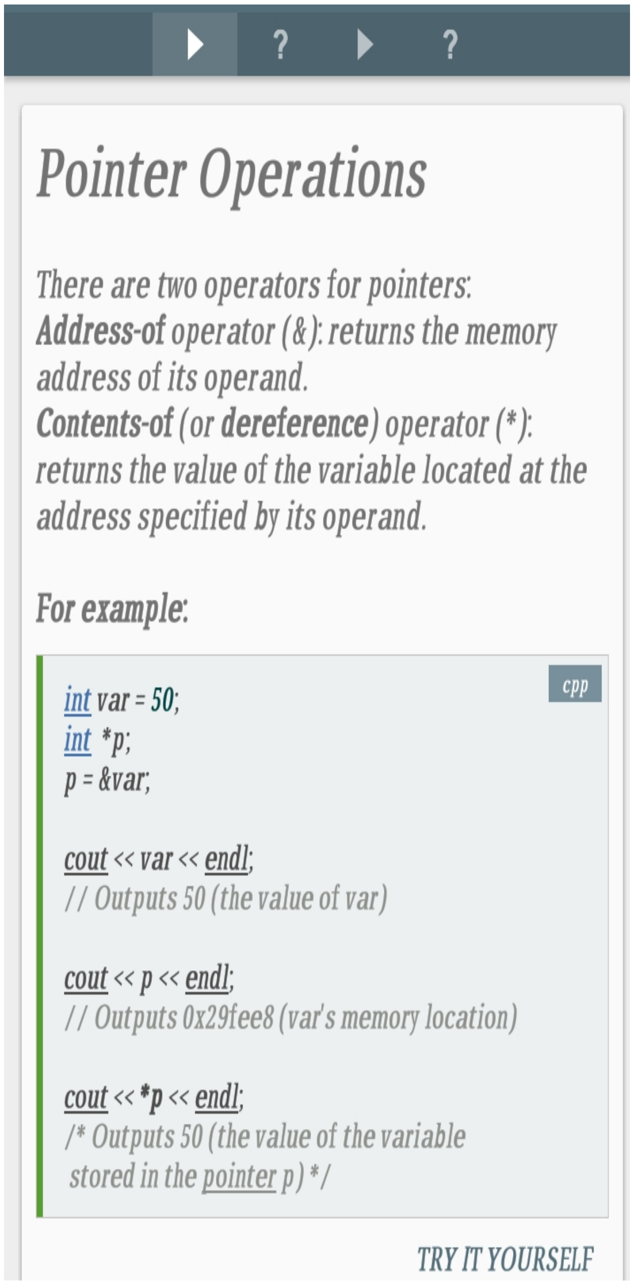
DAILY ASSESSMENT FORMAT

Date:	23rd June 2020	**Name:**	Soundarya NA
Course:	C++ programming	**USN:**	4AL16EC077
Topic:	Data types, arrays, pointers Functions	**Semester & Section:**	8th - B

FORENOON SESSION DETAILS

Image of session



Pointer Operations

There are two operators for pointers:
Address-of operator (&): returns the memory address of its operand.
Contents-of (or **dereference**) operator (*): returns the value of the variable located at the address specified by its operand.

For example:

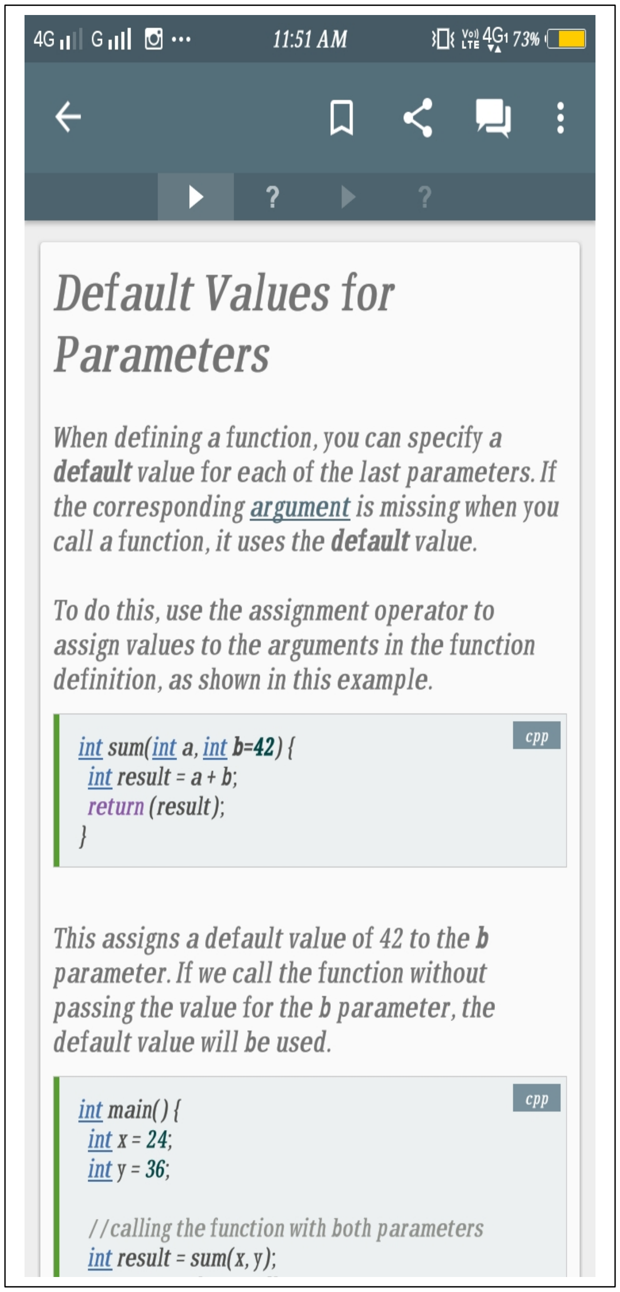
```cpp
int var = 50;
int *p;
p = &var;

cout << var << endl;
// Outputs 50 (the value of var)

cout << p << endl;
// Outputs 0x29fee8 (var's memory location)

cout << *p << endl;
/* Outputs 50 (the value of the variable
 stored in the pointer p) */
```

TRY IT YOURSELF



Default Values for Parameters

When defining a function, you can specify a **default** value for each of the last parameters. If the corresponding argument is missing when you call a function, it uses the **default** value.

To do this, use the assignment operator to assign values to the arguments in the function definition, as shown in this example.

```cpp
int sum(int a, int b=42) {
  int result = a + b;
  return (result);
}
```

This assigns a default value of 42 to the **b** parameter. If we call the function without passing the value for the b parameter, the default value will be used.

```cpp
int main() {
  int x = 24;
  int y = 36;

  //calling the function with both parameters
  int result = sum(x, y);
```

Report:

Pointers are the variables that hold address. Not only can pointers store address of a single variable, it can also store address of cells of an array.

Consider this example:

int* ptr;

int a[5];

ptr = &a[2]; // &a[2] is the address of third element of a[5].

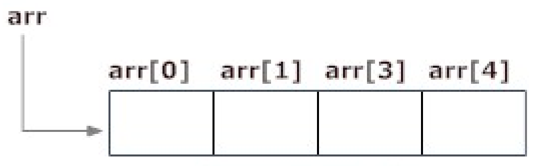


Figure: Array as Pointer

Suppose, pointer needs to point to the fourth element of an array, that is, hold address of fourth array element in above case.

Since ptr points to the third element in the above example, ptr + 1 will point to the fourth element.

You may think, ptr + 1 gives you the address of next byte to the ptr. But it's not correct.

This is because pointer ptr is a pointer to an int and size of int is fixed for a operating system (size of int is 4 byte of 64-bit operating system). Hence, the address between ptr and ptr + 1 differs by 4 bytes.

If pointer ptr was pointer to char then, the address between ptr and ptr + 1 would have differed by 1 byte since size of a character is 1 byte.

Code:

C++ Program to display address of elements of an array using both array and pointers:

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


int main()
{
   float arr[5];
   float *ptr;
   cout << "Displaying address using arrays: " << endl;
   for (int i = 0; i < 5; ++i)
   {
      cout << "&arr[" << i << "] = " << &arr[i] << endl;
   }

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

   cout<<"\nDisplaying address using pointers: "<< endl;
   for (int i = 0; i < 5; ++i)
   {
      cout << "ptr + " << i << " = "<< ptr + i << endl;
   }

   return 0;
}
```

Output:

```
Displaying address using arrays:
&arr[0] = 0x7fff5fbff880
&arr[1] = 0x7fff5fbff884
&arr[2] = 0x7fff5fbff888
&arr[3] = 0x7fff5fbff88c
&arr[4] = 0x7fff5fbff890

Displaying address using pointers:
ptr + 0 = 0x7fff5fbff880
ptr + 1 = 0x7fff5fbff884
ptr + 2 = 0x7fff5fbff888
ptr + 3 = 0x7fff5fbff88c
ptr + 4 = 0x7fff5fbff890
```

In the above program, a different pointer ptr is used for displaying the address of array elements arr. But array elements can be accessed using pointer notation by using same array name arr. For example:

```
int arr[3];
&arr[0] is equivalent to arr
&arr[1] is equivalent to arr + 1
&arr[2] is equivalen to arr + 2
```

Code:

C++ Program to display address of array elements using pointer notation.

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

int main() {
   float arr[5];

   cout<<"Displaying address using pointers notation: "<< endl;
   for (int i = 0; i < 5; ++i) {
      cout << arr + i <<endl;
   }

   return 0;
}
```

Output:

```
Displaying address using pointers notation:

0x7fff5fbff8a0

0x7fff5fbff8a4

0x7fff5fbff8a8
```

```
0x7fff5fbff8ac

0x7fff5fbff8b0
```

You know that, pointer ptr holds the address and expression *ptr gives the value stored in the address.

Similarly, you can get the value stored in the pointer ptr + 1 using *(ptr + 1).

Consider this code below:

int ptr[5] = {3, 4, 5, 5, 3};

- &ptr[0] is equal to ptr and *ptr is equal to ptr[0]
- &ptr[1] is equal to ptr + 1 and *(ptr + 1) is equal to ptr[1]
- &ptr[2] is equal to ptr + 2 and *(ptr + 2) is equal to ptr[2]
- &ptr[i] is equal to ptr + i and *(ptr + i) is equal to ptr[i]

Code:

C++ Program to insert and display data entered by using pointer notation:

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

int main() {
   float arr[5];

  // Inserting data using pointer notation
   cout << "Enter 5 numbers: ";
   for (int i = 0; i < 5; ++i) {
      cin >> *(arr + i) ;
   }

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

Functions:

A function is a block of code that performs a specific task.

Suppose we need to create a program to create a circle and color it. We can create two functions to solve this problem:

- a function to draw the circle
- a function to color the circle

Dividing a complex problem into smaller chunks makes our program easy to understand and reusable.

There are two types of function:

1. **Standard Library Functions:** Predefined in C++
2. **User-defined Function:** Created by users

C++ user defined functions:

C++ allows the programmer to define their own function.

A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).

When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

C++ function declaration:

Syntax:

```
returnType functionName (parameter1, parameter2,...) {
   // function body
}
```

Example:

```
// function declaration
void greet() {
   cout << "Hello World";
}
```

Calling a function:

In the above program, we have declared a function named greet(). To use the greet() function, we need to call it.

Here's how we can call the above greet() function.

```
int main() {

   // calling a function
   greet();

}
```

Code:

```
#include <iostream>
```

```
using namespace std;

// declaring a function
void greet() {
   cout << "Hello there!";
}

int main() {

   // calling the function
   greet();

   return 0;
}
```

Output:

```
Hello there!
```

Function Parameters:

As mentioned above, a function can be declared with parameters (arguments). A parameter is a value that is passed when declaring a function.

For example, let us consider the function below:

```
void printNum(int num) {
   cout << num;
}
```

Here, the int variable num is the function parameter.

We pass a value to the function parameter while calling the function.

```
int main() {
   int n = 7;

   // calling the function
   // n is passed to the function as argument
   printNum(n);

   return 0;
}
```

Code:

Function with parameters:

```
// program to print a text

#include <iostream>
using namespace std;

// display a number
void displayNum(int n1, float n2) {
   cout << "The int number is " << n1;
   cout << "The double number is " << n2;
}

int main() {

    int num1 = 5;
    double num2 = 5.5;

   // calling the function
   displayNum(num1, num2);

   return 0;
}
```

Output:

```
The int number is 5

The double number is 5.5
```

Code:

Return statement:

```
// program to add two numbers using a function

#include <iostream>

using namespace std;

// declaring a function
int add(int a, int b) {
   return (a + b);
}

int main() {

   int sum;

   // calling the function and storing
   // the returned value in sum
   sum = add(100, 78);

   cout << "100 + 78 = " << sum << endl;

   return 0;
}
```

Output:

```
100 + 78 = 178
```

Code:

Function prototype:

```
// using function definition after main() function
// function prototype is declared before main()

#include <iostream>

using namespace std;

// function prototype
int add(int, int);

int main() {
   int sum;

   // calling the function and storing
   // the returned value in sum
   sum = add(100, 78);

   cout << "100 + 78 = " << sum << endl;

   return 0;
}

// function definition
int add(int a, int b) {
   return (a + b);
}
```

Output:

```
100 + 78 = 178
```

Benefits of Using User-Defined Functions:

- Functions make the code reusable. We can declare them once and use them multiple times
- Functions make the program easier as each small task is divided into a function
- Functions increase readability

C++ Library Functions:

Library functions are the built-in functions in C++ programming.

Programmers can use library functions by invoking the functions directly; they don't need to write the functions themselves.

Some common library functions in C++ are sqrt(), abs(), isdigit(), etc.

In order to use library functions, we usually need to include the header file in which these library functions are defined.

For instance, in order to use mathematical functions such as sqrt() and abs(), we need to include the header file cmath.

Code:

C++ Program to Find the Square Root of a Number:

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

int main() {
  double number, squareRoot;

  number = 25.0;

  // sqrt() is a library function to calculate the square root
  squareRoot = sqrt(number);

  cout << "Square root of " << number << " = " << squareRoot;
```

```
    return 0;
}
```

Output:

```
Square root of 25 = 5
```