Math is fun.com is game web site for maths students

**C++:**

# include means we are importing libraries(header files)

<library name>

cout<<””; use to print data

puts(“”); also use to print data

cin>>var use to take input from user, input will save in variable

() called as paranthesis

[] called as brackets/square brackets

{} calles as braces / curley braces

Method to Take data from user: getline(waytotakedata,variableToStoreData) like getline(cin,number)

Using namespace std:

Std mean standard, using namespace std means every line of code after this will be coming from namespace standard file.

It is a way to group named entities(like variable, function) under single scope.

%d is known as place holder it takes the value.

Like printf(“the calculated value is %d,2+6);

Output: the calculated value is 8

**CPP Identifiers:**

cpp is case sensitive.

If upper and lower case then separate memory locations will be allocated.

Numbers should be at the end or between of variable name, it should not be in start.

\_ can be used to start a variable.\_ mean private character, two \_ at start means reserved keywords

Cannot used reserved keywords(75)

Non-latin(urdu) can be used but avoid to use it.

Use maximum 37 characters I keyword.

**Character literals:**

**\n:** new line

**\\:**  \ character

**\’:** ‘character

**\”:** “character

**\?:** ? character

**\a:** alert or bell

**\b:** backspace

**\f:** form feed

**\r:** carriage return

**\t:** horizontal tab

**\v:** vertical tab

**\ooo:** octal number one to three digits

**\xhhh:** hexadecimal number of one or more dogits

**Primitive Data types:**

Bool,

char,

int,

float, => double => long double => it shows more numbers after decimal according to float<double<long double

double(it shows float value but many numbers after point),

void(null or nothing)

**non-primitive data types:**

**string**: import string before using bcz it is non-primitive e.g. #include <string>

**note:** declaration means just make variable, initializing means assigning value to the variable.

Time: 1:17:50

**Pointer:**

Pointers points to a memory location, declate with datatype then use “\*” and assign name to pointer.

e.g. int \*khanopointer;

then we have to point to some variable

e.g,

int a=50;

int \*khanopointer;

. khanopointer=&a; & amperson, it points to memory address to a, so khanopointer wll point

a’s memory address;

if we want to print memory address then we simply print . khanopointer; if we want to data then we use \*khanopointer;

like printf(\*khanopointer)

if want to use specifier then we will use %p

we can assign references and assign variable to it;

after then when we assign a new value to the reference that we decled it will change the value of the variable

e.g.

**int** a = 50;

**int** &myref =a;

myref=70000;

cout<<"the value printed from reference is"<<myref;

output will be 70000 instead of 50;

**Arrays:**

It is contigeous memory location which store multiple values of same data types.

**int** myArr[6]={5,3,2,3,4,5};

cout<<myArr[0];

it will print 5

if we just print myArr; it will print just memory address;

we can change the 1st index value by using pointer,

like \*myArr=5; it will replace 1st value with 5;

if we want to access and change the next index value then we simply create new pointer, point it to our array,

and do increment to it and assign a new value, it will change the value of index according to increment.

e.g.

**int** myArr[6]={5,3,2,3,4,5};

**int** \*arrRef=myArr;

arrRef++;

\*arrRef=80;

cout<<myArr[1]<<"\n";

arrRef++;

\*arrRef=100;

cout<<myArr[2]<<"\n";

output: 80,100

**Integers: Note:**we can check size of any data type by sung size(); method, give value in byte

The type of integers holds different data size.

1 byte=8 bits

The size of bits depends on the machine and os in which it is being used we the short int is the smallest one and the long is longest one.

Short int< int<long int< long long int

All of them store non-decimal math value

**short int: 16bits**

**int: 32 bits**

if we assign vakue to int by 0b00101110, it will convert binary code into number, before binary code we have to write ob

**long int: 64 bits**

we can simple use just long

**long long int: 64bits or more according to OS**

we can simple use just long long

if we want to fix bit size for all systems then we can use library <cstdint> to specify bits. We can use it like int16\_t a=15; now int is of maximum 16 bits,(note: if are just declaring and not going to assign value during declaration then we can use uint16\_t ) it will use at the place of long long,short, long etc

**Conditionals:**

**If-else:**

      int a= 7;

       if (a>=18)

       {

       cout<<"congratulation! your are eligible for marriage";

       }else{

        int remaining\_year=18-a;

        cout<<"sorry! you have to wait "<<remaining\_year<<" years for marriage";

       }

**Ternary operators:**

int age = 17;

    age >= 18 ? cout << "you are eligible for marriage" : cout << "sorry you have to wait for " << 18 - age << " years for marriage";

**Switch case:**

int rating = 2;

    switch (rating)

    {

    case 1:

        cout << "so bad to know you don't like our product" << "";

        break;

    case 2:

        cout << "so bad to know you don't like our product" << "";

        break;

    case 3:

        cout << "your feedback will be good to improve our product" << "";

        break;

    case 4:

        cout << "good to know you like our product we will be improve our product according to your feedback" << "";

        break;

    default:

        cout << "Good hogya, done hogya te chass agae" << "";

        break;

    }

**Loops:**

**Forloop:**

string str="kamboh";

for (int i = 0; i < str.length(); i++)

{

   std::cout << str[i] << std::endl;

}

**While loop:**

    int i=str.length();

    while (i!=-1)

    {

      cout << str[i]<<"\n";

       i--;

    }

**Dowhile loop:**

int i=9;

do

{

if (i>=str.length())

{

std::cout << "number is bigger than string length" << std::endl;

}else{

std::cout << str[i] <<std::endl;

}

i++;

} while (i<str.length());

**For Range:**

it act as forEach loop in js

works with arrays and char data types

for\_range loop: it act like for each loop

**int** arr[]={2,3,4,5};

for (**int** i : arr)

{

    std::cout << i\*2 << std::endl;

}

**Loop with pointers:**

  char ch[]="ali\_akbar";

for (char \*mypntr = ch; \*mypntr!=0; mypntr++)

{

*/\* code \*/*

    std::cout << \*mypntr << std::endl;

}

**Try\_Catch:**

When we are writing some that that may give some error like Api calling then it may crash the code so we handle the error by using try catch so other code will not effect by this error

in catch block we have to define datatype which we are going to accept. Like catch(int err){}

but if we don’nt know which type of error will come then we can use … like catch(…){}

in try block we through error of type that may cause error, after through the rest of the other code will not be executed so so should through at the end of block

**Example:**

**float** num = 5;

    try

    {

        cout << "1st code in try block\n";

        cout << "2nd code in try block\n";

        throw num;

        cout << "the code after throw error\n";

    }

    catch (**int** err)

    {

        std::cout << "the error cause by this num which should give value: " << num << std::endl;

    }

    catch (...)

    {

        std::cout << "all type of data error can be executed through this block" << std::endl;

    }

**Functions:**

Before start the function we have to define type which the function is going to return.

We have to also define the type of argument that we are going to receive.

If a function is defined with a datatype then it is necessary that It have to return that data type

If we have multiple function then all of the function should be called inside the main function

**Linker:**

When we import library, before compile the import library file with our code file compile together and make an object, the object goes to linker, linker take standard runtime library to make .exe executable file.

Object is a numeric binary code that our coputer architecture can only understand

**Qualifiers:**

Two types:

1. Modification qualifier

Allow to modify or not a value, like const

Types:

* Const: if value assigned with cost then we cannot modify it later
* Volatile: use to share one variable from one thread to another thread
* Mutable: it make variables mutable

1. Life duration qualifier

How much time or area variable is available to me

Types:

* Static:

Track entire like cycle of the variable.

When we declare variable with static keyword it will be available as global scope, mean available inside or out of any block

**Static int val=5;**

* Register: give suggestion to compiler to store information to register(use while hardware handeling)
* Extern: use to store something in outerlibrary which will be done at the time linker

**Operations: it act as short cut and code optimization**

**A++ postfix**

**++A prefix**

**+=, A+=B => A=A+B**

**Logical Operators:**

! not

Or ||

And &&

**Bitvice Operations: (easyonlineconverter.com** is a web can be used for conversion of bits**)**

**Converts** value into 0-1 binary form then compare it or do operations acordinf to commands then convert back into human readable form

**Memory leaks:**

when we create a code with new keyword then it means we are forcefully creating memory then it can cause memory leak so it is our responsibility to free up that space which we created forcefully.

We can delete that space by using delete [] pointer; synthax

**Struct:**

It is like blueprints use to make multiple object with same functionality and same type of datatype but different data according to different requirements.

Syntax for struct is :

Struct xyz{

String Name;

Int Age;

};

How to make objects from struct blue print:

Xyz aliData={“ali”,16};

We can also assign values by using dot notation or modify these values.

Like: aliData.name=”akbar”; aliData.age=16;

We can create multiple objects by using this struct blueprint.

**Note:** if we are using const with pointer then we are making that pointer address constant not its value, its value can be changed but pointer reference cannot be changed.

If we want to create a pointer which will refer to an object then the data type of that pointer should be the same blueprint from which we have created the object to which we want to refer the point.

After that we cannot directly change the value of the we have to use this syntax.

Pointer ->object=5;

We also have constructor in the struct which is use to accept data from object and assign values inside the struct or we can assign predefined values to it.

We can do it by using

Struct\_name(): data1=3,

**Enum:**

It save only integers type automatically, we set the permanent values in inters which can be used in whole file.

Sentax to describe;

**enum** Colour

{

    COLOR = 123,

    TYPE = 11,

    SIZE = 4

};

How to use:

Just use name to attribute that we set, its value can be used just by using its name.

Like: std::cout << "The Enum value: " << COLOR << std::endl;

Note: if we do not assign any number then by default the 1st attribute will get value of 0, and the value of next attributes will be the increment of previous. If we assign number to one attribute and the attributes is not initialized with number then then next attribute will get increment by one of previous attributes number.

We can also do type casting by using syntax

**enum** Colour:int64\_t {}

**Auto:**

If we don’t know the data type that in future we are going to use to store in a variable then we use auto, it will detect the datatype automatically and assign that datatype to variable.

Auto myname;

Name=”ali;

Name=5;

Name=true;

We use it mostly in API Calls.

dataId(var); is an object that tells the data type.

Like:

**int** API\_CALL(){

    return 10;

}

string API\_CALL2(){

    return "API";

}

**int** main()

{

**auto** API\_1=API\_CALL();

**auto** API\_2=API\_CALL2();

    std::cout << API\_1 << std::endl;

    std::cout << API\_2 << std::endl;

    return 0;

}

**Stack Memory:**

It is fastest and default memory. It is predefine size. When we assign variable it assign location in stack.

When we simply declare variables it goes in stack memory

**Heap Memory:**

It is also predefined memory but can be increase in size if needed.

To use heap memory we have to create a poiter and initialize it with new keyword with data type.

Like int \*myp= new int;

We can assign value to it by using \*myp=15;

Creation of heap memory can cause memory leak so we can avoid It by deleting the unused memory.

delete myp;

**pass\_by\_Value:**

pass by value mean we make a copy of a variable and give it to some one, if the copy changed then it will not effect the original value. So it is useless if we want to change value by passing value. So we use reference

**pass\_by\_Reference:**

In pass by reference we take the memory address of the original variable and send it to any where, if any modification occur then it will be reflected in the original variable.

We can do by two ways:

1. **Send reference as argument and accept as Pointer:**

**Example:**

**Sending as argument:**

**int** life=4;

    lifeVal(&life);

**Accepting Reference:**

**int** lifeVal(**int** **\***life){

    ++(\*life);

    return \*life; }

1. **Sending variable but Accecpting the Reference of variable:**

**Sending variable:**

**int** ref=18;

    refAccept(ref);

**Accepting as reference:**

**int** refAccept(**int** **&**ref){

++(ref);

    return 0;

}

**Header file/import:**

We make header file name.h which we can import in our main file which we import by using #include “filename”

**The template code is:**

#include <iostream>

#ifndef adder\_h

#define adder\_h

All the methods or variable will be define here

#endif *// !adder\_h*

**How to import and use:**

#include “filename”

Method\_from\_file(argumen1,argument 2)

**Example:**

**Header making:**

#ifndef adder\_h

#define adder\_h

int add(int a,int b){

    return a+b;}

#endif

**Use:**

#include <iostream>

#include "head.h"

std::cout << add(3,8) << std::endl;

**output:**

**11**

We can use multiple outside libraries to do different work like web development, server development, AI etc.

**Template:**

We use template to create our custom Data type, which can take any other data type

To do this we use this syntax template <typename xyz>, now xyz is our data type

We can use it just like xyz add (xyz a, xyz b){ return a+b}

**Functional Pointer:**

These pointer get the reference of a function and we can call function later anywhere.

**Example:**

**void** val()

{    std::cout << "Data from val" << std::endl;}

**Calling this function:**

**int** main(){

    void (\*val\_pointer)() = val;

    val\_pointer();

    return 0; }

**Null Pointer:**

We can pass NULL directly in CPP, if we do this it gives an error, so to avoid error we redfine the NULL by using

#define NULL 0

But by doing this we reassign value 0 which is not good, there is a difference b/w null and 0;

So we use null pointer, the nullptr keyword is used to solve this problem.

**Example:**

**void** nullfn(**int** **\***val){

printf("the value of nullpointer will be here %p",val); }

**int** main()

{

nullfn(nullptr);

    return 0; }

**Macros in cpp:**

It is like we are predefine variable and store any value and any predefined keyword to it.

Example if in reach function we do return 0, we can make a macro end and use at the place of return o

#define end return 0

Fn(){

end; the value from macros copied and pasted to this place while compilation

}

**Verdiatic: (…)**

it is like rest operator in js.

It can take multiple inputs

Template <typename… T>

Int main(T…){}

**Object Oriented Programming:**

class: it is a keyword to start classes, it has syntax class Name{ }

name of class is Capital letter

the default behaviour of class is private, so if we want to give access to methods or data then we have to make that data of method public:

to make object from the class we simply do write class name then our new object name;

**Example:**

**Defining Class:**

class Mydata

{

public:

    string myname = "Default name";

    void name()

    {

        std::cout << "My name is " << myname << std::endl;

    }};

**Using Class to make object and access data:**

int main(int argc, char const \*argv[])

{

    Mydata ali;

    ali.myname="Ali Akbar Khan";

   ali.name();

return 0;

}

**Setter:**

setter is a concept to change the private data.

In classes we cannot access private data outside the class, so we are not able to modify data, but the private data is accessible within the class, so we make a public method inside the class which have functionality to change the private data.

Usually we start such methods with setName, but without set things works same, using set is a good practice so we should do the same.

**Example:**

**Note:** the secret data should be start with the \_

**Making setter:**

**void** setData(**int** **&**data, string **&**password){

if (password=="aliakbar")

{

    \_secret\_key=data;

    std::cout << "the new sercret\_key is: "<<secret\_key << std::endl;

}else{

    std::cout << "Wrong password" << std::endl;

}}

**Using setter:**

string password = "aliakbar";

    int newKey=121314;

    ali.setData(newKey,password);

**Getters:**

Getter are just like setters but in getter we do not modify data we just return secreat data to object after some modification into secret data.

**Example:**

**Making method:**

    int getSecret(string &password)   {

        if (password == "aliakbar")   {

            return secret\_key;

        }else  {

            std::cout << "Wrong Password" << std::endl;

            return 0;

        }}

**Using that Method:**

 string password = "aliakbar";

 std::cout << ali.getSecret(password) << std::endl;

**Accessing class data outside the class without making any object:**

we just initialize a method in the class then write the code of that method outside the class.

**Syntax:**

**datatype** className::function(){}

**Example:**

Class data{

void punjabschool();

};

void Student::punjabschool(){

  std::cout << "message" << std::endl;

}

in**t** main(){

    Student ali;

    ali.punjabschool();

    return 0; }

we do this to make different component to arrange data, just like MVC method.

**Note:**

Constant values can access constant objects, but constant objects cannot accept constant values.

**Constructors:**

1. **Default constructor:**

Example:

1. **Parametric constructor:**
2. **Copy Constructor:**

**Destructor:** Invoke automatically and delete the previous constructor to freeup some space.

**Example of constructors and destructor:**

#include <iostream>

#include <string>

using namespace std;

class Phone

{

    string \_name = "mobile";

    string \_os = "ololo";

    int \_price = 0;

public:

    Phone(); *// default constructor, invoke authomatically at object creation*

    Phone(string name, string os, int price); *// parameterize constructor*

    Phone(Phone &); *// copy constructor, copy the data of another object of same type of class.*

    ~Phone(); *// destructor, delete the constructor automatically*

    void getData() { std::cout << \_os << std::endl; }

};

*// Defination of default constructor*

Phone::Phone() : \_name("Samsung"), \_os("andrios 14"), \_price(60000) { std::cout << "Default constructor is working" << std::endl; };

*// Defination of parametrize constructor*

Phone::Phone(string name, string os, int price) : \_name(name), \_os(os), \_price(price) { std::cout << "parametrize constructor is working\n"

                                                                                                  << std::endl; };

*// copy Constructor*

Phone::Phone(Phone &data) : \_name(), \_os(), \_price()

{

    \_name = "new" + data.\_name;

    \_os = "new" + data.\_os;

    \_price = data.\_price;

    printf("copy constructor\n");

};

*// destructor: it invoked automatically to free upspace*

Phone::~Phone() { printf("the destructor is working for %s\n", \_name.c\_str()); };

int main(int argc, char const \*argv[])

{

*// Default constructor*

*//  Phone samsung;*

*//  samsung.getData();*

*// Parametrize constructor usage*

    Phone oppo("A57", "Andriod 9", 40000);

    oppo.getData();

*// Copy Constructor*

    Phone nokia(oppo);

    nokia.getData();

    return 0;

}

**This keyword:**

It points or refer the object and its context in which it is created. It is also know as self-referencing pointer.

This->ali();

**Inheritance:**

**Base/parent Class:**

It is the main class in which we write our methods from scratch.

**Derived/child Class:**

The class made from the base class, that contains all methods of base class and that can have more methods than the base class and can override the base class methods as well.

**How to create Derived class:**

Class childClassName : public parentClassName{}

**How to use base methods in derives class with different name and functionality includes base functionality:**

newMthod():baseMethod(){};

if name is same in both then use simple name and re-write defination

**Friend class:**

If we want to access private methods or data of parent class into child class then we can give access to child class through inside of the parent class by using friend

In private we can use friend class childClassName;

**Syntax:**

Class childClassName : public parentClassName, anotherParent,AnotherPArent{}

**Multiple inheritance:**

We can make one derived class from multiple parent class just by using comma.

**Access of types of methods between classes and objects:**

****

**Polymorphism:**

**Virtual Funation:**

If we have same name of method with different defination, and if we make object with pointer, then it will point to the original class method, to avoid this we make that method virtual so if any other class object through pointer called then it will used that derived class method.

**Pure virtual function:**

If a function marked as virtual and do not have any body then it is called as pure virtual function.

**Smart pointers: (**search on microsoft website’s documentation for cpp->open memory url**)**

they are the wrapper around the real raw pointers

Smart pointers automatically delete memory allocation automatically to avoid the manually deletion of memory allocation

New keyword allocates a memory

Delete keyword delete that memory

As pointer points the memory address it can casue memory leakage.

**Types of Smart pointers:**

1. **Unique pointer:** these ends at the end of the scope and we cannot copy them. No need to delete manually as after ending the scope it delete automatically. Sharing of memory references is not allowed bcz it cause memory leakage

**Example:**

**class** User{

**public:**

User(){std::cout << "constructor" << std::endl;};

~User(){std::cout << "destructor" << std::endl;};

**void** testFunc(){std::cout << "I am a test func" << std::endl;};

};

**int** main() {

    {

unique\_ptr <User> ali=make\_unique<User>();

ali->testFunc();

std::cout << "inside the scope" << std::endl;

    }

    std::cout << "outside the scope" << std::endl;

    return 0;

}

1. **Share pointer:**

We can share memory reference and it counts the pointers which are sharing the same memory.

When all the shared pointers complete their job then the memory will be freed automatically.

**Note:**This take memory of stack, if we use new we use heap memory which take extra space and have to delete manually

Example:

*{*

*shared\_ptr <User>ali=make\_shared<User>();*

*shared\_ptr <User>akbar=ali;*

*std::cout << akbar << std::endl;}*

1. **Weak pointer:**

untracked pointer that can point to shared memory address, we can directly reference to anyshared pointer with 1st making the make\_etc() like stuff.

Example:

*{*

*shared\_ptr <User>ali=make\_shared<User>();*

*weak\_ptr <User>akbar=ali;*

*std::cout << akbar << std::endl;}*

**Move Semantic:**

Create reference memory space instead of creating reserved memory

Use: int temp= move(b)

**Standard template library:**

**Vector:**

like array list in java or simple array in js, no need to define size if array. It contains methods like push,pop etc

use:

1st we have to import by using include

vector<datatype> name; , name.push();

**Lambda:**

it is a function that calls automatically just after initializing.invoked automatically at the time of creation

It works like IIFE in js.

Syntax: []{}()

[] {body defination will be here}()

**Example:**

  []{std::cout << "Ali Akbar Khan" << std::endl;}();

Other type of lambda work like arrow function in js

**Syntax** [](){};

**Example:**

**auto** sum= [](**auto** a,**auto** b){return a+b;};

std::cout << sum(4,6) << std::endl;

    return 0;