**Virtual functions and Polymorphism**

// virtpers.cpp

// virtual functions with person class

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class person //person class

 {

 protected:

 char name[40];

 public:

 void getName()

 { cout << " Enter name: "; cin >> name; }

 void putName()

 { cout << "Name is: " << name << endl; }

 virtual void getData() = 0; //pure virtual func

 virtual bool isOutstanding() = 0; //pure virtual func

 };

////////////////////////////////////////////////////////////////

class student : public person //student class

 {

 private:

 float gpa; //grade point average

 public:

 void getData() //get student data from user

 {

 person::getName();

 cout << " Enter student's GPA: "; cin >> gpa;

 }

 bool isOutstanding()

 { return (gpa > 3.5) ? true : false; }

 };

////////////////////////////////////////////////////////////////

class professor : public person //professor class

 {

 private:

 int numPubs; //number of papers published

 public:

 void getData() //get professor data from user

 {

 person::getName();

 cout << " Enter number of professor's publications: ";

 cin >> numPubs;

 }

 bool isOutstanding()

 { return (numPubs > 100) ? true : false; }

 };

////////////////////////////////////////////////////////////////

int main()

 {

 person\* persPtr[100]; //array of pointers to persons

 int n = 0; //number of persons on list

 char choice;

 do {

 cout << "Enter student or professor (s/p): ";

 cin >> choice;

 if(choice=='s') //put new student

 persPtr[n] = new student; // in array

 else //put new professor

 persPtr[n] = new professor; // in array

 persPtr[n++]->getData(); //get data for person

 cout << " Enter another (y/n)? "; //do another person?

 cin >> choice;

 } while( choice=='y' ); //cycle until not 'y'

 for(int j=0; j<n; j++) //print names of all

 { //persons, and

 persPtr[j]->putName(); //say if outstanding

 if( persPtr[j]->isOutstanding() )

 cout << " This person is outstanding\n";

 }

 return 0;

 } //end main()

**Virtual base class**

// normbase.cpp

**// ambiguous reference to base class**

class Parent

 {

 protected:

 int basedata;

 };

class Child1 : public Parent

 { };

class Child2 : public Parent

 { };

class Grandchild : public Child1, public Child2

 {

 public:

 int getdata()

 { return basedata; } // ERROR: ambiguous

 };

// virtbase.cpp

**// virtual base classes**

class Parent

 {

 protected:

 int basedata;

 };

class Child1 : virtual public Parent // shares copy of Parent

 { };

class Child2 : virtual public Parent // shares copy of Parent

 { };

class Grandchild : public Child1, public Child2

 {

 public:

 int getdata()

 { return basedata; } // OK: only one copy of Parent

 };

**Virtual destructors**

//vertdest.cpp

//tests non-virtual and virtual destructors

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Base

 {

 public:

 ~Base() //non-virtual destructor

// virtual ~Base() //virtual destructor

 { cout << "Base destroyed\n"; }

 };

////////////////////////////////////////////////////////////////

class Derv : public Base

 {

 public:

 ~Derv()

 { cout << "Derv destroyed\n"; }

 };

////////////////////////////////////////////////////////////////

int main()

 {

 Base\* pBase = new Derv;

 delete pBase;

 return 0;

 }

**Checking the Type of a class with dynamic\_cast**

//dyncast1.cpp

//dynamic cast used to test type of object

//RTTI must be enabled in compiler

#include <iostream>

#include <typeinfo> //for dynamic\_cast

using namespace std;

////////////////////////////////////////////////////////////////

class Base

 {

 virtual void vertFunc() //needed for dynamic cast

 { }

 };

class Derv1 : public Base

 { };

class Derv2 : public Base

 { };

////////////////////////////////////////////////////////////////

//checks if pUnknown points to a Derv1

bool isDerv1(Base\* pUnknown) //unknown subclass of Base

 {

 Derv1\* pDerv1;

 if( pDerv1 = dynamic\_cast<Derv1\*>(pUnknown) )

 return true;

 else

 return false;

 }

//--------------------------------------------------------------

int main()

 {

 Derv1\* d1 = new Derv1;

 Derv2\* d2 = new Derv2;

 if( isDerv1(d1) )

 cout << "d1 is a member of the Derv1 class\n";

 else

 cout << "d1 is not a member of the Derv1 class\n";

 if( isDerv1(d2) )

 cout << "d2 is a member of the Derv1 class\n";

 else

 cout << "d2 is not a member of the Derv1 class\n";

 return 0;

 }

Output

d1 is a member of the Derv1 class

d2 is not a member of the Derv1 class

**Changing pointer types with dynamic\_cast**

//dyncast2.cpp

//tests dynamic casts

//RTTI must be enabled in compiler

#include <iostream>

#include <typeinfo> //for dynamic\_cast

using namespace std;

////////////////////////////////////////////////////////////////

class Base

 {

 protected:

 int ba;

 public:

 Base() : ba(0)

 { }

 Base(int b) : ba(b)

 { }

 virtual void vertFunc() //needed for dynamic\_cast

 { }

 void show()

 { cout << "Base: ba=" << ba << endl; }

 };

////////////////////////////////////////////////////////////////

class Derv : public Base

 {

 private:

 int da;

 public:

 Derv(int b, int d) : da(d)

 { ba = b; }

 void show()

 { cout << "Derv: ba=" << ba << ", da=" << da << endl; }

 };

 class Derv1

 {

 };

////////////////////////////////////////////////////////////////

int main()

 {

 Base\* pBase = new Base(10); //pointer to Base

 Derv\* pDerv = new Derv(21, 22);

 Derv1\* pd = new Derv1(); //pointer to Derv

 //derived-to-base: upcast -- points to Base subobject of Derv

 pBase = dynamic\_cast<Base\*>(pd);

 // pBase = pDerv;

 pBase->show(); //"Base: ba=21"

 pBase = new Derv(31, 32); //normal

 //base-to-derived: downcast -- (pBase must point to a Derv)

 pDerv = dynamic\_cast<Derv\*>(pBase);

 pDerv->show(); //"Derv: ba=31, da=32"

 return 0;

 }

Output

Base: ba=21

Derv: ba=31, da=32

**The typeid operator**

// typeid.cpp

// demonstrates typeid() function

// RTTI must be enabled in compiler

#include <iostream>

#include <typeinfo> //for typeid()

using namespace std;

////////////////////////////////////////////////////////////////

class Base

 {

 virtual void virtFunc() //needed for typeid

 { }

 };

class Derv1 : public Base

 { };

class Derv2 : public Base

 { };

////////////////////////////////////////////////////////////////

void displayName(Base\* pB)

 {

 cout << "pointer to an object of: "; //display name of class

 cout << typeid(\*pB).name() << endl; //pointed to by pB

 }

//--------------------------------------------------------------

int main()

 {

 Base\* pBase = new Derv1;

 displayName(pBase); //"pointer to an object of class Derv1"

 pBase = new Derv2;

 displayName(pBase); //"pointer to an object of class Derv2"

 return 0;

 }

Output

pointer to an object of: 5Derv1

pointer to an object of: 5Derv2