// example: one class, two objects

#include <iostream>

using namespace std;

class Rectangle {

int width, height;

public:

void set\_values (int,int);

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

};

void Rectangle::set\_values (int x, int y) {

width = x;

height = y;

}

int main () {

Rectangle rect, rectb;

rect.set\_values (3,4);

rectb.set\_values (5,6);

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

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

return 0;

}

// example: class constructor

#include <iostream>

using namespace std;

class Rectangle {

int width, height;

public:

Rectangle (int,int);

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

};

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

width = a;

height = b;

}

int main () {

Rectangle rect (3,4);

Rectangle rectb (5,6);

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

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

return 0;

}

// copy constructor: deep copy

#include <iostream>

#include <string>

using namespace std;

class Example5 {

string\* ptr;

public:

Example5 (const string& str) : ptr(new string(str)) {}

~Example5 () {delete ptr;}

// copy constructor:

Example5 (const Example5& x) : ptr(new string(x.content())) {}

// access content:

const string& content() const {return \*ptr;}

};

int main () {

Example5 foo ("Example");

Example5 bar = foo;

cout << "bar's content: " << bar.content() << '\n';

return 0;

}

// overloading class constructors

#include <iostream>

using namespace std;

class Rectangle {

int width, height;

public:

Rectangle ();

Rectangle (int,int);

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

};

Rectangle::Rectangle () {

width = 5;

height = 5;

}

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

width = a;

height = b;

}

int main () {

Rectangle rect (3,4);

Rectangle rectb;

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

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

return 0;

}

// pointer to classes example

#include <iostream>

using namespace std;

class Rectangle {

int width, height;

public:

Rectangle(int x, int y) : width(x), height(y) {}

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

};

int main() {

Rectangle obj (3, 4);

Rectangle \* foo, \* bar, \* baz;

foo = &obj;

bar = new Rectangle (5, 6);

baz = new Rectangle[2] { {2,5}, {3,6} };

cout << "obj's area: " << obj.area() << '\n';

cout << "\*foo's area: " << foo->area() << '\n';

cout << "\*bar's area: " << bar->area() << '\n';

cout << "baz[0]'s area:" << baz[0].area() << '\n';

cout << "baz[1]'s area:" << baz[1].area() << '\n';

delete bar;

delete[] baz;

return 0;

}

// overloading operators example

#include <iostream>

using namespace std;

class CVector {

public:

int x,y;

CVector () {};

CVector (int a,int b) : x(a), y(b) {}

CVector operator + (const CVector&);

};

CVector CVector::operator+ (const CVector& param) {

CVector temp;

temp.x = x + param.x;

temp.y = y + param.y;

return temp;

}

int main () {

CVector foo (3,1);

CVector bar (1,2);

CVector result;

result = foo + bar;

cout << result.x << ',' << result.y << '\n';

return 0;

}

// example on this

#include <iostream>

using namespace std;

class Dummy {

public:

bool isitme (Dummy& param);

};

bool Dummy::isitme (Dummy& param)

{

if (&param == this) return true;

else return false;

}

int main () {

Dummy a;

Dummy\* b = &a;

if ( b->isitme(a) )

cout << "yes, &a is b\n";

return 0;

}

// class templates

#include <iostream>

using namespace std;

template <class T>

class mypair {

T a, b;

public:

mypair (T first, T second)

{a=first; b=second;}

T getmax ();

};

template <class T>

T mypair<T>::getmax ()

{

T retval;

retval = a>b? a : b;

return retval;

}

int main () {

mypair <int> myobject (100, 75);

cout << myobject.getmax();

return 0;

}

// template specialization

#include <iostream>

using namespace std;

// class template:

template <class T>

class mycontainer {

T element;

public:

mycontainer (T arg) {element=arg;}

T increase () {return ++element;}

};

// class template specialization:

template <>

class mycontainer <char> {

char element;

public:

mycontainer (char arg) {element=arg;}

char uppercase ()

{

if ((element>='a')&&(element<='z'))

element+='A'-'a';

return element;

}

};

int main () {

mycontainer<int> myint (7);

mycontainer<char> mychar ('j');

cout << myint.increase() << endl;

cout << mychar.uppercase() << endl;

return 0;

}

// destructors

#include <iostream>

#include <string>

using namespace std;

class Example4 {

string\* ptr;

public:

// constructors:

Example4() : ptr(new string) {}

Example4 (const string& str) : ptr(new string(str)) {}

// destructor:

~Example4 () {delete ptr;}

// access content:

const string& content() const {return \*ptr;}

};

int main () {

Example4 foo;

Example4 bar ("Example");

cout << "bar's content: " << bar.content() << '\n';

return 0;

}

// friend functions

#include <iostream>

using namespace std;

class Rectangle {

int width, height;

public:

Rectangle() {}

Rectangle (int x, int y) : width(x), height(y) {}

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

friend Rectangle duplicate (const Rectangle&);

};

Rectangle duplicate (const Rectangle& param)

{

Rectangle res;

res.width = param.width\*2;

res.height = param.height\*2;

return res;

}

int main () {

Rectangle foo;

Rectangle bar (2,3);

foo = duplicate (bar);

cout << foo.area() << '\n';

return 0;

}

// friend class

#include <iostream>

using namespace std;

class Square;

class Rectangle {

int width, height;

public:

int area ()

{return (width \* height);}

void convert (Square a);

};

class Square {

friend class Rectangle;

private:

int side;

public:

Square (int a) : side(a) {}

};

void Rectangle::convert (Square a) {

width = a.side;

height = a.side;

}

int main () {

Rectangle rect;

Square sqr (4);

rect.convert(sqr);

cout << rect.area();

return 0;

}

// derived classes

#include <iostream>

using namespace std;

class Polygon {

protected:

int width, height;

public:

void set\_values (int a, int b)

{ width=a; height=b;}

};

class Rectangle: public Polygon {

public:

int area ()

{ return width \* height; }

};

class Triangle: public Polygon {

public:

int area ()

{ return width \* height / 2; }

};

int main () {

Rectangle rect;

Triangle trgl;

rect.set\_values (4,5);

trgl.set\_values (4,5);

cout << rect.area() << '\n';

cout << trgl.area() << '\n';

return 0;

}

// constructors and derived classes

#include <iostream>

using namespace std;

class Mother {

public:

Mother ()

{ cout << "Mother: no parameters\n"; }

Mother (int a)

{ cout << "Mother: int parameter\n"; }

};

class Daughter : public Mother {

public:

Daughter (int a)

{ cout << "Daughter: int parameter\n\n"; }

};

class Son : public Mother {

public:

Son (int a) : Mother (a)

{ cout << "Son: int parameter\n\n"; }

};

int main () {

Daughter kelly(0);

Son bud(0);

return 0;

}

// multiple inheritance

#include <iostream>

using namespace std;

class Polygon {

protected:

int width, height;

public:

Polygon (int a, int b) : width(a), height(b) {}

};

class Output {

public:

static void print (int i);

};

void Output::print (int i) {

cout << i << '\n';

}

class Rectangle: public Polygon, public Output {

public:

Rectangle (int a, int b) : Polygon(a,b) {}

int area ()

{ return width\*height; }

};

class Triangle: public Polygon, public Output {

public:

Triangle (int a, int b) : Polygon(a,b) {}

int area ()

{ return width\*height/2; }

};

int main () {

Rectangle rect (4,5);

Triangle trgl (4,5);

rect.print (rect.area());

Triangle::print (trgl.area());

return 0;

}

// virtual members

#include <iostream>

using namespace std;

class Polygon {

protected:

int width, height;

public:

void set\_values (int a, int b)

{ width=a; height=b; }

virtual int area ()

{ return 0; }

};

class Rectangle: public Polygon {

public:

int area ()

{ return width \* height; }

};

class Triangle: public Polygon {

public:

int area ()

{ return (width \* height / 2); }

};

int main () {

Rectangle rect;

Triangle trgl;

Polygon poly;

Polygon \* ppoly1 = &rect;

Polygon \* ppoly2 = &trgl;

Polygon \* ppoly3 = &poly;

ppoly1->set\_values (4,5);

ppoly2->set\_values (4,5);

ppoly3->set\_values (4,5);

cout << ppoly1->area() << '\n';

cout << ppoly2->area() << '\n';

cout << ppoly3->area() << '\n';

return 0;

}

// abstract base class

#include <iostream>

using namespace std;

class Polygon {

protected:

int width, height;

public:

void set\_values (int a, int b)

{ width=a; height=b; }

virtual int area (void) =0;

};

class Rectangle: public Polygon {

public:

int area (void)

{ return (width \* height); }

};

class Triangle: public Polygon {

public:

int area (void)

{ return (width \* height / 2); }

};

int main () {

Rectangle rect;

Triangle trgl;

Polygon \* ppoly1 = &rect;

Polygon \* ppoly2 = &trgl;

ppoly1->set\_values (4,5);

ppoly2->set\_values (4,5);

cout << ppoly1->area() << '\n';

cout << ppoly2->area() << '\n';

return 0;

}

// pure virtual members can be called

// from the abstract base class

#include <iostream>

using namespace std;

class Polygon {

protected:

int width, height;

public:

void set\_values (int a, int b)

{ width=a; height=b; }

virtual int area() =0;

void printarea()

{ cout << this->area() << '\n'; }

};

class Rectangle: public Polygon {

public:

int area (void)

{ return (width \* height); }

};

class Triangle: public Polygon {

public:

int area (void)

{ return (width \* height / 2); }

};

int main () {

Rectangle rect;

Triangle trgl;

Polygon \* ppoly1 = &rect;

Polygon \* ppoly2 = &trgl;

ppoly1->set\_values (4,5);

ppoly2->set\_values (4,5);

ppoly1->printarea();

ppoly2->printarea();

return 0;

}

// dynamic allocation and polymorphism

#include <iostream>

using namespace std;

class Polygon {

protected:

int width, height;

public:

Polygon (int a, int b) : width(a), height(b) {}

virtual int area (void) =0;

void printarea()

{ cout << this->area() << '\n'; }

};

class Rectangle: public Polygon {

public:

Rectangle(int a,int b) : Polygon(a,b) {}

int area()

{ return width\*height; }

};

class Triangle: public Polygon {

public:

Triangle(int a,int b) : Polygon(a,b) {}

int area()

{ return width\*height/2; }

};

int main () {

Polygon \* ppoly1 = new Rectangle (4,5);

Polygon \* ppoly2 = new Triangle (4,5);

ppoly1->printarea();

ppoly2->printarea();

delete ppoly1;

delete ppoly2;

return 0;

}

// using standard exceptions

#include <iostream>

#include <exception>

using namespace std;

class myexception: public exception

{

virtual const char\* what() const throw()

{

return "My exception happened";

}

} myex;

int main () {

try

{

throw myex;

}

catch (exception& e)

{

cout << e.what() << '\n';

}

return 0;

}

// basic file operations

#include <iostream>

#include <fstream>

using namespace std;

int main () {

ofstream myfile;

myfile.open ("example.txt");

myfile << "Writing this to a file.\n";

myfile.close();

return 0;

}

// reading a text file

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main () {

string line;

ifstream myfile ("example.txt");

if (myfile.is\_open())

{

while ( getline (myfile,line) )

{

cout << line << '\n';

}

myfile.close();

}

else cout << "Unable to open file";

return 0;

}