Strings (1)

Strings is a collection of characters. It is similar to arrays, but has more functionalities than arrays. That means, all the array operations are valid on a string, but you have additional methods available as well.

```
#include<string> // Not needed if you are using bits/stdc++.h

using namespace std;

int main() {
   string s = "Hello World!";
   return 0;
}
```

Input and Output

For String I/O, you can use `cin` and `cout` statements.

```
#include<string>
#include<iostream>

using namespace std;

int main() {
    string s;
    cin>>s;
    cout<<"The string you entered is: "<<s<<endl;
    return 0;
}
```

Accessing characters

String eventually stores characters as an array. So like arrays, you can loop on the characters of string and access them by index. Or you can use the `for-each` loop as well

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

for(char x: s) {
   cout<<x<<" ";
}
```

Updating characters

Similar to arrays, you can update character at any index by `=` operator. Strings in C++ are mutable unlike some programming languages like Java, Python

```
string s = "Hellw";
s[4] = 'o';
```

String methods

Besides array-like operations, you can perform many other operations on Strings using some predefined methods. The following are examples of few commonly used ones:

```
string s = "Hello World";

// length
s.length();

// concatenation
string a = "Hello ", b = "World";
string res = a + b; // "Hello World"

// Equality
string a = "Test", b = "Test";
bool res = a == b; // true
a.compare(b); // = 0 if equal, < 0 is a is smaller, > 0 if a is larger

// Substring extraction
s.substr(1, 4); //ello, first argument is starting index and second is length
```

```
// Substring search
s.find("orl"); // First index where "orl" is present in string. If not found returns

// Replacing substring
s.replace(0, 3, "Hi"); // Replaces 3 characters from index 0 with "Hi"

// Inserting substring
s.insert(6, "new "); // Hello new World, Inserts the substring at index 6

// Iterators
s.begin() and s.end();

// Erase
s.erase(2, 3); // Erases 3 characters starting from index 2
s.erase(s.begin() + 1, s.begin() + 5); // Erases all characters in between indices [
```

Pointers (1)

Memory Allocation

1.Heap-> Dynamic (free memory)
2.Stack->Local variables and functions
3.Global(static)->global variables
4.Program code

💡 Statically declared arrays (i.e arrays whose size is fixed and known at compile time) are stored in Stack memory

Variables whose size is determined at run time i.e dynamic sized variables are stored in heap memory. For example, dynamic sized arrays are stored in heap memory. Basically, whatever is created at run time uses heap memory. And the variables whose size are fixed and known at compile time are created in stack memory.

What are pointers?

Pointers store the address of a variable. The address it stores can be either of a stack memory or heap memory.

```
int x = 10; // We define an integer variable. It gets assigned 4 bytes in stack memo
int *ptr = &x; // We create a pointer on variable x. The pointer (ptr) stores the ad
```

💡 `&` is the addressOf operator when used on the **right hand** of an `=` sign and can be used for defining references if used on left hand side or in function arguments.

Similarly, `*` operator generally is used as valueOf operator except when it is declared with a data type in front of it, in that case it is for declaring a pointer variable.

💡 In C++, references just means aliases (alternative name) to existing variables. They do not occupy any memory.

We can directly modify values of variables using pointers. For example,

```
int x = 10;
int *ptr = &x; // We create a pointer on x
*ptr = 20; // Updates the value of x to 20.
```

Copy

Note, in the above example the usage of `*` operator. In the second line we have `int *ptr = &x;` . This tells us to create a pointer and assign it the value of address of variable x.

In Line 3, we write `*ptr = 20` . As mentioned above, `*` without a data type before it is considered as valueOf operator. So it is read as:

Set valueOf variable pointed by pointer ptr (i.e x) to 20.

Data type of pointer

Pointers themselves are always integers / longs as they hold memory location. They generally occupy a size of 8 bytes (or 4 bytes for 32-bit system)

However, we might see the different types of pointers are defined depending on the data type whose address it holds.

For example we define `int *ptr` for pointers holding address of integer variables. Similarly, we define `double *ptr` for pointers holding address of double variables. But the type of the pointer only signifies what is the data type of the variable whose address it holds and does not have anything to do with its memory allocation.

Pointer of Pointer

It is a pointer to some pointer, i.e stores address of a variable which is a pointer.

```
int x = 10;
int *ptr = &x; // ptr stores address of x
int *ptr1 = &ptr; // ptr1 stores address of ptr
```

Copy

```
cout<< *ptr1; // Prints address of x
cout<< **ptr1; // Prints value of x
```

Dynamic allocation (1)

Dynamic arrays

One way to create dynamic arrays is using pointers. Dynamic allocation allocates memory in the **heap space.**

As mentioned, pointers holds the address of a variable. If we can occupy consecutive memory locations, we can use pointers to iterate on them without additional help. How?

The pointer holds the base address / first element address. The type of pointer determines the type of variable it points to, for example integer pointers are pointing to integer variables. So from that we know the memory each variable uses (Ex 4 bytes for integers). So we can just increment the base address by 4 bytes to get the address of the next element. This is called **pointer arithmetic**.

```
int *ptr;

// Let's say ptr stores 2000 which is address of a variable
// Data type of the variable is integer - 4 bytes

ptr++; // ptr becomes 2004 i.e address of the next element
ptr--; // ptr becomes 1996 i.e address of previous element
```

```
ptr = ptr + sizeof(type)
```

Creating a dynamic array

```
int *arr;
int n;
cin>>n;
arr = new int[n];

// Now we can do normal array operations

for(int i=0; i<n; i++) cin>>arr[i];
```

Let's understand what happens when we do `arr[i]` .

Internally it translates into this:

`arr[i] = *(arr + i)`

Now by pointer arithmetic, it should skip by the size of the data type. So,

```
arr[i] = *(arr + i * sizeof(int))
```

The name of the array (i.e arr) holds the **base address.**

Function call argumens (1)

Call by value

```
int f(int x, int y) {
  x = 10;
  y = 20;
}

int a = 30;
```

```
int b = 40;

f(a, b);

cout<<a<<" "<< b; // Prints 30 40
```

The above code is an example of call by value approach. Here, 2 new variables `x` and `y` gets created. So when we change the value of x and y, it does not affect a and b.

💡 Whenever you use call by value, new variable is created. So be mindful of this especially during recursions.

Call by reference

```
int f(int &x, int &y) {
  x = 10;
  y = 20;
}

int a = 30, b = 40;
f(a, b);

cout<<a<<" "<<b; // Prints 10 20
```

The above code is an example of call by reference approach. As mentioned before, `&` means creating a reference which is an alias / alternate name for the variable. Here x and y just becomes alternate names for the variables a and b respectively. They don't have any memory allocation of their own. So setting `x = 10` is equivalent to writing `a = 10`.

Passing pointers

```
int f(int *x, int *y) {
  *x = 10;
  *y = 20;
}

int a = 30, b = 40;
f(&a, &b);
```

```
cout<<a<<" "<<b; // Prints 10 20
```

Here, we are passing address of a and b. So x holds address of a and y holds address of b.

Now we are directly making the change in the address, so value of a and b are updated.

Classes and objects (1)

C++ is an object oriented language. The 2 fundamental concepts are:

1. Classes - A class is a blueprint or template for creating objects. It defines the properties (data members) and behaviors (member functions) that objects of the class will have. Think of classes as user defined types.
2. Objects - An object is an instance of a class. It represents a specific realization of the class blueprint, with its own set of data members and methods.

Example, consider you are building a Student management system. Now, for each student you want to store their name, roll no, address and CGPA. One approach is creating different lists for storing each of them. However, that makes things difficult to manage. The better approach is to create a type Student that holds all these values and also exposes methods that can mutate these values and perform certain operations for each student. This idea is called **Encapsulation**.

```
class Student {
   string name;
   string address;
   int roll_no;
   int attendance;
   float cgpa;

   public:
   void mark_attendance() {
          this.attendance += 1;
   }
```

```
}

Student ram = new Student();
```

this keyword

`this` keyword is used inside a class, that is used to represent the instance of the class calling the method. For example, if we have 2 students, Ram and Shyam and we call `ram.mark_attendance()` then `this` keyword represents the object ram.

Constructors

Constructors are the first method that is called when an object of the class is created. It is for assigning initial values to the data members. There are 3 types of constructors:

1. Default constructor - This is by default present, even if we don't manually define a constructor. It does not take any argument and assigns default values to the data members.
2. Parameterised constructor - This takes parameters of the initial values and assigns the data members these values.
3. Copy constructor - Copies the data member initial values from another object.

```
class Student {
   string name;
   string address;
   int roll_no;
   int attendance;
   float cgpa;

   public:
   // default constructor
   Student() {
      this.name = "";
      this.address = "";
      this.roll_no = 0;
      this.attendance = 0;
      this.cgpa = 0.0;
   }

   // Parameterised constructor
   Student(string name, string address, int roll, float cgpa) {
       this.name = name;
       this.address = address;
```

```
        this.roll_no = roll;
        this.attendance = 0;
        this.cgpa = cgpa;
    }

    // Copy constructor
    Student(Student &s) {
        this.name = s.name;
        this.address = s.address;
        this.roll_no = s.roll_no;
        this.attendance = 0;
        this.cgpa = s.cgpa;
    }

    void mark_attendance() {
            this.attendance += 1;
    }
}
```

Q - Why do we pass reference in copy constructor?

Access specifiers

Control the visibility of class members (variables and functions) to other parts of the program

1. **Private**: Only methods within the classes can access these data members and member functions.
2. **Protected**: Only methods within the class and in derived class can access these.
3. **Public**: Accessible from anywhere, inside or outside the class.

Q - Where should the constructor of a class be?

We generally will not need concepts like Inheritance, Polymorphism, Data hiding, Abstractions, Friend classes for DSA so we will skip them. However, these are important Software Engineering concepts and you should definitely learn them to be able to ship good quality code.

When will we use classes for DSA?

1. **Defining new data type for making the task easier and code readable**
   Sometimes we would come across the need of creating a custom data type. For example, let's

say we want to store 2 or 3 values for each array element. An example can be a LinkedList. In LinkedList, a node stores the value of the current node and also a pointer to the next node. We can do this without classes as well, but the code quality takes a hit, and also it becomes difficult to manage. With classes, we can do like this:

```
class Node {
   int data;
   Node *next; // A pointer to the next node
}
```

1. **Encapsulating some common functionalities to make the code cleaner**
   Sometimes, when our code becomes significantly bigger, and let's say we are using some utility data structure to solve a part of it (like graph), we can encapsulate it into a class and expose some methods that the rest of the code can interact with. Let's say, we need an utility that calculates minimum distance between 2 nodes, we can do something like this:

```
class Graph {
   // data members needed for graph representation

   public:
   int calculate_minimum_distance(int node1, int node2) {}
}
```

1. **Avoiding global variables**
   In some problems like graph based, DP based and some other problems, we might need to maintain some global variables. Generally, it is okay wrt solving DSA problems, but treated as a bad pattern in interviews. Ideally in these cases, you should encapsulate the variables and the functions in classes. The global variables becomes data members, and the functions using them becomes methods. For example,

```
bool found = false; // Global variable

void f() {
   if(some condition) {
           found = true;
           return;
   }
}

// Instead we can do

class Solve {
  bool found;

  public:
```

```
    Solve() {
        this.found = false;
    }

    void f() {
            if (some condition) {
                    found = true;
                    return;
            }
    }
}
```