→ C++ type String :-

C++ type string comes with STL <string>. It also provide many useful and easy to use functions.

Unlike C type string, C++ type String can be declared and initialize as integer type.

Also we can compare two C++ strings by using conditional operators like (==, !=, <, <=, >, >=).

Declaration and initialisation :-

```
String string_name;
String_name = "Akash";
OR, Cin>> String_name;
```

Also while ~~cin~~ using string, we don't need to think about size as it provide dynamic memory allocation.

NOTE:- If you have declared a string and using Cin to take input, it is recommended to use getline( ):

ex-

```
String str;
getline(cin, str);
```

<String> provides many functions to work, we will learn about some of them:-

- length( ):

→ Used to give length of string.

```
cout << str.length();
int n = str.length();
```

- begin( ):

→ Returns pointer to the first character of string.

```
string :: iterator it;
it = str.begin();
cout << *it;
```

- end()

→ Returns pointer to a special element end which denotes that end of string has come.

```
string :: iterator it;
it = str.end();
cout << *(it-1);
```

- Size();

→ Returns the no. of characters in the string. same as using length().

```
int s = str.size();
```

- clear();

→ Clear all characters in the string, thus makes it empty.

```
str.clear();
```

- empty();

→ Checks for wheather a string is

empty or not and return boolean values.

True if empty.
False if not empty.

```
if (str.empty())
    cout << "String is empty \n";
```

- insert();

→ Insert another string in one string at certain position as given.

```
str1.insert(6, str2);
```

position → index at which str2 will be inserted in str1.

- push_back()

→ Append a character to end of string.

```
String str = "abc";
str.push_back('d');
cout << str << endl;
```

- pop_back()

→ Removes a character from the end of string.

```
String str = "abcd";
str.pop_back();
cout<< str <<endl;
```

- find()

→ Finds occurence of one string in another. If string is found, the position/index of occurance is returned else string::npos is returned (indicating not-found):

```
String s1 = "Anvi";
String s2 = "nvi";
String s3 = "abc";

cout<< s1.find(s2)<<endl;
cout<<s1.find(s3)
if (s1.find(s3) == string::npos)
cout<<"string" << s3 << "not found\n";
```

· substr();

→ Generates substring from given string.

takes two input arguments

1st → position from where to cut
2nd → no of characters to cut.

If 2nd arguments is not given it will go to till end.

String S = "STLCPPAnuj";

Cout<< s.substr(3, 3)<<endl;

Cout <<s.substr(3) <<end;

Output :-

CPP
CPPAnuj

NOTE :- To concatinate an string to other we can only use '+' operator and it will do so.

```
string Str = "Anoj";
Str = Str + "kumar";
cout << Str << endl;
cout << Str.size();
```

Output :-

```
Anojkumar
9
```

Traversing through string :-

```
for(int i=0; i<str.length(); i++)
    cout << str[i];
```

```
for(auto x : str)
    cout << x << endl;
```

```
for(it = str.begin(); it != str.end(); it++)
    cout << *it;
//here it is an iterator.
```

→ References / Reference Variables:-

They are ~~created are~~ made for existing variables.

Reference of a given variable is nothing but a name change. If you something with one it will happen with another one ~~too~~.

They are helpful when we have to pass arguments. If we pass arguments as normal it makes copy of actual parameter, but when we pass arguments via reference it will not create copy of actual parameters but actuall actual & formal parameters are same. It will avoid making copy in the case of big values.

Also if we pass values via references we can change the values of actual variables, but we can't do if we pass it as normal variables.

If you are passing values to a function using references and don't want to change their values but only wanted to not make extra copy of variable, you can use 'const'.

ex:- Some examples of reference values

```
int main(){
    vector<int> vect{10,20,30,40};
        for(int &x : vect)
            x=x+5;
        for(int x : vect)
            cout<<x<<" ";

        return 0;
}
```

output 15 25 35 45

This coad works fine only if we have refersece in first loop we directly write it as

```
for(int x: vect)
    x=x+5;
```

and print it as we have done as

In second loop it will print

10 20 30 40

without modified as

while chaing values if we have not passed 'x' as reference it will make copy of ~~vectex~~ vect and change it, but it will not effect original one but if we have passed x as reference it will also change original one without making copy of it.

It seems more useful if your vector is big and have data more than thousand.

NOTE:- One reference variable can only refer to that same initial variable, it can't refer to other variable later in programme like pointer. Also reference of a variable must declared and initialize at same time, so, that variable whose reference you want to make must declared earlier.

ex- int x;
int

```
int x;
int &y=x;
```
right

```
int &y=x;
int x;
```
wrong

And

```
int x;
int &y=x;
int z;
int &y=z;
```
→ wrong, it can't.

→ Pointer :-

As we know pointer is a special variable which stores addresses of other variable.

declaration :-

```
int *ptr;
```

- `int` → Data type of variable whose address is going to stored in this pointer
- `ptr` → pointer variable name;

If you use sizeof() function of pointer variable of different type, you will get same value

as data type of pointer written before pointer variable is not the data type of pointer variable itself, it is data type of variable whose address is going to store in this variable(pointer).

But if you use sizeof() function on dereferencing a pointer you will get size of variable whose address is stored in pointer.

ex -

```
int main()
{
    int * ptr;
    char * ptr_c;
    double * ptr_d;

    cout<< sizeof(ptr) <<" "<< sizeof(ptr_c)
        <<" "<< sizeof(ptr_d) <<endl;

    cout<<sizeof(*ptr)<<" "<<sizeof(*ptr_c)
        <<" "<<sizeof(*ptr_d)<<endl;
    return 0;
}
```

output

8 8 8 // (Size of pointer is decided by // Compiler)

4 1 8

NEVER DEREFRENCE AN UNEINITIONALIZE POINTER.

Applications of pointers:-

- Changing values of passed parameters

```
void swap (int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main()
{
    int x=10, y=20

    swap(&x, &y);
    cout << x << " " << y << "\n";
    return 0;
}
```

Pointer airthmetic:-
If a pointer is declared as
int arr[5]; int * ptr;
ptr = arr;
ptr will store add of first index of arr.
now,
If you do *(ptr+1)
it will start to point to next index as
ptr+1 will add 1x(size of type of pointer) to that intial add.
So,
after dereferencing (ptr+1) you will get value in next index
and
*(ptr+2) will give value of 2nd index (indexing start from 0).

- passing large array/vectors to a function to avoid coping of them same as using references.

```
void processVec(vector<int> & v)
{
    //code;
}
```

```
int main()
{
    vector<int> v;
    for(int i=0; i<1000; i++)
        v.push_back(i);
    Processvec(&v);
    return 0;
}
```

We can also use references for this.

- In Dynamic memory allocation.
- Implimenting data structures like linked list, BST, Tree etc.

We can't use references in this case as we know if a reference is initialize it can't reinitialize but ~~it is~~ this problem is not with pointer, we can reinitialize them to point to different memory location.

Also pointer give more hold on memory and references in C++ are not as powerful as pointers so we need pointers along with references.

- To return multiple values:-

What if you have given two values given a and b and you have to do two works on them and get result, you have to make two functions as one function can only return one value.

But using pointers you can get ans of these two work from one function

ex.-

```
void docalculation(int a, int b,
                   int *add,
                   int * multi)
{
    *add = a + b;
    *multi = a * b;
}

int main()
{
    int x=10;
    int y = 20;
    int sum=0;
    int multiply=1;
```

docalculation(x, y, &sum, &multiply);

cout << sum << " " << multiply << endl;

return 0;

}

NOTE:- If you pass any array to a function it is always passed as pointer i.e. it is passed as pointer holding address of first element of array.

So, if you use sizeof() function on that passed array you will get size of pointer not array, So, NEVER USE SIZEOF() ON AN ARRAY WHICH IS PASSED TO A FUNCTION.

Because of this we pass size of array with array in function.

Also to traverse through array we use

arr[i]

but compiler always converts it in form of pointer as

*(ptr+i);

So, you can use any of these definition to traverse through array compiler always uses second one.

++ *ptr;

Q. Guess the output :-

i> int main(){
int arr[] = {10, 20};
int *ptr = arr;

++ *ptr;

Cout<<arr[0]<<" "<<arr[1]<<" "<<*ptr;
return 0;
}

→

output 11 20 11

++ *ptr;

This line has two operators working here ++ (pre increment) and * (dereference)

both have same precedence in operator precedence so we have to go with associativity.

Since associativity is right to left it will follow as

++ (*ptr)

So, value at 10 will be changed to 11.

(ii) int main(){
int arr[] = {10,20};
int *ptr = arr;

cout<< *++ptr <<" ";

cout<<arr[0] <<" "<<arr[1] <<" "
<<*ptr;

return 0;
}

→
output

20 10 20 20

Since, in *++ptr, it is pre-increment so effect happen at that same time.

(iii) int main(){
int arr[] = {10,20};
int *ptr = arr;
*ptr++;
cout<< arr[0] <<" "<<arr[i]<<" "
<<*ptr;

return 0;
}

→ output 10 20 20

post increment operator has higher precedence than pre increment operator and *.

So it will be evaluated as

*(ptr++)

iv) int main()
{
int arr[] = {10, 20};
int *ptr = arr;
cout << *ptr++ << " ";
cout << arr[0] << " " << arr[1] << " " << *ptr;
return 0;
}

→

output :-

10 10 20 20

As *ptr++ is evalues at

*(ptr++)

So ptr is stated to point to 20 but since it is post increment its effect will from next time.