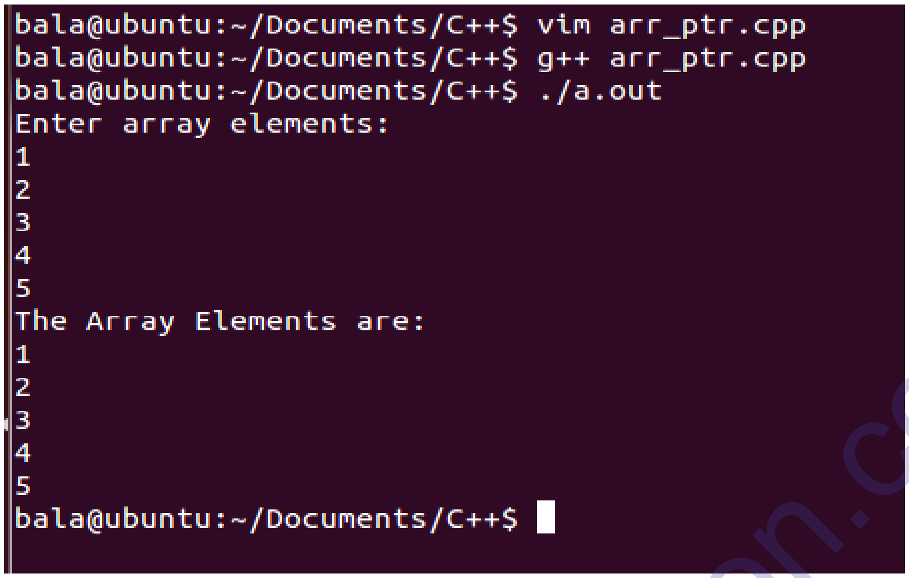
Figure11: Result of the program

Let us consider example program to read input values from user and display on the console using pointer notation.



```
/* Example program to read input from user and display on the console us
ing pointer notation*/
#include <iostream>
using namespace std;

int main()
{
        int arr[5],i;
        cout << "Enter array elements:" << endl;
        for (i = 0; i<5; i++){

        cin >> *(arr+i); // reading input using pointer notation
        }

        cout << "The Array Elements are: " << endl;
        for (i = 0; i<5; i++){

        cout << *(arr+i) << endl; //displaying using pointer notation
}
        return 0;
}
```

Figure12: Example program to read input values from user and display on the console using pointer notation.



```
bala@ubuntu:~/Documents/C++$ vim arr_ptr.cpp
bala@ubuntu:~/Documents/C++$ g++ arr_ptr.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
Enter array elements:
1
2
3
4
5
The Array Elements are:
1
2
3
4
5
bala@ubuntu:~/Documents/C++$
```

Figure13: Result of the program

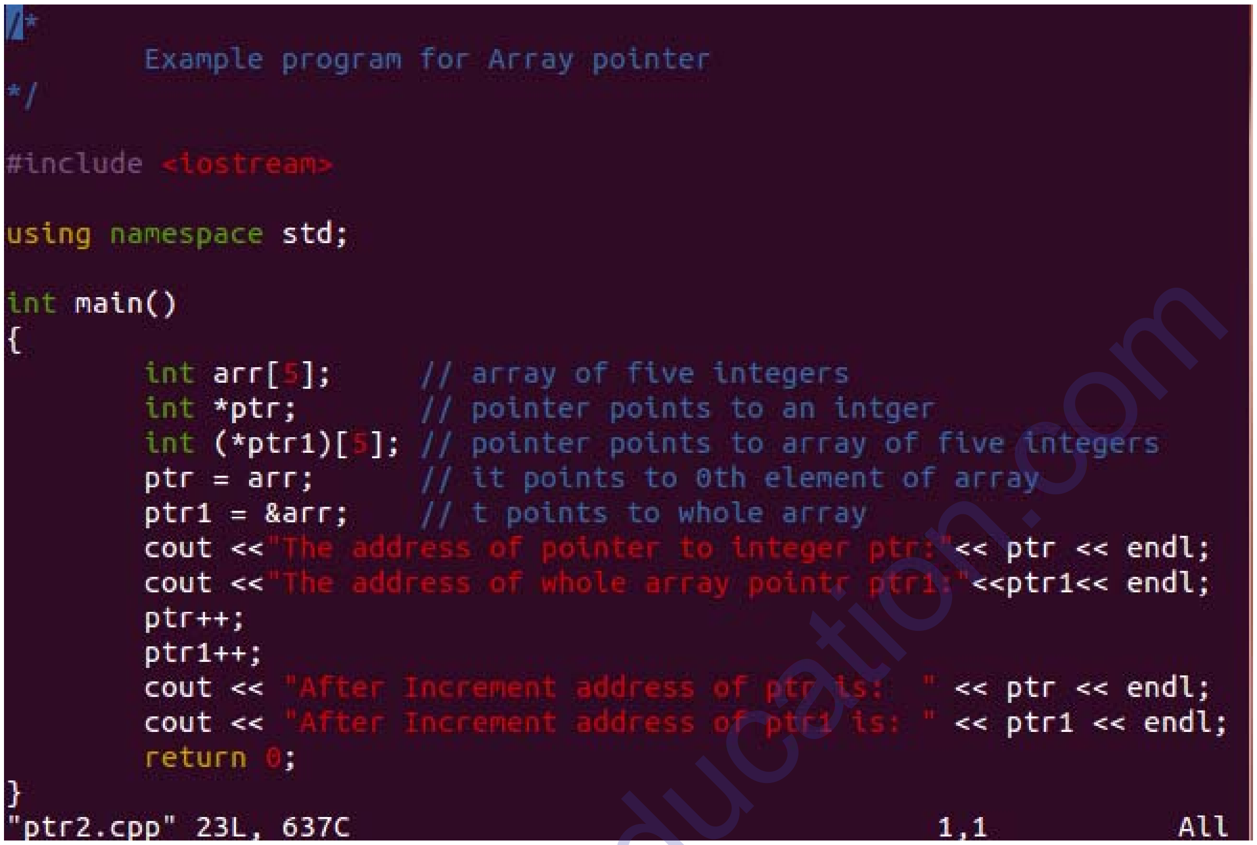
Array pointer

Till now we learn that a pointer is pointed to 0th element of array. We can also declare a pointer that can point to the whole array instead of only pointing one element of array. These types of pointers are useful when talking about multidimensional arrays.

Example:

```
int (*ptr)[5];
```

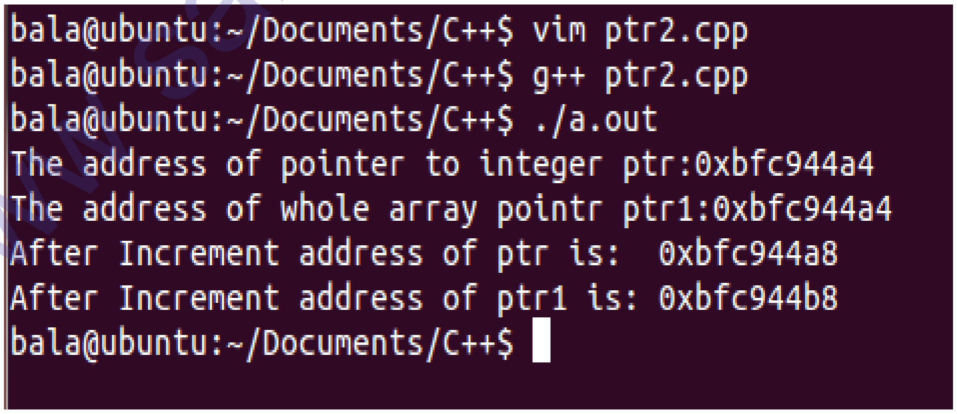
Here ‘ptr’ is a pointer to array of five integers. The subscripts have the higher precedence than indirection operator, so it is necessary to enclose indirection operator and pointer name in the parentheses. The pointer that points to 0th element of array and the pointer that points to the whole array are different. Let us consider a program which will show the difference between this.



```
/*
        Example program for Array pointer
*/

#include <iostream>

using namespace std;

int main()
{
        int arr[5];     // array of five integers
        int *ptr;       // pointer points to an intger
        int (*ptr1)[5]; // pointer points to array of five integers
        ptr = arr;      // it points to 0th element of array
        ptr1 = &arr;    // t points to whole array
        cout <<"The address of pointer to integer ptr:"<< ptr << endl;
        cout <<"The address of whole array pointr ptr1:"<<ptr1<< endl;
        ptr++;
        ptr1++;
        cout << "After Increment address of ptr is:  " << ptr << endl;
        cout << "After Increment address of ptr1 is: " << ptr1 << endl;
        return 0;
}
"ptr2.cpp" 23L, 637C                                        1,1           All
```

Figure14: Example program for array pointer

In the above, the pointer 'ptr' points to 0th element of array, whereas the pointer 'ptr1' points to whole array. After arithmetic increment on both pointers the 'ptr' shifted forward only four bytes and whereas 'ptr1' is shifter, forward 20 bytes since 'ptr1' is pointer to point array of five elements.



```
bala@ubuntu:~/Documents/C++$ vim ptr2.cpp
bala@ubuntu:~/Documents/C++$ g++ ptr2.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
The address of pointer to integer ptr:0xbfc944a4
The address of whole array pointr ptr1:0xbfc944a4
After Increment address of ptr is:  0xbfc944a8
After Increment address of ptr1 is: 0xbfc944b8
bala@ubuntu:~/Documents/C++$
```

Figure15: Result of the program