--------------------------------------------------------------------------------------------------------------------------

STRINGS IN C++

--------------------------------------------------------------------------------------------------------------------------

/\* C++ implements a powerful string object class similar to that

\* of java. Thus there are two ways in string manipulation

\* The first method is derived from C where strings are noting but

\* 'null' terminated character arrays.

\* In the second method 'string' objects are used like other data types

\* such as int etc

\* It will be shown that string objects of C++ provide all features of

\* Strings so it is

\* preferable to use string objects only and ignore including C specific

\* code like strcat strcpy etc which are very much required if we use

\* C's implementation of strings that is null terminated char arrays

\*/

#include <iostream>

#include <string> //To utilise string objects implemented in C++

#include <cstdlib>

#include <string.h>

/\*We can include C header files so we are including string.h library

from C to get access to strcat strcpy etc

Alternatively we could haves included the 'cstring' library to get

access to functions like strcat strlen etc

\*/

using namespace std;

int main()

{

int a=1,b=2;

//Approach 1 : Using string objects

string s="Hello this is a string.";

cout<<s<<a<<endl;

//String objects can be reassigned to other literals

s="This is another string.";

cout<<s<<b<<endl;

//String obects can also be added to one another

s=s+"This is third string.";

cout<<s<<(a+b)<<endl;

//s.size() gives size of the string

for(int i=0;i<s.size();i++)

{

if(s[i]=='\0') //Like C string objects are also null terminated.

break;

cout<<s[i]; //Individual string characters can be accesed using [] notation

}

cout<<endl;

//Approach 2: Using char arrays;

char str[80]="Hello this is a string.";

cout<<str<<a<<endl;

strcpy(str,"This is another string.");

cout<<str<<b<<endl;

strcat(str,"This is third string.");

cout<<str<<(a+b)<<endl;

return 0;

}

--------------------------------------------------------------------------------------------------------------------------

STRINGSTREAM

---------------------------------------------------------------------------------------------------------------------------

/\* In C++ streams are treated as objects and we can perfrom extraction

\* and inclusion of information through >> and << operators respectively

\* There exixts a mechanism where stings in C++ can be converted to

\* stream objects using 'stringstream' method. The stringstream objetct

\* so created behaves similar to other stream objects like 'cin' or 'cout'

\* \*/

#include <iostream>

#include <string> // For String Objects

#include <sstream> // For StringStream implementation

using namespace std;

int main()

{

int i,n1;

float f;

string n;

string s="12 13 14 15 16 5.6 ";

//Creating strinstream object

stringstream stream1;

stream1.str(s);

cout<<"We are displaying utitility of 'stringstream' class"<<endl<<

" This is a standard string and we will extract tokens from it"

<<s<<endl;

/\*

stream1>>i;

cout<<i<<endl;

stream1>>i;

cout<<i<<endl;

stream1>>i;

cout<<i<<endl;

stream1>>i;

cout<<i<<endl;

stream1>>i;

cout<<i<<endl;

stream1>>i;

cout<<i<<endl;

\*/

for(i=1;i<=6;i++)

{

if(i<=5)

{

stream1>>n1;

cout<<n1<<endl;

}

else

{

stream1>>f;

cout<<f<<endl;

}

}

return 0;

}

--------------------------------------------------------------------------------------------------------------------------

DYNAMIC ALLOCATION

--------------------------------------------------------------------------------------------------------------------------

#include <iostream>

#include <new>

using namespace std;

int main()

{

int i,n;

int \* p;

cout<<"How many numbers would you like to type?";

cin>>n;

/\* While using new operator if the request for memory is unsuccesful

\* the program execution is stopped as an Exception is thrown

\* The nothrow modifier allows us to proceed with the program

\* even if memory could not be allocated

\* if 'new' fails then it returns '0' to the pointer

\*

\* \*/

p= new (nothrow) int[n];

if(p==0)

{

cout<<"Memory Could not be allocated";

return 1;

}

else

{

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

{

cout<<"Enter no: ";

cin>>p[i];

}

}

cout<<"You have entered.";

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

cout<<p[i]<<" ";

//Freeing up memory

delete[] p;

/\* To delete more than one memory block like array delete [] is used

\* to delete a single element just 'delete' followed by pointer name

\* is sufficient

\* \*/

return 0;

}

--------------------------------------------------------------------------------------------------------------------------

CLASSES

--------------------------------------------------------------------------------------------------------------------------

/\* ---------------------------------------------------------------------

\* C++ supports classes. Classes are data structures which not only

\* contain data memebers but also functions. Thus C++ implementaion of

\* a class is similar to that of struct except than we can include

\* functions. Besides this access specifier for each member of the

\* class can be specified

\* An 'access specifier' specifies or lays down rules for accesibility

\* of a variable in different parts of code. Whether we can acces a

\* variable or modify (directly and indirectly) depends on the access

\* specifer associated with the variable

\* In C++ the following access specifiers are provide

\* private : Concerned element (both data and functions) can only be

\* accessed from within the class. This is the default specifer for all

\* elements of a class unless otherwise specified.

\* protected : Concerned element can be accessed from within the class

\* and only from within derived classes.

\* public : Concerned element can be accessed fom anywhere freely.

\* ---------------------------------------------------------------------

SYNTAX

class class\_name

{

access\_specifier\_1 : member\_1;

access\_specifier\_2 : member\_2

...

...

}object\_names;

\*/

#include <iostream>

using namespace std;

class CRectangle{

int x,y; //private data members cant't be accessed out of class

public :

void setvalues(int,int);

int area() { return x\*y; }

};

void CRectangle::setvalues(int a,int b){

x=a;

y=b;

}

int main(){

CRectangle rect;

rect.setvalues(3,5);

//If we try to access protected memberes directly we have error

//rect.x=5;

//rect.y=6;

cout<<"Area is : "<<rect.area();

return 0;

}

--------------------------------------------------------------------------------------------------------------------------

/\* SYNTAX of '::'

\* --------------------------------------------------------------------

\* The :: operator is called scope operator and it is used to specify

\* the scope of an element

\* the usage is x::y where

\* y is the element whose scope is to be specified

\* and x is the scope of the elemeny like namespace or class

\* Examples are std::cout

\* where cout is the standard stream object and std is its scope

\*/

#include <iostream>

using namespace std;

class CRectangle {

int width,height;

public:

CRectangle();

CRectangle(int,int);

double area();

};

//Defininng Members of Class

CRectangle::CRectangle(){

width=0;

height=0;

}

CRectangle::CRectangle(int a,int b){

width=a;

height=b;

}

double CRectangle::area(void){

return width\*height\*1.0;

}

//Class defination ends

int main(){

CRectangle recta,rectb(3,5);

/\*To use default specifer simply write the object name without

\* parenthesis

\* so recta is ok

\* but recta() is not allowed

\* -----------------------------------------------------------------

\* Furthermore C++ provides for default constructor iff and only

\* if no constructors for the clas has been explicitly specified

\* Thus iff we specify even one constructor then we need to manually

\* specify the default constructor for the class

\* The default constructor is the constructor that has no paramaters

\* and objects can be instantiated simply by specifying their class

\* and the name of the object without providing default values to

\* the data members of the class.

\*------------------------------------------------------------------

\*/

cout<<recta.area()<<" "<<rectb.area()<<endl;

return 0;

}

--------------------------------------------------------------------------------------------------------------------------

POINTERS TO CLASSES

-----------------------------------------------------------------------------------------------------------------------------

#include <iostream>

using namespace std;

class CRectangle{

int width,height;

public:

void set\_values(int,int);

int area(void) { return width\*height; }

};

void CRectangle::set\_values(int a,int b){

width=a;

height=b;

}

int main(){

CRectangle a,\*b,\*c;

//Pointers pointing to CRectangle Objects

CRectangle \*d=new CRectangle[2]; //Array of crectangle obects

b=new CRectangle; //Default Constructor

/\*Use of new operator enusues that concerned variable is created

\* at runtime from heap memory so 'b' and 'd' are created from heap

\*/

c=&a;

a.set\_values(1,2);

b->set\_values(3,4);

d->set\_values(5,6); //Accesing array elements using pointer

d[1].set\_values(7,8); //Pointer is first dereferenced then used

// Note that areas of a and c will be equal coz c points to a

cout<<"a area: "<<a.area()<<endl;

cout<<"b area: "<<b->area()<<endl;

cout<<"c area: "<<c->area()<<endl;

cout<<"d area: "<<d[0].area()<<endl; //Also note (d+0)->area();

cout<<"d[1] area: "<<d[1].area()<<endl; // Equivalent to d[1].area()

//b and d were explicitly allocated form heap using new keyword.

//A good program must always clean up its heap memory usage though \

the process is automatic

delete[] d;

delete b;

//NO need to delete a coz it is allocated from stack

return 0;

}

--------------------------------------------------------------------------------------------------------------------------

INHERITANCE IN C++

----------------------------------------------------------------------------------------------------------------------------

/\* Inheritance is a feature in C++ where one class (called derived class)

\* can derive the

\* attributes i.e both data and functions of another class (called

\* base class)

\*

\* SYNTAX

\* ---------------------------------------------------------------------------

\* return\_type derived\_class\_name : max\_access\_level base\_class

\*

\* A class may derive data from another class but what about the

\* acces specifiers of the concerned data.

\* The max\_data\_level specifies the max access level that derived data

\* obtains in the derived class

\* Let us say class C has 1 private 1 protected and 1 public data membrs

\* Another class D inherits C so it has same 3 data members but width

\* possibly different acces scopes.

\* If the max access for D is specified as public it means that

\* all members will be inherited keeping their access specifiers intact

\* However if max access was specifies as private it means that

\* data members with acces greater than private will be inherited as

\* private only

\* Furthermore if max access level is protected it means

\* that private members will be inherited with their access levels intact

\* and data members with acces level greeater than max access will be

\* inherited but their acccess will be reduced to max access in the

\* derived class

\* public has highest access level and private has lowest

\*

\* \*/

#include <iostream>

using namespace std;

class CPolygon {

protected:

int height,width;

public:

void set\_values(int a,int b){

height=a;

width=b;

}

};

class CRectangle: public CPolygon {

public:

int area(){

return width\*height;

}

};

class CTriangle : public CPolygon {

public :

double area(){

return width\*height\*0.5;

}

};

int main(){

CRectangle rect;

CTriangle trg;

rect.set\_values(2,4);

trg.set\_values(2,4);

cout<<rect.area()<<endl;

cout<<trg.area()<<endl;

return 0;

}

------------------------------------------------------------------------------------------------------------------------------

POLYMORPHISM IN C++

--------------------------------------------------------------------------------------------------------------------------

/\* ---------------------------------------------------------------------

\* In computer science polymorphism is the ability of an entity with

\* a fixed identity to behave differently in different scenarious

\* Thus one particular behaviour can not be used to identify the

\* entity

\* For example a function may have fixed name but have different

\* implementations which not only take different parameters as inputs

\* but also the underlying functionality may be different

\* In C++ this is the case of 'overloaded' and 'virtual functions'

\* Any class which has atleast one virtual function is called a polymor-

\* phic class.

\* A virutal function is a function whose defination provided in one

\* base class can be modified or kept intact in all classes that

\* derive from this class.

\* Thus virtaul functions only serve to give an indication of what

\* is to be performed

\* A virtual function which has atleast some minimal functionality

\* defined in the base class is called an 'impure virtual' function.

\* On the other hand if the virtual function has zero implementation/

\* functionality defined in it is called a 'pure virtual' function

\* Virtual functions are decalared by placing keyword 'virtual'

\* in their declaration in the base class

\*

\* SYNTAX

\* ---------------------------------------------------------------------

\* class base\_class {

\* //Some data members

\* //Some functions

\*

\* //A virtual function (impure)

\* virtual int area1() { //Some Implementaion }

\* //A pure virtual function (pure)

\* virtual int area2()=0; //Note the '0' specified in declaration

\* };

\* ---------------------------------------------------------------------

\*

\* A class that has atleaat one 'pure virtual' function is called an

\* abstract class. Onjects of such class cannot be instantiated.

\* A class that derives an abstract calss may choose to define

\* one or more of functions of abstract class however at its own

\* indescrition

\* ---------------------------------------------------------------------

\* In C++ plus a pointer to a derived class is type compatible

\* with a pointer to base class .

\* However the reverse is not true

\* A pointer to base class is not type compatible with pointer to

\* derived class. This is because the derived class may declare one

\* or more of its own data members and so a pointer to a base class

\* cannot possibly have any knowledge of theese members.

\* ---------------------------------------------------------------------

\*/

#include <iostream>

using namespace std;

//Base Class

class CPolygon{

protected:

int height,width;

public:

void set\_values(int a,int b){

height=a;

width=b;

}

};

//Derived Classes

class CRectangle : public CPolygon{

public:

int area(){

return height\*width;

}

};

class CTriangle : public CPolygon {

public:

int area() {

return height\*width/2;

}

};

int main(){

CRectangle crect;

CTriangle ctrg;

//Pointers of base class

CPolygon \* cp1=&crect;

CPolygon \* cp2=&ctrg;

/\*Pointers of a base class when pointing to a derived class

can only access those membrs which derived class inherits from that \

particular base class

\*/

cp1->set\_values(2,4);

cp2->set\_values(2,4);

/\* We have to access area function using crect and ctrg

cp1 and cp2 cannot be used to access area() because area is not

defined in CPolygon

\*/

cout<<"Area of CRectangle is: "<<crect.area()<<endl;

cout<<"Area of CTriangle is: "<<ctrg.area()<<endl;

return 0;

}

--------------------------------------------------------------------------------------------------------------------------

NAMESPACES

--------------------------------------------------------------------------------------------------------------------------

/\*

------------------------------------------------------------------------

In most programming languages to access a variable we need to specify:

it's name and it's scope. In C++ in addition to theese two we have

another paramter called namepace. Thus two variables having same name

and scope can belong to different namespaces and thus occupy separate

memory locations and be different from one another.

All elements defined in C++ standard library are defined under 'std'

namespace therefore to access them we need to prefix their namespace

in this manner, namespace::element\_name

Examples: std::cout , std::cin

An alternative is to already direct the compiler to look for variables

in a namespace which is predefined

Example: using namepace std;

Multiple namepaces can be stacked one on top of other using multiple

using statements one after all

using namepace namespace\_1

using namespace namespace\_2

Thus namepaces can be thought of like JAVA packages. Classes,

functions and variables are first declared and then grouped into

namespaces so that theese elements dont become keywords and forbid the

use of other elements which happen to have a same name.

It is considered not to import entire namepaces u via the 'using'

command rather import only speciffic elements using syntax like

using namespace:variable;

------------------------------------------------------------------------

\*/

#include <iostream>

using namespace std;

//Assigning some Global Variable with same name but different namespaces

namespace first {

int x = 1;

int y = 2;

}

namespace second{

int x = 3;

int y = 4;

}

int main(){

// To avoid conflicts arising due to same variable name

//the using statements have been grouped in { } blocks dont remove them otherwise compiler wont understand

// which variable we are using and may lead to unpredictable behaviour.

{

using namespace first;

cout << "The Value of x in namespace 'first': " << x<<endl; //Refferrin to x of first namepace

cout << "The Value of x in namespace 'second': " << second::x<<endl;

}

{

using namespace second;

cout << "The Value of x in namespace 'second': " <<x<<endl; //Referring to x of second namespace

cout << "The Value of y in namespace 'first': " <<first::x<<endl;

}

return 0;

}

--------------------------------------------------------------------------------------------------------------------------

TEMPLATE:

----------------------------------------------------------------------------------------------------------------

/\* Often we donnot know in advance what is the data type of a variable. What we do

\* know in advance is some operations that we wish to peform on the variable .

\* And that theese operations are largely independent of the type of variable.

\* Say for example if we wish to compare two numbers a and b. The code for comparing the two is similar for both int and float.

\* The question arises how do we implement this function if the data type is not known

\* beforehand ?

\* C++ provides us with a feature called Templates where we can write code using

\* 'template ' variables. This variable can be anything legal data type in C++.

\* The exact type of the varible is determined by the compiler when the code is

\* beinf compiled and the compiler undertakes the duty to ensure that correct code

\* is generated in accordance with the data type of the variable.

\*

\* SYNTAX

\*-------------------------------------------------------------------------------------------------------------

\* template <Class T> //Any class T which will be used to refer to data type

\*

\* T greaterOfTwo(T a,T b) {

\* if a>b

\* return a;

\* else

\* return b;

\* }

\* To call the function we would do greaterOfTwo<int>(a,b) or greaterOfTwo<float>(a,b)

\* Thus we see that the data type is specified explicitly in the method calling statement

\*

\*

----------------------------------------------------------------------------------------------------------------

\* \*/

#include <iostream>

#include <string>

using namespace std;

template <class T>

T getMax(T a,T b){

T result;

result = a > b ? a : b;

return result;

}

int main() {

int i = 5,j = 6, k;

float l = 10.0, m = 5.6, n ;

string a= "Hello World", b= "World";

k = getMax<int>(i,j);

n = getMax<float>(l,m);

//For int values

cout << "Value of k is: " << k << endl;

//For floating point values

cout << "Value of n is: " << n << endl;

//Even Strings are supported;

cout << getMax<string>(a,b);

return 0;

}