

//Arrays pointers and objects

#include <iostream>

using namespace std;

class public\_data; //forward declaration

int func (int \*p,int &x);

int func\_o (public\_data\*);

class public\_data

{

public:

int a,b,c,d[20];

char c1;

public\_data(void);

~public\_data(void);

};

public\_data::public\_data(void)

{

a = 111;

b = 222;

c = 333;

d[0] = 444;

cout << "Public data created\n";

}

public\_data::~public\_data(void)

{

cout << "Public data destroyed\n";

}

class testClass

{

int var1,var2;

public:

testClass();

void set\_vars(int,int);

int ret\_vars(int);

};

testClass::testClass()

{

var1 = 1;

var2 = 0;

cout << "testClass object initialized" <<endl;

}

void testClass::set\_vars(int a,int b)

{

var1 = a;

this->var2 = b; //assignment using "this" pointer

}

int testClass::ret\_vars(int a)

{

switch(a)

{

case 1: return this->var1;

case 2: return var2;

default : return 0;

}

}

int main(void)

{

int i=1,j=2,k=3;

public\_data dat1;

testClass obj1, \*o; //"o" is a pointer to object

o = &obj1; //points at obj1

o->set\_vars(88,99); //accessing member functions using pointers

cout << o->ret\_vars(1) << endl;

func (&i,j);

cout << i << j << endl;

func\_o(&dat1);

return 0;

}

// C style, C++ style(reference operator)

int func (int \*p , int &x)

{

\*p = \*p + 10;

x = x + 10; //x is the reference (no need of \*x)

x++;

return 0;

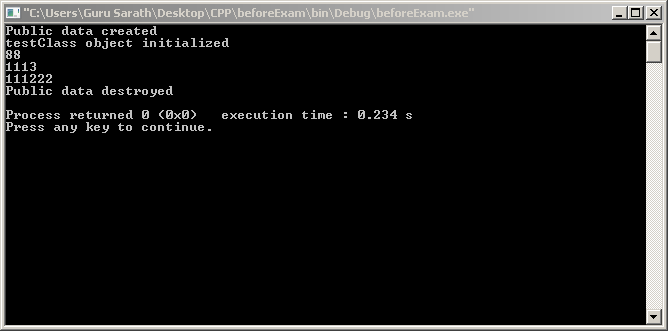
}

int func\_o (public\_data \*pd)

{

cout << pd->a << pd->b << endl;

}



//Classes and objects

#include <iostream> //new headers

using namespace std;

class int\_stack; //Forward declaration of the class (needed because the class name is specified in sample\_function)

int sample\_function(int,int,int,int\_stack);

class int\_stack

{

// Default is private

int data[20],tos;

friend int access(int ele,int\_stack a); //friend function (Not a member function)

public:

static int object\_count; //This static variable is common to all the objects

int\_stack(int); //This is the constructor function (Has the same name as the class)

int\_stack(); //constructor overload

~int\_stack(void); //This is the destructor function

void push(int);

int pop(void);

inline int add\_elements(int a,int b) {return data[a] + data[b];} //this is an inline function

};

int int\_stack::object\_count; //This golbal declaration defines the static variable object\_count

// Static variable s are automatically initialised to '0'

int access(int ele,int\_stack a) //friend function;class is passed as a parameter

{

//Object 'a' is created

//(When a object is passed as parameter to a function, it is pass by value by default)

return(a.data[ele]);

//object destructor will be called here

}

void int\_stack::push(int ele)

{

data[tos] = ele;

tos++;

}

int int\_stack::pop(void)

{

tos--;

return data[tos];

}

int\_stack::int\_stack(int i)

{

int k;

tos = 0;

object\_count ++;

for(k=0;k<20;k++) data[k] = i;

cout << "stack was initialised; Objectcount = " << object\_count <<endl;

}

int\_stack::int\_stack()

{

int k;

tos = 0;

object\_count ++;

for(k=0;k<20;k++) data[k] = 0;

cout << "stack was initialised (overload); Objectcount = " << object\_count <<endl;

}

int\_stack::~int\_stack(void)

{

cout << "Object destroyed" << endl ;

}

int main()

{

int\_stack stack1 = 0;

stack1.push(5);

stack1.push(12);

stack1.push(44);

stack1.push(78);

//sample\_function(0,0,0,stack1);

// copy of object

// is made (call by value)

cout << stack1.pop() <<access(0,stack1) << endl; //destructor funtion will be called once more after the execution of this statement

cout << stack1.pop() <<stack1.add\_elements(0,2)<<endl;

int\_stack stack2 = 5,stack3(1),stack4(0) ; //Different ways of passing value to constructor

int\_stack stack\_ov[2]; //overload required

int\_stack stack\_ary[3] = {0,1,2};

cout << stack1.object\_count<<endl;

}

int sample\_function(int a,int b,int c,int\_stack x)

{

x.push(99);

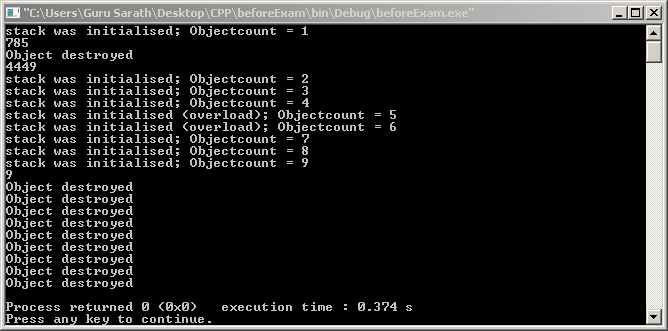
a=2;

b=3;

c=5;

return 0;

}



//Check the output of this program with and without the copy constructor to understand the difference

//Copy contructor will be called under the following cases

// -> When an object initializes another object in a declaration

// -> When copy of the object is passed to a function

// -> When a temporary object is generated (usually in return statements of functions)

#include <iostream>

using namespace std;

class box

{

public:

int \*p;

int length;

box() {cout << "box was created\n"; p = new int;}

~box() {cout << "box was destroyed\n"; delete p;}

box(const box &o){cout << "box was created using copy contructor \n"; p = new int;}

//&o is the reference to object on the right

};

class dynamic\_stack

{

int \*p,\*first\_element;

public:

int LEN;

dynamic\_stack(int s\_size) //constructor

{

try{

p = new int[s\_size];

}catch(bad\_alloc xa) {cout << "FAILED TO BUILD STACK";}

first\_element = p;

LEN = s\_size;

}

dynamic\_stack() //constructor overloaded

{

try{

p = new int[20];

}catch(bad\_alloc ax) {cout << "FAILED TO BUILD STACK";}

first\_element = p;

LEN = 20;

}

dynamic\_stack(const dynamic\_stack &o) //copy constructor

{

cout << "\nSTACK copy constructor invoked\nCOPY SUMMARY\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \n";

try{

p = new int[o.LEN];

}catch(bad\_alloc ax){cout << "COPY FAILED !! \n";}

first\_element = p; cout << first\_element << " first element \n";

LEN = o.LEN; cout << LEN << " LEN \n";

p += (o.p - o.first\_element); cout << p << " ptr \n";

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

//initialize the stack

for(int i=0;i<o.LEN;i++)

\*(first\_element+i) = \*(o.first\_element + i) ;

}

~dynamic\_stack() //destructor

{

delete []p;

cout << "stack erased\n";

}

void push(int ele)

{

try{

if(p == (first\_element+LEN)) throw LEN;

\*p = ele;

p++;

}catch (int x)

{

cout << "Stack overflow MAX LEN = " << LEN << endl;

}

}

int pop()

{

try{

if(p == first\_element) throw 0;

p--;

return \*p;

}catch(int x)

{

cout << "STACK ptr at 0 !!!\n";

return \*p;

}

}

void list\_content()

{

for(int i=0;i<20;i++)

{

cout << " " <<\*(first\_element+i);

if((first\_element+i) == p)

cout << " <- ptr";

cout << endl;

}

cout << "------------\n";

}

};

int main()

{

cout << "size of int = " << sizeof(int) << endl;

box b1;

box b2 = b1; //copy contructor will be called

\*(b1.p) = 10;

\*(b2.p) = 20;

cout << \*(b1.p) << \*(b2.p) << endl;

dynamic\_stack S1;

S1.pop();

for(int i=1;i<=5;i++)S1.push(i\*11);

S1.list\_content();

dynamic\_stack S2 = S1;

S2.list\_content();

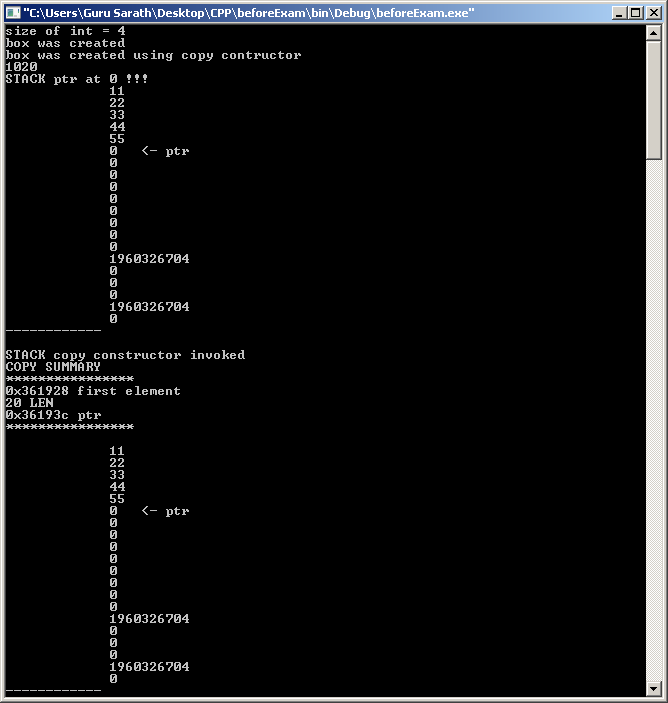
for(int i=1;i<=3;i++)cout << S2.pop() << endl;

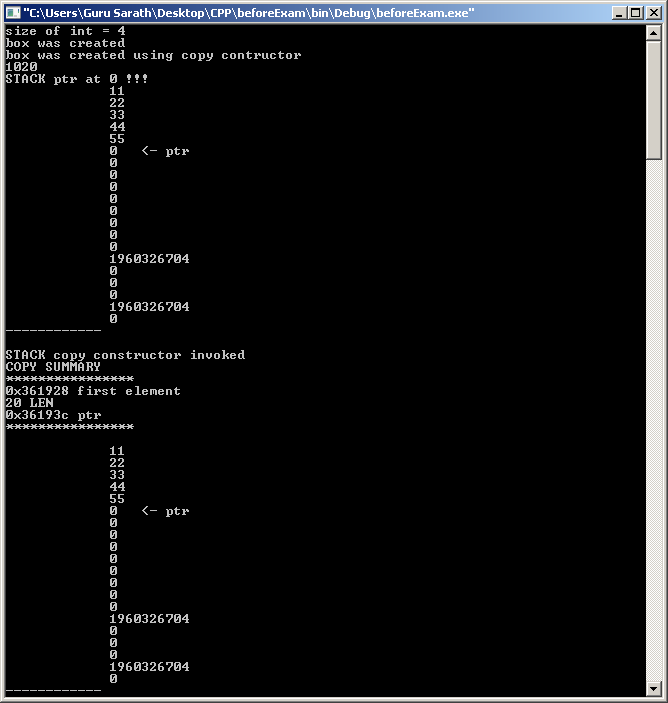
S2.list\_content();

for(int i=1;i<=16;i++)S2.push(i\*93);

S2.list\_content();

}





//Declaration - naming an object

//Definition - create the object (allocate memory)

//initialization - set initial value for the object

//CONSTANTS

//1. Literal 2.Defined 3.Memory constant

//You cannot have a variable of type void

//Integer types -- int (short,long,long long) , char , bool

//Real types -- float , double (long)

//complimenting the third bit from the left converts a alphabet from lower to uppercase

#include <iostream>

using namespace std;

#define PI 3.1415 // Defined constant

void func(float); //Declaration of the function

int main()

{

int x; // This is both Declaration and initialization

const int y = 7; //Memory constant

//y =9; //causes an error

short v; //same as short int v;

long vv; // same as long int vv;

int vvv; // same as signed int vvv; (SIGNED is default)

cout << "char " << sizeof(char) << endl;

cout << "short int " << sizeof(short int) << endl;

cout << "int " << sizeof(int) << endl;

cout << "long int " << sizeof(long int) << endl;

cout << "long long int " << sizeof(long long int) << endl << endl;

cout << "float " << sizeof(float) << endl;

cout << "double " << sizeof(double) << endl;

cout << "long double " << sizeof(long double) << endl << endl;

// Literal constants (unnamed constants used to specify data)

'G'; //char

'\0'; //Null character (zero)(represents 8zero bits)

112; //signed integer (DEFAULT)

7734U; //unsigned integer constant

-489489L; //Long signed integer

783463458LL; //Long Long

// note that there is no way to represent short int

0.0; //double (DEFAULT)

.0; //double

0.; // double

3.14; //double

3.14F; //float

33.333321L; //long double

cout << "sizeof(-3.1415) = " << sizeof(-3.1415) << endl;

cout << "sizeof(12.4F) = " << sizeof(12.4F) << endl;

cout << "sizeof(345) = " << sizeof(345) << endl;

cout << "sizeof(345LL) = " << sizeof(345LL) << endl;

cout << "sizeof(34523453245) = " << sizeof(34523453245) << endl;

try

{

throw 3.14;

}catch(float x)

{

cout << "caught a float \n\n";

}catch(double x)

{

cout << "caught a double \n\n";

}

int a=2,b=9,c = 5;

cout << -c++ << endl; //-(c++)

cout << c << endl;

//Precedence - Used to determine the order in which different operators are evaluated

//Associatively - Used to determine the order when operators of same precedence are present

x = a\*b/c%4; //Left to right associative

a += b -= c \*= 5; //Rigth to left associative (a += (b -= (c \*= 5)))

x = 7; // This expression as a whole has a value 7

func(x=98);

bool bool\_var = 75; // Non zero value is converted to 1 (true)

int int\_var = true; // true = 1 and false = 0

cout << "Int variable - " << int\_var << " bool variable - " << bool\_var << endl;

}

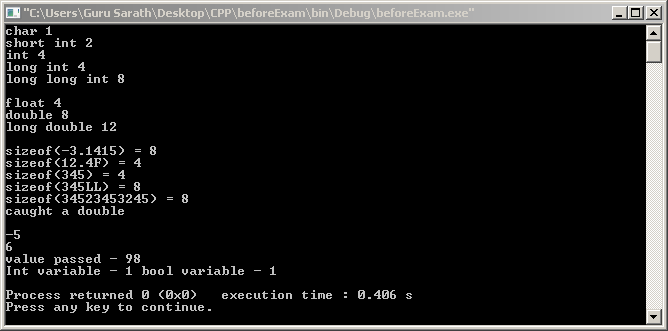
void func(float x) //Definition of the function

{

cout << "value passed - " << x << endl;

return;

}



//Dynamic allocation ,Deallocation

#include <iostream>

using namespace std;

class student

{

public:

int marks,age;

char name[10];

student(int m,int a) {marks = m; age = a; cout << "student created \n";}

~student(){cout << "Student destroyed \n";}

};

int main()

{

int a,i;

int \*p,\*k;

student \*l;

a = 3;

if(a>1)

{

int b;

b = a;

p = &b;

}

//b goes out of scope but b still exists in memory

cout << \*p << endl; //prints the value of b

\*p = 22; //changes the variable b

k = new int(4444); //dynamically create an integer and initialise it to 4444

cout << \*k << endl;

delete k;

k = new int[10]; //dynamic allocation (array cannot be initialised using the new operator)

for(i=0;i<10;i++)

\*(k+i) = i\*10; //array initialisation

for(i=0;i<10;i++)

cout << \*(k+i) << endl;

delete []k; //delete the array

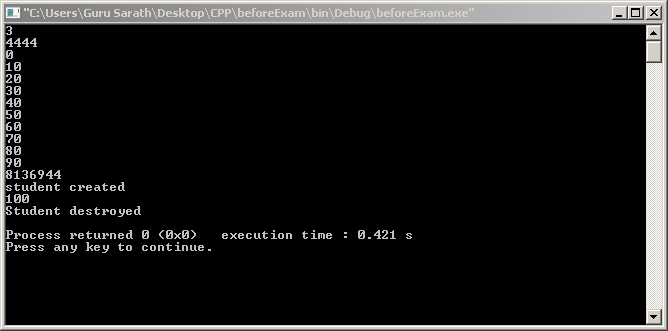
cout << \*k << endl;

l = new student(100,9); //dynamically created object with values passed to the constructor

cout << l -> marks << endl;

delete l;

}



// An exception can be thrown from outside the try block or from a function which is inside the try block

// DERIVED CLASS CATCH CLAUSE

// A catch clause for a base class will also catch a class derived from the base class

//To catch all exceptions simply use

// catch(...) { }

//You can restrict the type of exception a function

//return\_type func\_name(arg list) throw (type list)

//if the type list is empty the function cannot throw any exception

//type list is a comma seperated list

//you can rethrow an exception by

//catch

//{

// throw;

//}

//unexpected() function is called when

//a function tries to throw an exception that is not allowed by its throw list

//terminate() function is called when

//there is no matching catch statement

//a program tries to rethrow an exception when no exception was originally thrown

#include <iostream>

using namespace std;

class base

{

public:

int i;

};

class derived : public base

{

public:

int i;

};

int main()

{

try

{

cout << "Inside the first try block\n";

throw 10;

cout << "This statement will not be executed\n";

}

catch(int a)

{

cout << "Exception detected\n\n";

}

int i;

try

{

cout << "inside the second try block \n input i ? " ;

cin >> i;

if(i >= 10) throw i;

cout << "value of i is less than = 10 (NO EXCEPTION)";

}

catch (int a)

{

cout << "Exception detected with the input data \ni = " << i << endl;

}

derived D1;

try

{

throw D1;

}catch(base B)

{

cout << "\nBase class exception caught" << endl;

}catch(derived B)

{

//This will not execute

cout << "Derived class exception caught\n";

}

//SOLUTION TO ABOVE PROBLEM - Reverse the order of catch clause

try

{

throw D1;

}catch(derived B)

{

cout << "Derived class exception caught" << endl;

}catch(base B)

{

cout << "Base class exception caught\n";

}

int v;

cout << "Enter a value for v (0 or 1 or 2) ";

cin >> v;

try

{

if(!v) throw 1;

if(v == 1) throw "LALALA";

if(v == 2) throw 'a';

throw 123.22;

}catch(char x)

{

cout << "caught a char\n";

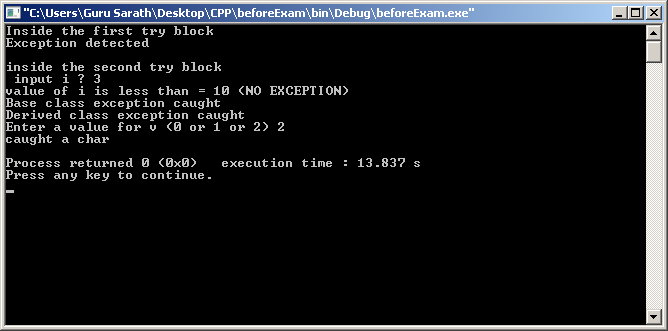
}catch(...)

{

cout << "caught something\n";

}

}



//Function overloading

#include <iostream>

using namespace std;

/\* The functions that differs only by their return type cannot be overloaded \*/

int first\_function(int i); // first\_function overloaded

int first\_function(float i);

class metal

{

public:

float density,strength;

char color[10],name[10];

//overload constructor

metal(float den,float stren){density = den; strength = stren;}

metal() {density = 9.7; strength = 62.6;}

void get\_data();

};

void metal::get\_data()

{

cout << "density - " << density <<endl;

cout << "strength - " << strength << endl <<" -------------- " << endl;

}

int main()

{

int a,b,c,d;

float k = 2.5;

a = first\_function(17); //int first\_function(int i); will be called

b = first\_function(k); //int first\_function(float i); will be called

cout << a << "\t" << b << endl << endl;

metal iron(9.6,102.3) , silver , copper; //initialised and an uninitialised object

iron.get\_data();

silver.get\_data();

copper.density = 8.9;

copper.strength = 87.2;

copper = iron;

iron.strength = 203;

copper.get\_data();

iron.get\_data();

}

int first\_function(int i)

{

i = i + 10;

return i;

}

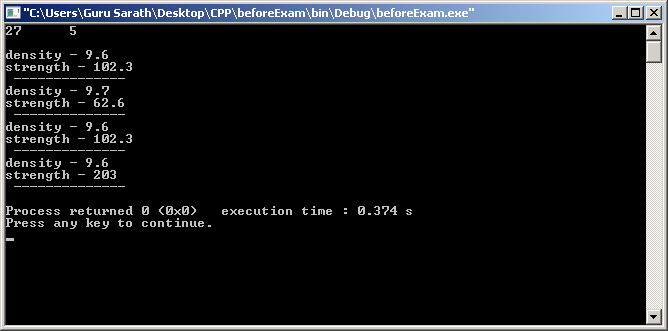
int first\_function(float i)

{

i = i + 1;

return 5;

}



//INHERITANCE

#include <iostream>

using namespace std;

class base\_class

{

private:

int i,j; //these two variables cannot be accessed by any of the derived classes

public:

base\_class() {cout << "Constructing base class \n";}

~base\_class() {cout << "Destroying base class \n";}

void set\_values(int,int);

void default\_values();

void print\_values();

protected:

int p1,p2;

};

void base\_class::set\_values(int x,int y)

{

i = x;

j = y;

}

void base\_class::print\_values()

{

cout << i << j << endl;

}

void base\_class::default\_values()

{

i = 10;

j = 20;

}

class derived\_class1 : public base\_class

{

//cannot access variables i and j (private members of the base class)

private:

int d1;

public:

derived\_class1(){cout << "Constructing derived class \n";}

~derived\_class1(){cout << "Destroying derived class \n";}

void set\_d1(int);

void set\_p(int,int);

void print\_pd\_values();

};

void derived\_class1::set\_d1(int d)

{

d1 = d;

}

void derived\_class1::set\_p(int val1,int val2)

{

//variables p1 and p2 are protected members of base class

p1 = val1;

p2 = val2;

}

void derived\_class1::print\_pd\_values()

{

cout << p1 << p2 << d1 << endl;

}

int main()

{

derived\_class1 obj1;

//accessing public members of the base class

obj1.set\_values(24,34);

obj1.print\_values();

obj1.default\_values();

obj1.print\_values();

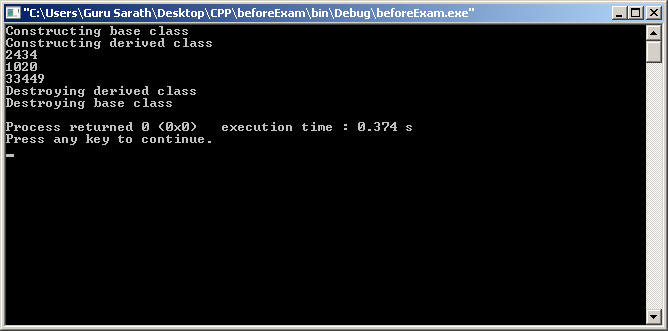
//accessing the protected members of the base class using the derived class

obj1.set\_p(33,44); //variables p1 and p2 are protected members of the base class

obj1.set\_d1(9);

obj1.print\_pd\_values();

}



/\*

OPERATOR OVERLOADING-------------------------

#you can overload an operator in two ways (using the keyword "operator")

1 - using member function (no of arguments = no of operands -1)

2 - using friend function (no of arguments = no of operands)

#you cannot overload . :: .\* ? sizeof operators

#you cannot overload = () -> using friend function operator overloading

#if the operator overload function is a member function, the operand(object) on the left side

of operator(only binary operator) calls the operator overload function.

# prefix form ret\_type operator++();

postfix form ret\_type operator++(int x); x has the value 0

#friend function s cannot use "this" operator hence ++ and -- are overloaded

\*/

#include <iostream>

using namespace std;

class int\_stack

{

int data[20],loc;

public:

int\_stack() {loc = 0; for(int i = 0; i<20 ; i++) data[i] = 0;} //constructor

void push(int);

int pop();

void display();

int operator+(int\_stack); // Overloading '+' operator (operator overload function as class member)

int\_stack operator=(int\_stack);

friend int\_stack operator-(int\_stack,int\_stack);// overloading using a friend function

friend int operator\*(int,int\_stack);

int operator++(); //prefix operator overloaded (++a)

int operator++(int x); //postfix operator overloaded (a++) //x has a value 0

};

int\_stack int\_stack::operator=(int\_stack op)

{

for(int i;i<20;i++)

data[i] = op.data[i];

loc = op.loc;

return \*this; //returns the object that generated the call (operand on the left)

// this is necessary because this allows multiple assignment (eg. a=b=c)

}

int int\_stack::operator++() //prefix

{

//add 1 to last data that was pushed to stack

data[loc-1]++;

return data[loc-1];

}

int int\_stack::operator++(int x) //postfix

{

//add 2 to last data data that was pushed to stack

data[loc-1] += 2;

return data[loc];

}

int int\_stack::operator+(int\_stack op2) //Operator overload funciton

{

int ans;

//pop the last two data and add them

ans = op2.pop() + pop();

return ans;

}

int\_stack operator-(int\_stack op\_left,int\_stack op\_right)

{

int i;

int\_stack temp;

//subtract each and every data in the stack

for(i=0;i<20;i++)

{

temp.data[i] = op\_left.data[i] - op\_right.data[i];

}

return temp;

}

void int\_stack::push(int ele)

{

if(loc == 19)

{

cout << "error : stack limit";

return;

}

data[loc] = ele;

loc++;

}

int int\_stack::pop()

{

if(loc == 0)

{

cout << "error : Empty stack" << endl;

return 0;

}

loc --;

return data[loc];

}

void int\_stack::display()

{

for(int i;i<20;i++)

cout << data[i] << endl;

cout << "------------" << endl;

}

int operator\*(int n,int\_stack obj)

{

return n\*(obj.data[(obj.loc)-1]);

}

int main()

{

int\_stack stack1,stack2,stack3;

int i;

stack1.push(12);stack1.push(36);stack1.push(34);

stack2.push(20);stack2.push(6);stack2.push(21);

stack1++; //34+1 = 35

stack2++; //21+1 = 22

++stack1; //35+2 = 37

stack1.display();

cout << stack1.pop() << stack1.pop() << endl;

cout << stack1+stack2 << endl;

cout << stack1.pop() << endl;

stack1.display();

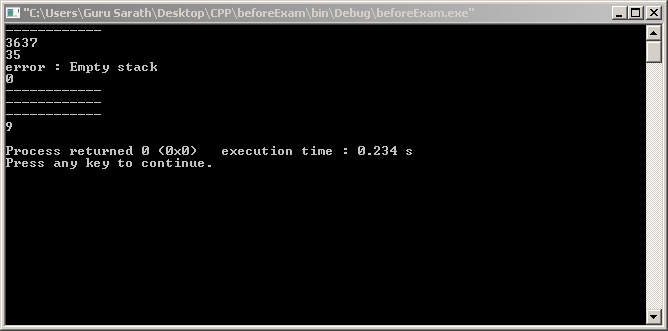
stack2.display();

stack3 = stack1 - stack2;

stack3.display();

cout << 3\*stack3 << endl;

}



// overloading [] and ()

// [] operator is treated as a binary operator

// syntax

// return\_type operator[](int x) { }

// need not be int x

// obj1(3) translates to return\_type operator[](3) { }

#include <iostream>

using namespace std;

class students

{

float marks[10];

public:

void set\_marks()

{

cout << "Enter the marks of 10 students\n";

for(int i=0;i<10;i++)

cin >> marks[i];

}

void print\_marks()

{

for(int i=0;i<10;i++)

cout << marks[i] << " ";

cout << endl;

}

students operator+(students M)

{

students temp;

for(int i=0;i<10;i++)

temp.marks[i] = marks[i] + M.marks[i];

return temp;

}

float operator[](int x)

{

return marks[x];

}

float operator()(int x,int y,char c)

{

if(c == 'A')

return marks[x] + marks[y];

else

return marks[x] - marks[y];

}

};

int main()

{

students section1,section2,section3;

section1.set\_marks();

section2.set\_marks();

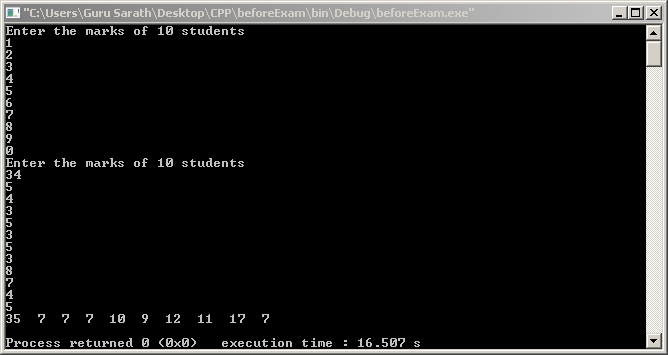
cout << section1[3] << endl;

cout << section1(1,2,'A') << endl;

section3 = section1 + section2;

section3.print\_marks();

}



//POINTERS AND REFERENCES

#include <iostream>

using namespace std;

class base\_class;

int func (int\*,int);

char \*func2 (char\*);

void swap\_values(int&,int&); //using reference operator

void sample\_func(base\_class&); //reference to an object

class base\_class

{

int x;

public:

int y;

void set\_x(int z)

{

x = z;

}

void print\_x()

{

cout << x << endl;

}

};

class derived\_class : public base\_class

{

public:

int xx;

int yy;

};

int main()

{

int ary[10],i,j = 50,k,\*p,\*m;

char \*c,\*c1,str[15] = "ThisisGuru";

for(i=0;i<10;i++) ary[i] = 0; //initialize the array elements

p = ary; //Name of the array is the pointer to the first element of the array

m = &j;

c = str; //points to the first character of the string

func(p,i);

for(i=0;i<10;i++) cout << ary[i] << endl;

cout << m << endl; // prints the address of j

cout << &j << endl; // prints the address of j

cout << \*m << endl; // prints the value of j

c1 = func2(c); //c1 will contain the address of character 'G'

\*c1 = 'X'; // Repalce 'G' with 'X'

cout << str << endl << " OBJECTS ------------------------ \n";

/\* Pointer to objects \*/

base\_class objB1[3];

derived\_class objD1;

objB1[0].set\_x(5);

objB1[1].set\_x(6);

objB1[2].set\_x(7);

base\_class \*objP; //pointer to an object of type base\_class

objP = objB1;

objP -> print\_x(); //objB1[0]

(objP + 1) -> print\_x(); //objB1[1]

(objP + 2) -> print\_x(); //objB1[2]

objP = &objD1; //base class pointer can also point to derived class

//(but can access only the members of the base class)

//A derived class pointer cannot point to an object of type base class

objP -> set\_x(9);

objP -> print\_x();

//objP -> xx = 23; //leads to an error (xx is not a member of base class)

cout << "Reference operator -----------------" << endl;

int v = 5 , w = 9;

int &vref = v; //independent reference

//vref is just an alternate name for v

v = 1;

cout << v << " " << vref << endl;

vref = 3;

cout << v << " " << vref << endl;

v = 5; w = 9;

cout << "v = " << v << " w = " << w << endl;

swap\_values(v,w);

cout << "v = " << v << " w = " << w << endl;

sample\_func(objB1[0]);

objB1[0].print\_x();

//Similar to pointers a base class reference can be used to point to a derived class

}

//Function illustrating call by reference

int func(int \*p,int a)

{

\*p = a; // first element of the array

\*(p+1) = a+1; // second element of the array

\*(p+2) = a+2; // third element of the array

return a;

}

char \*func2(char \*c)

{

while(\*c != 'G') //wait for the first occurance of 'G'

c++;

return c;

}

void swap\_values(int &x,int &y)

{

//you cannot make &x or &y to point to another location

//x++ will not point to next int location

int temp;

temp = x;

x = y;

y = temp;

}

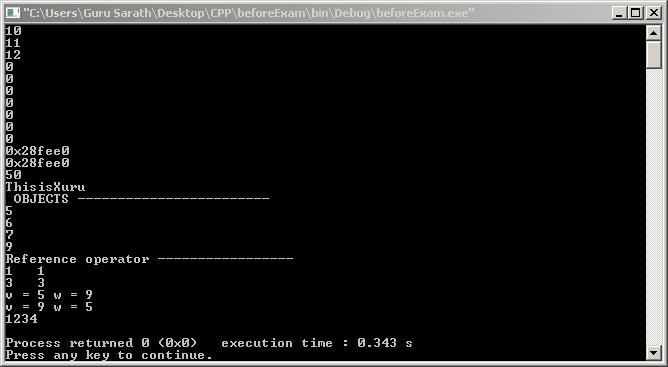
void sample\_func(base\_class &base)

{

base.set\_x(1234);

base.y = 4321;

}



//Preprocessoe directives

#include <iostream>

using namespace std;

#define PI 3.14

#define ABS(a) (a)<0?(a):-(a)

#define X 15

#if X == 15

#define Y 300

#endif

#ifndef Y

#define Z 400

#endif // Y

#define EARTH cout << "Earth\n"

int main()

{

float i = PI;

int value1 = Y;

EARTH;

#undef EARTH

//EARTH; //will cause an error

cout << i << " " << ABS(-9) << value1 ;

#if X > 20

cout << "X>20";

#elif X > 10

cout << "\nX>10\n";

#else

cout << "X<10";

#endif

#ifdef Z

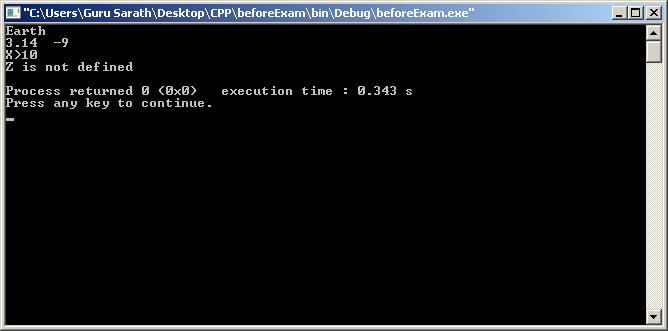
cout << "Z is defined \n"

#else

cout << "Z is not defined\n";

#endif // Z

}



//Templates

#include <iostream>

using namespace std;

template <class type> //First generic function or template function

void swapVals(type &a,type &b)

{

type temp;

temp = a;

a = b;

b = temp;

}

template <class type>

type addVals(type a,type b)

{

cout << "operation on one type of data ANS = ";

return (a+b);

}

template <class type1, class type2>

int addVals(type1 a,type2 b) //Overloaded generic function

{

cout << "operation on two different data ANS = ";

return(a+b);

}

template <class type>

void bubbleSort(type \*p,int len) //mixed parameter list

{

for(int i=0;i<len;i++)

{

for(int i=0;i<len-1;i++)

{

if(\*(p+i) > \*(p+i+1))

{

\*(p+i) = \*(p+i+1) + \*(p+i);

\*(p+i+1) = \*(p+i) - \*(p+i+1);

\*(p+i) = \*(p+i) - \*(p+i+1);

}

}

}

}

int main()

{

int a = 5, b = 8;

char x = 'a', y = 'b';

swapVals(a,b);

swapVals(x,y);

cout << a << b;

cout << x << y << endl;

cout << addVals(a,b) << endl;

cout << addVals(x,y) << endl;

cout << addVals(a,x) << endl;

cout << addVals(b,x) << endl;

int ary[10] = {2,3,4,1,2,6,7,9,4,5};

char str[12] = "TGURUSARATH";

bubbleSort(ary,10);

bubbleSort(str,12);//arrange characters in alphabetical order

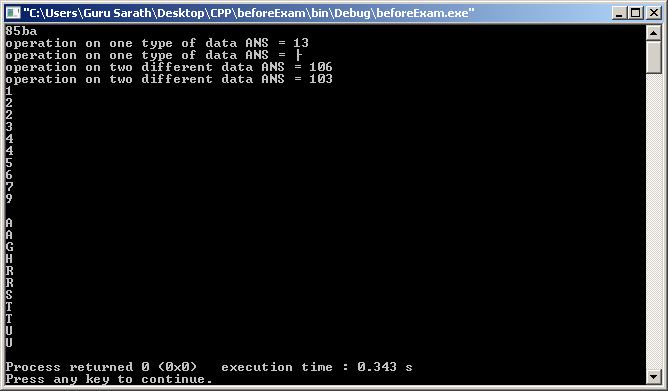
for(int i=0;i<10;i++)

cout << ary[i] << endl;

for(int i=0;i<12;i++)

cout << str[i] << endl;

}



// Virtual function is a function which is declared in base class and redefined in derived class (ploymorphism)

// Prototype of virtual function must match exactly as that of the base class prototype (hence different from function overloading)

// Virtual functions cannot be friend function

// Constructor cannot be virtual function

// Destructor can be a virtual function

// You can use a base class pointer to access any class derived from base class

// You can use a base class reference to access any class derived from the base class

// No matter how many times you inherit a base class containing a virtual function the virtual function remains virtual

#include <iostream>

using namespace std;

class base

{

public:

virtual void print\_out() {cout << "BASE \n";}

};

class derived : public base

{

public:

//print\_out function is still virtual (virtual nature of print\_out function is inherited)

void print\_out() { cout << "DERIVED \n"; } //overriding base class function

};

class derivedA : public base

{

public:

//base class funtion is not overridden

//hence when print\_out is called, function from base class is called

};

class derivedB : public derived

{

public:

//again print\_out function (virtual function) is not overridden

//but now when print\_out function is called, derived's print\_out will be called (nearest function will be called)

};

int main()

{

derived obj1 ;

base obj2 , \*ptr;

obj1.base::print\_out(); //access the base class function

obj1.print\_out(); //access the derived class function

obj2.print\_out();

ptr = &obj1; // ptr is an base class pointer but here it is used to point to dreived class object

ptr->print\_out();

derivedA DA;

derivedB DB;

DA.print\_out(); //will call base class function

DB.print\_out(); //will call derived class function (nearest function)

}

