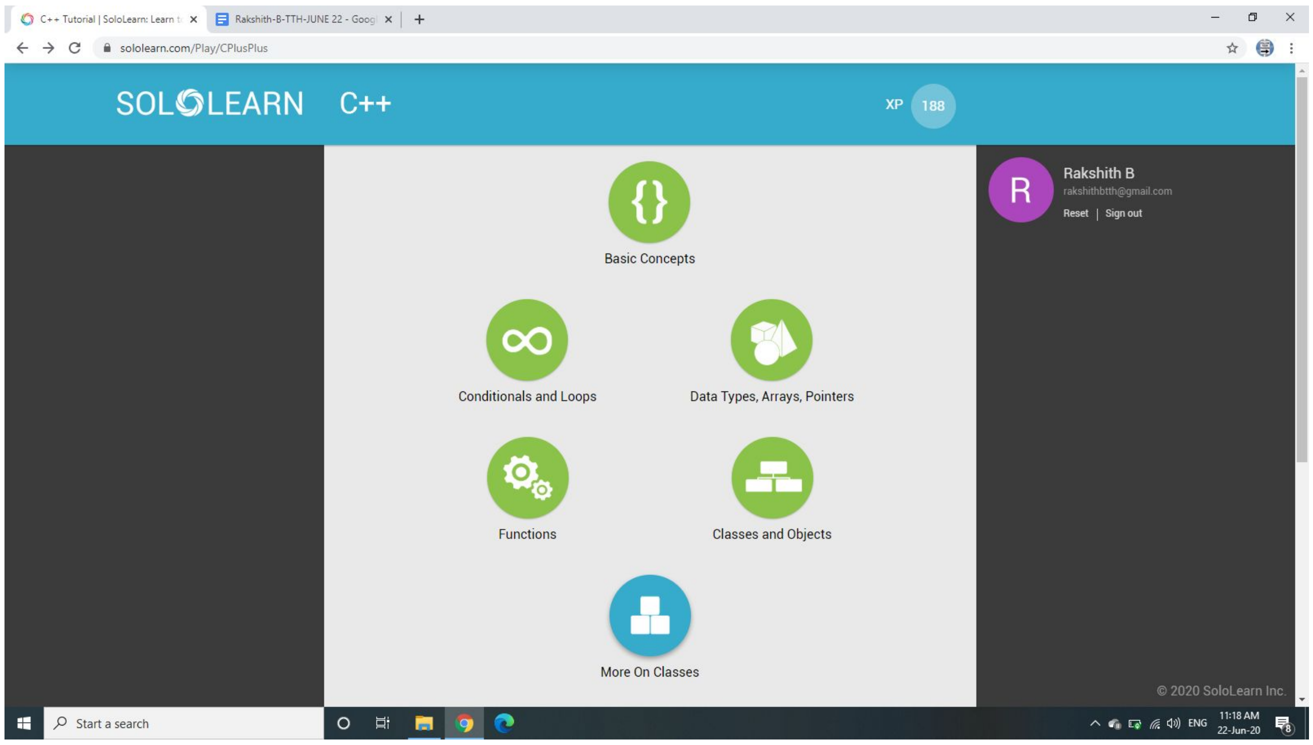
REPORT JUNE 23

Date:	23 JUNE 2020	Name:	Rakshith B
Course:	Solo Learning	USN:	4AL16EC409
Topic:	Data types,Arrays,Pointers,Functions	Semester & Section:	6th SEM B
Github Repository:	Rakshith-B		

FORENOON SESSION DETAILS

Image of session



Report –

Data Types

The operating system allocates memory and selects what will be stored in the reserved memory based on the variable's **data type**.

The data type defines the proper use of an identifier, what kind of data can be stored, and which types of operations can be performed.

Expressions

The examples below show legal and illegal C++ expressions.

```
55+15 //  legal C++ expression
//Both operands of the + operator are integers
55 + "John" // illegal
// The + operator is not defined for integer and string
```

Numeric Data Types

Numeric data types include:
Integers (whole numbers), such as -7, 42.
Floating point numbers, such as 3.14, -42.67.

Strings & Characters

A **string** is composed of numbers, characters, or symbols. String literals are placed in **double quotation** marks; some examples are "Hello", "My name is David", and similar.
Characters are single letters or symbols, and must be enclosed between **single quotes**, like 'a', 'b', etc.

Booleans

The Boolean data type returns just two possible values: **true** (1) and **false** (0).

Integers

The **integer** type holds non-fractional numbers, which can be positive or negative. Examples of integers would include 42, -42, and similar numbers.

Integers

Use the **int** keyword to define the integer data type.

```
int a = 42;
```

Several of the basic types, including integers, can be modified using one or more of these type **modifiers**:
signed: A signed integer can hold both negative and positive numbers.
unsigned: An unsigned integer can hold only positive values.
short: Half of the default size.
long: Twice the default size.

For example:

```
unsigned long int a;
```

Floating Point Numbers

A **floating point** type variable can hold a real number, such as 420.0, -3.33, or 0.03325.
The words floating point refer to the fact that a varying number of digits can appear before and after the decimal point. You could say that the decimal has the ability to "**float**".

There are three different floating point data types: **float**, **double**, and **long double**.
In most modern architectures, a **float** is 4 bytes, a **double** is 8, and a **long double** can be equivalent to a double (8 bytes), or 16 bytes.
For example:

```
double temp = 4.21;
```

Strings
A **string** is an ordered sequence of characters, enclosed in **double quotation marks**.
It is part of the Standard Library.
You need to include the **<string>** library to use the string data type. Alternatively, you can use a library that includes the string library.

```
#include <string>
using namespace std;
int main() {
 string a = "I am learning C++";
 return 0;
}
```

Characters
A **char** variable holds a 1-byte integer. However, instead of interpreting the value of the **char** as an integer, the value of a char variable is typically interpreted as an ASCII character.
A character is enclosed between **single quotes** (such as 'a', 'b', etc).
For example:

```
char test = 'S';
```

Booleans
Boolean variables only have two possible values: **true** (1) and **false** (0).
To declare a boolean variable, we use the keyword **bool**.

```
bool online = false;
bool logged_in = true;
```

Arrays
An **array** is used to store a collection of data, but it may be useful to think of an array as a collection of variables that are all of the **same type**.
Instead of declaring multiple variables and storing individual values, you can declare a single array to store all the values.
When declaring an array, specify its element types, as well as the number of elements it will hold.
For example:

```
int a[5];
```

In the example above, variable **a** was declared as an array of five integer values [specified in square brackets].
You can initialize the array by specifying the values it holds:

```
int b[5] = {11, 45, 62, 70, 88};
```

The values are provided in a **comma** separated list, enclosed in **{curly braces}**.

Initializing Arrays

If you omit the size of the array, an array just big enough to hold the initialization is created.
For example:

```
int b[] = {11, 45, 62, 70, 88};
```

This creates an identical array to the one created in the previous example.
Each element, or member, of the array has an **index**, which pinpoints the element's specific position.

Arrays in Calculations

The following code creates a program that uses a **for** loop to calculate the sum of all elements of an array.

```
int arr[] = {11, 35, 62, 555, 989};
int sum = 0;
for (int x = 0; x < 5; x++) {
 sum += arr[x];
}
cout << sum << endl;
//Outputs 1652
```

Pointers

Every variable is a **memory** location, which has its **address** defined.
That address can be accessed using the **ampersand (&)** operator (also called the address-of operator), which denotes an **address in memory**.
For example:

```
int score = 5;
cout << &score << endl;
//Outputs "0x29fee8"
```

Pointers

A **pointer** is a variable, with the address of another variable as its value.
In C++, pointers help make certain tasks easier to perform. Other tasks, such as dynamic memory allocation, cannot be performed without using pointers.
All pointers share the same data type - a long **hexadecimal** number that represents a memory address.

Pointers

A **pointer** is a variable, and like any other variable, it must be declared before you can work with it.

The **asterisk** sign is used to declare a pointer (the same asterisk that you use for multiplication), however, in this statement the asterisk is being used to designate a variable as a pointer.
Following are valid pointer declarations:

```
int *ip;  // pointer to an integer
double *dp;   // pointer to a double
float *fp;  // pointer to a float
char *ch;  // pointer to a character
```

Just like with variables, we give the pointers a name and define the type, to which the pointer points to.

Using Pointers

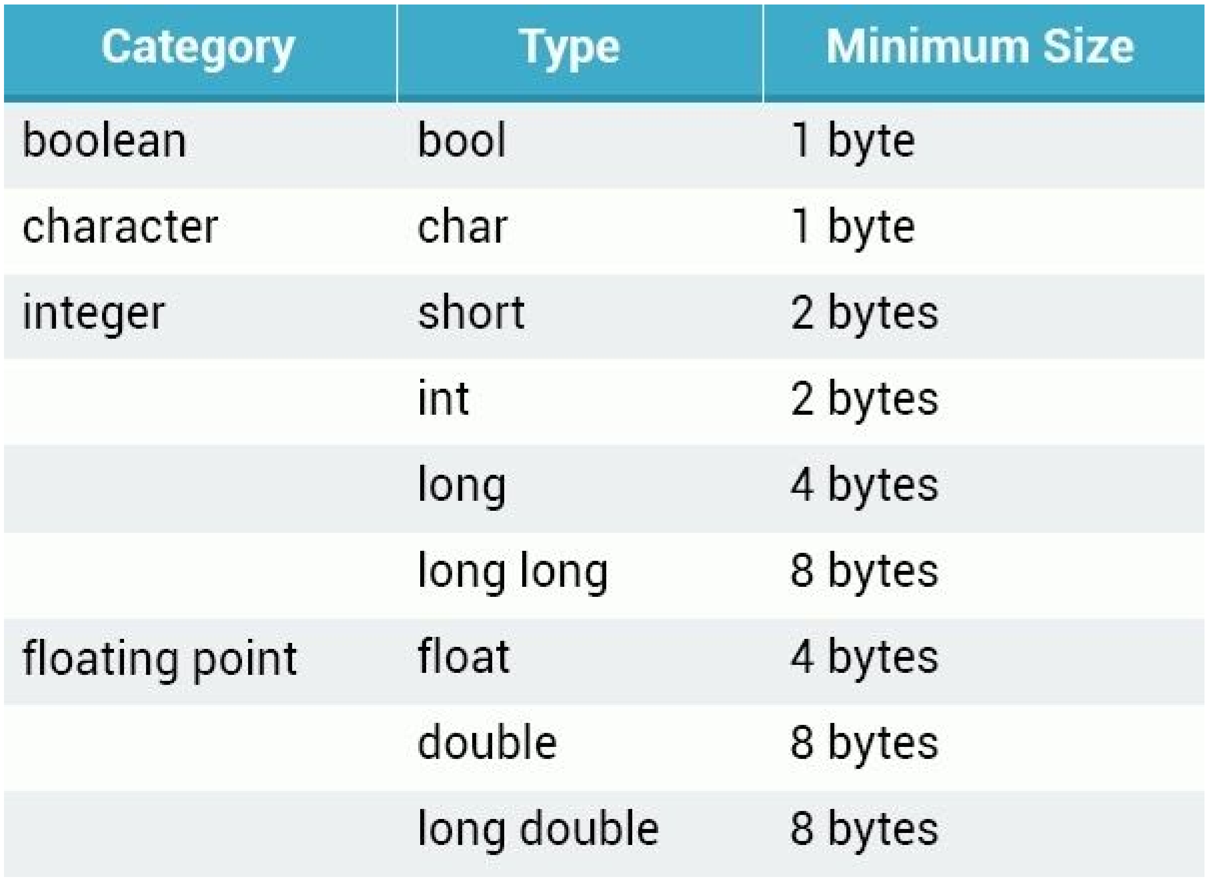
Here, we assign the address of a variable to the pointer.

```
int score = 5;
int *scorePtr;
scorePtr = &score;
cout << scorePtr << endl;

//Outputs "0x29fee8"
```

sizeof

While the size allocated for varying data types depends on the architecture of the computer you use to run your programs, C++ does guarantee a minimum size for the basic data types:



Category	Type	Minimum Size
boolean	bool	1 byte
character	char	1 byte
integer	short	2 bytes
	int	2 bytes
	long	4 bytes
	long long	8 bytes
floating point	float	4 bytes
	double	8 bytes
	long double	8 bytes

The **sizeof** operator can be used to get a variable or data type's size, in bytes.
Syntax:

```
sizeof (data type)
```

Size of an Array
The C++ **sizeof** operator is also used to determine the size of an **array**.
For example:

```
double myArr[10];
cout << sizeof(myArr) << endl;
//Outputs 80
```

Functions
A **function** is a group of statements that perform a particular task.
You may define your own functions in C++.

Using functions can have many advantages, including the following:
- You can reuse the code within a function.
- You can easily test individual functions.
- If it's necessary to make any code modifications, you can make modifications within a single function, without altering the program structure.

- You can use the same function for different inputs.

The Return Type

The **main** function takes the following general form:

```
int main()
{
 // some code
 return 0;
}
```

A function's **return type** is declared before its name. In the example above, the return type is **int**, which indicates that the function returns an integer value.
Occasionally, a function will perform the desired operations without returning a value. Such functions are defined with the keyword **void**.

Defining a Function

Define a C++ function using the following syntax:

```
return_type function_name( parameter list )
{
  body of the function
}
```

return-type: Data type of the value returned by the function.
function name: Name of the function.
parameters: When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function.
body of the function: A collection of statements defining what the function does.

Defining a Function

As an example, let's define a function that does not return a value, and just prints a line of text to the screen.

```
void printSomething()
{
 cout << "Hi there!";
}
```

Our function, entitled **printSomething**, returns **void**, and has no parameters.
Now, we can use our function in **main()**.

```
int main()
{
  printSomething();

  return 0;
}
```

You must declare a function prior to calling it.

For example:
```
#include <iostream>
using namespace std;

void printSomething() {
 cout << "Hi there!";
}
int main() {
 printSomething();

 return 0;
}
```

Multiple Parameters
You can define as many parameters as you want for your functions, by separating them with **commas**.

Let's create a simple function that returns the sum of two parameters.
```
int addNumbers(int x, int y) {
// code goes here
}
```

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.
```
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.

Overloading
Function **overloading** allows to create multiple functions with the **same name**, so long as they have different parameters.

For example, you might need a **printNumber()** function that prints the value of its parameter.
```
void printNumber(int a) {
 cout << a;
}
```

This is effective with **integer** arguments only. Overloading it will make it available for other types, such as **floats**.

```
void printNumber(float a) {
 cout << a;
}
```

Function Overloading

You **can not** overload function declarations that differ only by **return** type.
The following declaration results in an error.

```
int printName(int a) { }
float printName(int b) { }
double printName(int c) { }
```

Recursion

A **recursive function** in C++ is a function that calls itself.

To demonstrate recursion, let's create a program to calculate a number's **factorial**.
In mathematics, the term factorial refers to the product of all positive integers that are less than or equal to a specific non-negative integer (n). The factorial of **n** is denoted as **n!**
For example:

```
4! = 4 * 3 * 2 * 1 = 24
```

Let's define our function:

```
int factorial(int n) {
 if (n==1) {
   return 1;
 }
 else {
   return n * factorial(n-1);
 }
}
```

The if statement defines the exit condition. In this case, it's when n equals one, return 1 (the factorial of one is one).
We placed the recursive function call in the else statement, which returns n multiplied by the factorial of n-1.
For example, if you call the factorial function with the argument 4, it will execute as follows:
return 4 * factorial(3), which is 4*3*factorial(2), which is 4*3*2*factorial(1), which is 4*3*2*1.

Function Arguments

There are two ways to pass arguments to a function as the function is being called.

By value: This method copies the argument's actual value into the function's formal parameter. Here, we can make changes to the parameter within the function without having any effect on the argument.

By reference: This method copies the argument's reference into the formal parameter. Within the function, the reference is used to access the actual argument used in the call. This means that any change made to the parameter affects the argument.

Passing by Reference

Pass-by-reference copies an argument's address into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

To pass the value by reference, argument **pointers** are passed to the functions just like any other value.

```
void myFunc(int *x) {
 *x = 100;
}
int main() {
 int var = 20;
 myFunc(&var);
 cout << var;
}
// Outputs 100
```