02 BEGINNING WITH C++

Comments

```
// Single line comment
/* Multi                  }  can be used b/w code, ex:
   Line                   }  for( ;/*Infinite loop */; );
   Comment */             }
```

Hello World

```
#include <iostream>
using namespace std;
int main(){
    cout<< “Hello World”;
    return 0;
}
```

Input and Print

```
cout<< “Text = ” << aVariable << “\n”
     << “Text in next line”;

cin >> variable1 >> variable2   //if we type ‘1 2’
                                // 1 is stored in variable 1
                                // 2 is stored in variable 2
```

Program Structure

Include Files
Class declaration
Member functions definations
Main function

Class

```
class className {
private:
      int numVar
      //only accessible by class member functions
public:
        int newNumVar //accessible publicly
        outputType fnName(inputType var1,...)
};

outputType className:: fnName (inputType var1,...){
        //code
        return outputTypeData;
}

int main(){
        className var;
        outputType var1;
        var1 = var. fnName (...);
        return0;
}
```

Relationship with C

C++ is a superset of C.

So, most of everything that work in C, work in C++ too.

03

TOKENS, EXPRESSIONS, AND CONTROL STRUCTURES

Tokens

Smallest individual units of program; keywords, identifiers, constants, strings, operators

reserved identifiers
like: break, while, void etc

names of vars, fns, classes etc
starts with char or underscore
can have char, digits, underscore

Basic datatypes

① User defined: struct, union, class, enum

② Derived: array, fn, pointer, reference

③ Built in: integral (int, char), floating (float, double), void

Enum

```
enum  enumName {a,b,c}; // a=0, b=1, c=2
enum  enumName {a,b,c=5,d,e};// 0 1 5 6 7
enum  enumName {a,b=5,c,d=8,e}; // 0 5 6 8 9

int num=a; // num=0
```

Reference var(&)

dataType & refVar = orginalVar;

Both refVar and originalVar now point to same thing.

```
(ex) int a=1;
     int&b=a;
     b=2; // Now both a and b =2
(ex) type fnName(int &x){
          x=2; }
     fnName(y) // y will be 2
```

Scope resolution operator (::)

:: variableName

allows to use the global version of variableName

Memory management operators (new, delete)

```
dataType *ptrVar1= new dataType ;
dataType *ptrVar2= new dataType[x][y]; // for array

delete ptrVar1; // Equivalent to
delete[]ptrVar2 ; // free() in C
```

Manipulators (endl, setw)

endline ≡ "\n"

```
cout<< setw(5)<<10<<endL        //    10      reserve 5 spaces and
    << setw(5)<<10000;          // 10000      right justify
```

Use "left" to change setw() to left justify... ex:

```
cout<< left << setw(5)<<10 ; // 10...     3 space after 10
```

setw() needs #include <iomanip> at top

Control structures

```
(if-else if-else)              (switch-case)
if (condition) {               switch (expression){   //expression must
    //code }                                          given int value
                                 case possibleVal1:
else if (condition){                   //code
                                       break;
   //code }                      case possibleVal2:   // If this then
else {                                 //code         default code also executed
  //code }                                            as no break;
                                 default:
                                       //code }
```

```
(while)                  (do-while)                (For)
while(condition){        do{                       for(initialization;test;incr)
    //code }                  //code }             {
                         while(condition);             // code
                                                   }
//condition checked      //condition checked       // init, test, code, inc,
before code is run        after code is run         test, code, inc, test...
```

04 FUNCTIONS IN C++

Prototype

```
                                        optional
returnDataType fnName(dataType argVar1, ...);
//If we want to use fnName before defining it, we must put the
// prototype above the area where we use it (Generally at top).
```

Defination

```
returnDataType fnName(dataType argVar1, ...){
        //code
        return returnDataTypeVar;
}
```

Call

```
returnDataTypeVar = fnName(argumentsList);
```

Call by reference

```
void addOne(int &var){     //var points to a rather than
        var+=1; }          //being a copy of it
int a=1;
addOne(a) //a will be 2
```

Return by reference

```
int &max(int&a, int&b){
        return (a>=b)?a:b ; }
int a=1, b=2;
max(a,b)=0; // b is max, so fn returns &b i.e. b=0
```

Inline Functions

```
inline {FunctionHeader}
{ //body }
```

Expands the fn wherever it is called. Increases speed for short fns. Not beneficial for larger fns due to increased memory usage.

Default arguments :

```
void fnName(int a, int b, float c, int d=4, float e=4.5){//code}
```

all arguments with default values must be on the end.

```
fnName(1,2,3.2) // 1  2  3.2  4  4.5
fnName(1,2,3.2,6) // 1  2  3.2  6  4.5
```

Function overloading.

We can have multiple functions of the same name with difft number and types of arguments. Program uses whichever of them best matches the number and type of arguments in the function call.

Math library

ceil(x) → Rounds to next larger integer

floor(x) → Rounds to previous smaller integer

pow(x), sqrt(x), fabs(x), log(x), log10(x), pow(x,y)

sin(x), cos(x), tan(x)

$|x|$ base e ($\ln$)   base 10 ($\log_{10}$) x^y

//x and return value are doubles.

//include <cmath> header needed.

M T W T F

Page No

Date:

05 Classes and Objects

Structures

```
struct structName {               //defining a
     dataType var1;               //structure
     dataType var2; };

struct structName var3;    // declare a structure variable
var3. var1 = 3             // members can be accessed with do
struct structName var4[10];// Array of structures
```

$\underbrace{\texttt{struct}}_{\text{optional}}$

Class

```
class className {
private:
     //private var declarations
     // private function declarations
public:
     //public var declarations
     // public function declarations };

retun Type className :: fnName (argumentsList) {
     //code };     //class fn defirations canbe made outside
                  // like this or directly inside class (inline)
                  // But we just use inline fns and dont define
                  // fns inside class

className var;          // Declare a class variable
var. publicVars = 2;
var. publicFn(args);
className varArray[10]; // Array of classes
```

- Private stuffs are only accessible by class functions
- Class fns can call other class fns just by fnName

Classes as Fn Args and Returns

```
void fnName1 (className c1);    //pass by value
void fnName2 (className& c1);   //pass by reference

className fnName3(argList);  // return a className class
```

Friendly Functions

```
class className{
public:
      friend <function header>; };
```

A friend fn can access private stuffs of classes in which it is declared friend. Usually used with class objs as arguments.

06 CONSTRUCTORS AND DESTRUCTORS

Constructor

```
class className{
public:
     className(argList); };

className::className (argList){
     //This is called whenever an obj of the class is created }

className var1 (argList);                //Method 1
className var2 = className (argList);    // Method 2
```

- They can be overloaded.
- They can have reference to its own class in args (copy constructor)

Destructor

```
class className{
public:
     ~className(); };

className::~className(){
     //This is called whenever the obj go out of scope
     // Takes no arguments
     // Used to delete dynamically allocated memory }
```

07

OPERATOR OVERLOADING AND TYPE CONVERSIONS

Overloading operators

```
class className{
public:
    returnType operator <operatorActual> (argList);          //Way1
    Friend returnType operator <opActual> (argList); }     //Way2
```

- argList has 0args for ~~fn~~ member fn and 1 arg for friend fn for unary operators.
- arglist has 1 arg for member fn and 2 args for friend fn for binary operators.
- Operators can be overloaded multiple times for difft types.
- For binary ops and member fn, the right value is passed to the member fn of left value.

Non-overloadable operators

- sizeof, ., .*, ::, ?: operators cannot be overloaded.
- =, (), [], → are operators where friend fns cannotbe used.

Type Conversions

(Basic to Class type)

```
class className{                              // Define a constructor
public:                                       // with the basic datatype
    ~~String~~ className(basicDataType var);};  // as argument.

classNameVar = basicDataTypeVar;              // Fn above invoked for
//classNameVar and the basicDataTypeVar passed as argument.
```

(Class to Basic type)

```
class className{
public:
    operator basicDataType(); }; //return basicDataType
basicDataTypeVar = basicDataType(classObj);    //Way1
basicDataTypeVar = classObj;                   //Way 2
```

(Class to Class type)

We can use the class to basic type method and treat the 2nd class as basicDataType.

08 INHERITANCE : EXTENDING CLASSES

Inheriting basic

```
class baseClass {
public: //stuff
protected: // stuff
private: // stuff };
```

class derivedClass : <public or protected or private> baseClass{//stuff};
↑visibility mode

Visibility

(Base class)		(Derived Class)		
		(Public mode)	(Protectedmode)	(Private Mode)
Private	⟶	Not inherited	Not inherited	Not Inherited
Protected	⟶	Protected	Protected	Private
Public	⟶	Public	Protected	Private

Protected label

Members and fns of this label can be inherited by derived classes but not accessible by own class objects similar to that of the private label.

Data members/fns with same name.

IF both derived and base class has a var/fn with same name, the derived class one will be used.

Multiple inheritance

class derivedClass : visibility bC1, visibility bC2 . . . { //stuff};
//derivedClass inherits both bC1 and bC2 classes

If multiple baseClasses have same var/fn names, we can define it in the derivedClass separately and pass whichever class's fn/var to use using scope resolution.
ex: void repeatedFn(void) { bC1 :: repeatedFn(); }

Multipath Inheritance

class grandParent
class parent1 class parent2
class derivedChild

All members/Fns of grandParents are inherited twice by derived Child.

To fix this double inheritance, visibility mode of parent1 and 2 can be set as: <public/protected/private> <virtual>
ex:- class parent1 : public virtual grandParent {//stuff};

Constructor of derived classes

```
Class derivedClass: visibility bc1 , visibility bc2 {
public:
      derivedClass (argList): bc1(args1), bc2 (args2) {
            //stuff }
};
```

Nested classes

```
class className {
private:
      class1 obj1;            // Using objects of other
      class 2 obj2;           // classes as members
public:
      className(args): obj1(args1 ), obj2(args2) { //stuff }
};
```

These need to be initialized this way

09 POINTERS, VIRTUAL FUNCTIONS AND POLYMORPHISM

Pointer

```
dataType *pointerVar;
pointerVar = &dataTypeVar; //now we can use *pointerVar instead
                           //of dataTypeVar
(array)
int a[10]; // a is a ptr to the first element
int *aptr; // of the array
aptr=a; // *aptr : 1stelement, *(aptr+1) : 2nd element etc
(String)
char *text = "Roshan"; // can be used to initialize text
cout<<text; // Prints Roshan
(function)
returnType (*ptrName)(argList);
ptrName = &fnName; // ptrName(argList) can now be used
(Objs)
className *ptr, obj;
ptr=&obj;
ptr → memberFn();      //way 1
(*ptr).memberFn();     // way 2
```

*this Pointer

All class members have access to a *this ptr which points to the object itself.

10 MANAGING CONSOLE I/O OPERATIONS

Get, Put

cin.get(charVar); // input a char
cout.put(charVar); // print a char

(Note) cin.get() waits until newline char, bring all to memory, then distribute to get(). Char is not transferred as soon as we type it.

Getline, Write

cin.getline(textVar, size); //Reads until \n or size no.of chars (whichever early)
cout.write(textVar, size); // writes; if text less than size, it
//prints gibberish values

Manipulators

cout << manipulator << otherstuff;

- manipulators are like flags that affect stuff to immediate right.
- multiple manipulators can be used one after another and they all affect the stuff to right after the manipulators.

(common)

- (showpos, noshowpos): + prefix before positive numbers; active until switched using the complementary one.
- (dec, hex, oct): all numbers after these are converted and printed from decimal to that system.
- (showpoint, noshowpoint): show zeros after decimal
- (setprecision(n) +fixed): Sets precision of floating pt nums.
- (fixed, scientific): decimal notations
- (left, right): Left/right justify sign and value

  (internal): left justify sign, right justify value
- (setw(int)): reserve weldth (by default right justified)
- (setfill(char)): fill character
- (endl): \n char

Note: Most of them needs #include <iomanip>

custom manipulators

```
ostream & manipulator(ostream &output){
          output << "Custom";
          return output; }
cout<< manipulator<<"a";   // Prints "Customa"
//This can also be used to stack multiple manipulators into a custom
```

11 WORKING WITH FILES

Stream declaration

#include <fstream>
ifstream fInVar (“filename”, mode); // Input from file
ofstream fOutVar (“filename”, mode); // Output to file

modes

- ios::app → append at end
- ios::in → input from file (default for ifstream)
- ios::out → output to file (default for ofstream)
- ios::ate → open and take stream ptr to EOF
- ios::trunc → delete file contents in exists
- ios::binary → open as binary file
- ios::nocreate → fn fails if file doesnot exist
- ios::noreplace → fn fails if file exists

Basic read write

(Note) I/O fns of previous chapter work with file streams too
(Note) fIn/outVar=0 if fails to open

Array/Class obj read write

fIn.read((char*)&arrayOrObjOrObjArray, sizeof( ));
fOut.write(");

File stream ptr

- seekg(intOffset, fromPosFlag) → seek for get (reading from file) move pointer to another location
- seekp(") → used while writing to file. ^ (seek for put)
- tellg() → gives int offset from beginning. For using while reading.
- tellp() → " writing.

(fromPosFlags)
ios::beg (from beginning), ios::cur (from current)
ios::end (from end)

Error Handling Fns

fIn.good() → True if no error occurred yet

fIn.bad() → True if error occurred

fIn.fail() → True if a read/write failed

fIn.clear() → Clears all these errror states

fIn.eof() → True if EOF detected

(Note) For reading till end of file, put the read line directly in a while statement like → while((ch=fIn.get())!= EOF);

while(fIn.read((char*)&objVar, sizeof(objVar));

12 TEMPLATES

Functions

```
template <typename tVar1, typename tVar2 ...> //default allo
returnType FnName (args) {
      // We can use tVarN as datatype anywhere in this fn }

fnName < dataType1, dataType2 ... > (args); // fn call
```

Classes

```
template < typename tVar1, typeName tVar2 = int, ... >
class ClassName {
private:
      tvar1 mem1;
      tvar 2 mem2;
public:
      returnType FnName(args); };

template< typename tvar1 ... >
returnType className<tVar1, ...>:: FnName (args) { //stuff }

className < dataType1, dataType2 ... > obj(args);
```

13 EXCEPTION HANDLING

try, throw, catch

```
try{
    throw <string, int, double, class...>
catch(const char *errorString){ //Stuff}
catch (customClass &classObj){ //Stuff}
catch(...){ //If thrown type doesnot match any catch blocks
            // this is executed}
```

Note

- We can use throw anywhere, not just inside a try block directly.
- When a throw encountered, it travels from fn caller to caller until a try catch block catch it or it reaches main() where a generic message is shown and program exits.

Restrict exception types

```
returnType FnName(args) throw(listofAllowExcTypes);
                                   ↑
                               int, double
                               const char *
                                 etc

//If we throw any other exception type than allowed, program
//terminates completely with a generic message.
```

INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY

STL

Container, algorithms, iterators

Containers

(Sequence)

① Vector: Dynamic array. Direct access to all elements. Slow insertion and deletion (except at back end).

② Deque: Direct access to all elements. Slow insertion and deletion (except at both ends).

③ List: Doubly linked list. Fast insertion and deletion. Slow read as no direct access.

(Associative)

① set: Unique elements stored.

② multiset: Duplicates allowed

③ map: Elements are key value pairs with unique keys.

④ multimap: Duplicate keys allowed

(Derived)

① Stack: LIFO ② Queue: FIFO ③ Priority Queue

Algoritms (Some Important Ones) #include <algorithm>

(Nonmutating)

- count(): occurance of a value
- find() or find-end(): find position from value
- equal(): True if 2 ranges are equal
- search(): Find subsequence

(Mutating)

- copy() or copy_backward(): copy sequence to another
- Fill(): fill all with a value
- remove(): delete by value
- replace() • reverse() • swap()

(Sorting)

binary-search(), merge(), sort(), nth-element()
↖put element at a place

(set)

includes(), set-difference(), set_intersection, set_union()

(relational)

equal(), $\underbrace{\text{max(), min()}}_{\text{of 2 values}}$, $\underbrace{\text{max\_element, min-element()}}_{\text{of a sequence}}$, $\underset{\text{1st difference}}{\underset{\uparrow}{\text{mismatch()}}}$

(Numeric)

accumulate(), partial-sum(), inner_product()

Vector

#include<vector>
vector<int/dataType> variable;
vector<dataType>::iterator itrVar; //itrVar acts as a ptr to ele
variable[i] // element //need to be initialized as
(vectorVar.fn()) //itrVar = vecVar.begin()

- begin(): 1st element reference
- end() : last "
- size() : no.of elements
- push_back(int/dataType) : add new ele to back
- pop-back() : delete last ele
- erase(startRef, stopRef) : delete elements
- insert(posRef, val) : insert element at a position

List

#include<list>
list<dataType> liVar;
list<dataType>::iterator itrVar = liVar.begin()
(listVar.fn())
All vector fns are valid here too.

Map

```
#include<map>
map<keyDataType, valueDataType> mVar;
map<        "                  >::iterator itVar=mVar.begin();
(*itVar).first or second //gives key and value respectively
mVar[key]=value //add new element
value = mVar[key] // get value
```

Set, deque

Similar to the ones above

15 MANIPULATING STRINGS

Creating

```
#include <string>
string s1, s2(“text”), s3(s2);
s1 = “text”;
cin>>s1;
getline(cin,s1);
```

Operators

• =	Assignment	• ==	equality
• +	Concatenate	• !=	not equal
• +=	Concatenate +assignment	• <, >, <=, >=	comparison

Functions

• size() or length()	total chars
• capacity()	current max size before array needs to grow
• max-size()	max chars the array can grow into
• begin() or end()	reference to first and last
• at(i) or [i]	char at i posn
• substr(startPt, charsFromStartPt)	substring
• find(subString)	starting pt of subString in string
• find_first_of(char)	first char occurance
or find_last_of (char)	last char occurance
• insert(atPos, string)	insert string atPos of another
• replace(startPos, endPos, string)	replace a part of string by location
• erase(startPos, endPos)	delete a part of string
• swap(anotherString)	swap 2 strings

Note

Most STL algorithms work with strings.