**Pointers and Memory Allocation**

#include <iostream>

using namespace std;

**int** main()

{

 **int** x, \*p, \*\*q;

 x = 10;

 p = &x;

 q = &p;

 cout << \*\*q; // prints the value of x

 **return** 0;

}

10

**Dynamic memory allocation for 2D arrays**

In the following examples, we have considered ‘**r**‘ as number of rows, ‘**c**‘ as number of columns and we created a 2D array with r = 3, c = 4 and following values

 1 2 3 4

 5 6 7 8

 9 10 11 12

 **Using a single pointer:**
A simple way is to allocate memory block of size r\*c and access elements using simple pointer arithmetic.

|  |
| --- |
| int main(){    int r = 3, c = 4;    int \*arr = new int[r \* c];     int i, j, count = 0;    for (i = 0; i <  r; i++)      for (j = 0; j < c; j++)         \*(arr + i\*c + j) = ++count;     for (i = 0; i <  r; i++)      for (j = 0; j < c; j++)         printf("%d ", \*(arr + i\*c + j));    /\* Code for further processing and free the       dynamically allocated memory \*/      return 0;} |

 **Using an array of pointers**
We can create an array of pointers of size r. Note that from C99, C language allows variable sized arrays. After creating an array of pointers, we can dynamically allocate memory for every row.

|  |
| --- |
| int main(){    int r = 3, c = 4, i, j, count;     int \*arr[r];    for (i=0; i<r; i++)         arr[i] = new int[c];     // Note that arr[i][j] is same as \*(\*(arr+i)+j)    count = 0;    for (i = 0; i <  r; i++)      for (j = 0; j < c; j++)         arr[i][j] = ++count; // Or \*(\*(arr+i)+j) = ++count     for (i = 0; i <  r; i++)      for (j = 0; j < c; j++)         cout<< arr[i][j];     /\* Code for further processing and free the       dynamically allocated memory \*/    return 0;} |

**Using pointer to pointer**

/\* 2-D Dynamically allocated array of chars \*/

#include

using namespace std;

int main() {

int cols = 4;

int rows = 3;

// Allocate a 2-d array of ints 3 x 2

char\*\* charArray = new char\*[rows];

for(int i = 0; i < rows; ++i) {

charArray[i] = new char[cols];

}

// Fill the array

for(int i = 0; i < rows; ++i) {

for(int j = 0; j < cols; ++j) {

charArray[i][j] = char(i + 65);

}

}

// Output the array

for(int i = 0; i < rows; ++i) {

for(int j = 0; j < cols; ++j) {

cout << charArray[i][j];

}

cout << endl;

}

// Deallocate memory by deleting

for(int i = 0; i < rows; ++i) {

delete [] charArray[i];

}

delete [] charArray;

Output

|  |  |
| --- | --- |
| 123 | AAAABBBBCCCC |

 To understand this better, consider what is happening with the memory addresses:



**Sort objects by using array of pointers**

 #include <iostream>

 #include <string>

 using namespace std;

 **class** person{

 **protected**:

 string name;

 **public**:

 **void** setName()

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

 **void** printName()

 { cout << endl << name; }

 string getName()

 { **return** name; }

 };

 **int** main(){

 **void** bsort(person\*\*, **int**);

 person\* persPtr[100];

 **int** n = 0;

 **char** choice;

 **do** {

 persPtr[n] = **new** person;

 persPtr[n]->setName();

 n++;

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

 cin >> choice;

 }**while**( choice=='y' );

 cout << "\nUnsorted list:";

 **for**(**int** j=0; j<n; j++)

 {

 persPtr[j]->printName();

 }

 bsort(persPtr, n);

 cout << "\nSorted list:";

 **for**(**int** j=0; j<n; j++)

 {

 persPtr[j]->printName();

 }

 cout << endl;

 **return** 0;

 }

 **void** bsort(person\*\* pp, **int** n){

 **void** order(person\*\*, person\*\*);

 **int** j, k;

 **for**(j=0; j<n-1; j++)

 **for**(k=j+1; k<n; k++)

 order(pp+j, pp+k);

 }

 **void** order(person\*\* pp1, person\*\* pp2){

 **if**( (\*pp1)->getName() > (\*pp2)->getName() ) {

 person\* tempptr = \*pp1;

 \*pp1 = \*pp2;

 \*pp2 = tempptr;

 }

 }

**Dynamic linked list**

// linklist.cpp

// linked list

#include <iostream>

using namespace std;

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

struct link //one element of list

 {

 int data; //data item

 link\* next; //pointer to next link

 };

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

class linklist //a list of links

 {

 private:

 link\* first; //pointer to first link

 public:

 linklist() //no-argument constructor

 { first = NULL; } //no first link

 void additem(int d); //add data item (one link)

 void display(); //display all links

 };

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

void linklist::additem(int d) //add data item

 {

 link\* newlink = new link; //make a new link

 newlink->data = d; //give it data

 newlink->next = first; //it points to next link

 first = newlink; //now first points to this

 }

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

void linklist::display() //display all links

 {

 link\* current = first; //set ptr to first link

 while( current != NULL ) //quit on last link

 {

 cout << current->data << endl; //print data

 current = current->next; //move to next link

 }

 }

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

int main()

 {

 linklist li; //make linked list

 li.additem(25); //add four items to list

 li.additem(36);

 li.additem(49);

 li.additem(64);

 li.display(); //display entire list

 return 0;

 }

**Pointer to function**

 A function pointer, or a pointer to a function, can be best thought as the address of the code executed when the function is called.

int ((\*fp) (int i, int j)

declares fp to be variable of type “pointer to a function that take two integers arguments and returns an integer as its value. “

Example

double (\*fp)(double);
int main()
{
 table(sin, 0,180,10);
}
void table( double(\*fp) (double), int init, int end, int incr)
{ int theta;

for(theta = int; theta<=end; theta += incr)
 cout<<theta<<” “<<(\*fp)(theta/180.0\*3.1416));

}//end table

**Inheritance**

**Single Inheritance**

// inheritance using English Distances

#include <iostream>

using namespace std;

enum posneg { pos, neg }; //for sign in DistSign

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

class Distance //English Distance class

 {

 protected: //NOTE: can't be private

 int feet;

 float inches;

 public: //no-arg constructor

 Distance() : feet(0), inches(0.0)

 { } //2-arg constructor)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void getdist() //get length from user

 {

 cout << "\nEnter feet: "; cin >> feet;

 cout << "Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 };

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

class DistSign : public Distance //adds sign to Distance

 {

 private:

 posneg sign; //sign is pos or neg

 public:

 //no-arg constructor

 DistSign() : Distance() //call base constructor

 { sign = pos; } //set the sign to +

 //2- or 3-arg constructor

 DistSign(int ft, float in, posneg sg=pos) :

 Distance(ft, in) //call base constructor

 { sign = sg; } //set the sign

 void getdist() //get length from user

 {

 Distance::getdist(); //call base getdist()

 char ch; //get sign from user

 cout << "Enter sign (+ or -): "; cin >> ch;

 sign = (ch=='+') ? pos : neg;

 }

 void showdist() const //display distance

 {

 cout << ( (sign==pos) ? "(+)" : "(-)" ); //show sign

 Distance::showdist(); //ft and in

 }

 };

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

int main()

 {

 DistSign alpha; //no-arg constructor

 alpha.getdist(); //get alpha from user

 DistSign beta(11, 6.25); //2-arg constructor

 DistSign gamma(100, 5.5, neg); //3-arg constructor

 //display all distances

 cout << "\nalpha = "; alpha.showdist();

 cout << "\nbeta = "; beta.showdist();

 cout << "\ngamma = "; gamma.showdist();

 cout << endl;

 return 0;

 }

 **Overriding functions in the subclasses**

// models employee database using inheritance

#include <iostream>

using namespace std;

const int LEN = 80; //maximum length of names

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

class employee //employee class

 {

 private:

 char name[LEN]; //employee name

 unsigned long number; //employee number

 public:

 void getdata()

 {

 cout << "\n Enter last name: "; cin >> name;

 cout << " Enter number: "; cin >> number;

 }

 void putdata() const

 {

 cout << "\n Name: " << name;

 cout << "\n Number: " << number;

 }

 };

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

class manager : public employee //management class

 {

 private:

 char title[LEN]; //"vice-president" etc.

 double dues; //golf club dues

 public:

 void getdata()

 {

 employee::getdata();

 cout << " Enter title: "; cin >> title;

 cout << " Enter golf club dues: "; cin >> dues;

 }

 void putdata() const

 {

 employee::putdata();

 cout << "\n Title: " << title;

 cout << "\n Golf club dues: " << dues;

 }

 };

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

class scientist : public employee //scientist class

 {

 private:

 int pubs; //number of publications

 public:

 void getdata()

 {

 employee::getdata();

 cout << " Enter number of pubs: "; cin >> pubs;

 }

 void putdata() const

 {

 employee::putdata();

 cout << "\n Number of publications: " << pubs;

 }

 };

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

class laborer : public employee //laborer class

 {

 };

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

int main()

 {

 manager m1, m2;

 scientist s1;

 laborer l1;

 cout << endl; //get data for several employees

 cout << "\nEnter data for manager 1";

 m1.getdata();

 cout << "\nEnter data for manager 2";

 m2.getdata();

 cout << "\nEnter data for scientist 1";

 s1.getdata();

 cout << "\nEnter data for laborer 1";

 l1.getdata();

 //display data for several employees

 cout << "\nData on manager 1";

 m1.putdata();

 cout << "\nData on manager 2";

 m2.putdata();

 cout << "\nData on scientist 1";

 s1.putdata();

 cout << "\nData on laborer 1";

 l1.putdata();

 cout << endl;

 return 0;

 }

**Public and private inheritance**
// tests publicly- and privately-derived classes

#include <iostream>

using namespace std;

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

class A //base class

 {

 private:

 int privdataA; //(functions have the same access

 protected: //rules as the data shown here)

 int protdataA;

 public:

 int pubdataA;

 };

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

class B : public A //publicly-derived class

 {

 public:

 void funct()

 {

 int a;

 a = privdataA; //error: not accessible

 a = protdataA; //OK

 a = pubdataA; //OK

 }

 };

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

class C : private A //privately-derived class

 {

 public:

 void funct()

 {

 int a;

 a = privdataA; //error: not accessible

 a = protdataA; //OK

 a = pubdataA; //OK

 }

 };

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

int main()

 {

 int a;

 B objB;

 a = objB.privdataA; //error: not accessible

 a = objB.protdataA; //error: not accessible

 a = objB.pubdataA; //OK (A public to B)

 C objC;

 a = objC.privdataA; //error: not accessible

 a = objC.protdataA; //error: not accessible

 a = objC.pubdataA; //error: not accessible (A private to C)

 return 0;

 }

**Levels of inheritance**

// multiple levels of inheritance

#include <iostream>

using namespace std;

const int LEN = 80; //maximum length of names

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

class employee

 {

 private:

 char name[LEN]; //employee name

 unsigned long number; //employee number

 public:

 void getdata()

 {

 cout << "\n Enter last name: "; cin >> name;

 cout << " Enter number: "; cin >> number;

 }

 void putdata() const

 {

 cout << "\n Name: " << name;

 cout << "\n Number: " << number;

 }

 };

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

class manager : public employee //manager class

 {

 private:

 char title[LEN]; //"vice-president" etc.

 double dues; //golf club dues

 public:

 void getdata()

 {

 employee::getdata();

 cout << " Enter title: "; cin >> title;

 cout << " Enter golf club dues: "; cin >> dues;

 }

 void putdata() const

 {

 employee::putdata();

 cout << "\n Title: " << title;

 cout << "\n Golf club dues: " << dues;

 }

 };

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

class scientist : public employee //scientist class

 {

 private:

 int pubs; //number of publications

 public:

 void getdata()

 {

 employee::getdata();

 cout << " Enter number of pubs: "; cin >> pubs;

 }

 void putdata() const

 {

 employee::putdata();

 cout << "\n Number of publications: " << pubs;

 }

 };

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

class laborer : public employee //laborer class

 {

 };

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

class foreman : public laborer //foreman class

 {

 private:

 float quotas; //percent of quotas met successfully

 public:

 void getdata()

 {

 laborer::getdata();

 cout << " Enter quotas: "; cin >> quotas;

 }

 void putdata() const

 {

 laborer::putdata();

 cout << "\n Quotas: " << quotas;

 }

 };

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

int main()

 {

 laborer l1;

 foreman f1;

 cout << endl;

 cout << "\nEnter data for laborer 1";

 l1.getdata();

 cout << "\nEnter data for foreman 1";

 f1.getdata();

 cout << endl;

 cout << "\nData on laborer 1";

 l1.putdata();

 cout << "\nData on foreman 1";

 f1.putdata();

 cout << endl;

 }

**Member functions in multiple inheritance**

// englmult.cpp

// multiple inheritance with English Distances

#include <iostream>

#include <string>

using namespace std;

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

class Type //type of lumber

 {

 private:

 string dimensions;

 string grade;

 public: //no-arg constructor

 Type() : dimensions("N/A"), grade("N/A")

 { }

 //2-arg constructor

 Type(string di, string gr) : dimensions(di), grade(gr)

 { }

 void gettype() //get type from user

 {

 cout << " Enter nominal dimensions (2x4 etc.): ";

 cin >> dimensions;

 cout << " Enter grade (rough, const, etc.): ";

 cin >> grade;

 }

 void showtype() const //display type

 {

 cout << "\n Dimensions: " << dimensions;

 cout << "\n Grade: " << grade;

 }

 };

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

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //no-arg constructor

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void getdist() //get length from user

 {

 cout << " Enter feet: "; cin >> feet;

 cout << " Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 };

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

class Lumber : public Type, public Distance

 {

 private:

 int quantity; //number of pieces

 double price; //price of each piece

 public: //constructor (no args)

 Lumber() : Type(), Distance(), quantity(0), price(0.0)

 { }

 //constructor (6 args)

 Lumber( string di, string gr, //args for Type

 int ft, float in, //args for Distance

 int qu, float prc ) : //args for our data

 Type(di, gr), //call Type ctor

 Distance(ft, in), //call Distance ctor

 quantity(qu), price(prc) //initialize our data

 { }

 void getlumber()

 {

 Type::gettype();

 Distance::getdist();

 cout << " Enter quantity: "; cin >> quantity;

 cout << " Enter price per piece: "; cin >> price;

 }

 void showlumber() const

 {

 Type::showtype();

 cout << "\n Length: ";

 Distance::showdist();

 cout << "\n Price for " << quantity

 << " pieces: $" << price \* quantity;

 }

 };

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

int main()

 {

 Lumber siding; //constructor (no args)

 cout << "\nSiding data:\n";

 siding.getlumber(); //get siding from user

 //constructor (6 args)

 Lumber studs( "2x4", "const", 8, 0.0, 200, 4.45F );

 //display lumber data

 cout << "\nSiding"; siding.showlumber();

 cout << "\nStuds"; studs.showlumber();

 cout << endl;

 return 0;

 }

**Constructors in multiple inheritance**

// multiple inheritance with English Distances

#include <iostream>

#include <string>

using namespace std;

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

class Type //type of lumber

 {

 private:

 string dimensions;

 string grade;

 public: //no-arg constructor

 Type() : dimensions("N/A"), grade("N/A")

 { }

 //2-arg constructor

 Type(string di, string gr) : dimensions(di), grade(gr)

 { }

 void gettype() //get type from user

 {

 cout << " Enter nominal dimensions (2x4 etc.): ";

 cin >> dimensions;

 cout << " Enter grade (rough, const, etc.): ";

 cin >> grade;

 }

 void showtype() const //display type

 {

 cout << "\n Dimensions: " << dimensions;

 cout << "\n Grade: " << grade;

 }

 };

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

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //no-arg constructor

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void getdist() //get length from user

 {

 cout << " Enter feet: "; cin >> feet;

 cout << " Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 };

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

class Lumber : public Type, public Distance

 {

 private:

 int quantity; //number of pieces

 double price; //price of each piece

 public: //constructor (no args)

 Lumber() : Type(), Distance(), quantity(0), price(0.0)

 { }

 //constructor (6 args)

 Lumber( string di, string gr, //args for Type

 int ft, float in, //args for Distance

 int qu, float prc ) : //args for our data

 Type(di, gr), //call Type ctor

 Distance(ft, in), //call Distance ctor

 quantity(qu), price(prc) //initialize our data

 { }

 void getlumber()

 {

 Type::gettype();

 Distance::getdist();

 cout << " Enter quantity: "; cin >> quantity;

 cout << " Enter price per piece: "; cin >> price;

 }

 void showlumber() const

 {

 Type::showtype();

 cout << "\n Length: ";

 Distance::showdist();

 cout << "\n Price for " << quantity

 << " pieces: $" << price \* quantity;

 }

 };

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

int main()

 {

 Lumber siding; //constructor (no args)

 cout << "\nSiding data:\n";

 siding.getlumber(); //get siding from user

 //constructor (6 args)

 Lumber studs( "2x4", "const", 8, 0.0, 200, 4.45F );

 //display lumber data

 cout << "\nSiding"; siding.showlumber();

 cout << "\nStuds"; studs.showlumber();

 cout << endl;

 return 0;

 }

 **Ambiguity in multiple inheritance**

// ambigu.cpp

// demonstrates ambiguity in multiple inheritance

#include <iostream>

using namespace std;

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

class A

 {

 public:

 void show() { cout << "Class A\n"; }

 };

class B

 {

 public:

 void show() { cout << "Class B\n"; }

 };

class C : public A, public B

 {

 };

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

int main()

 {

 C objC; //object of class C

// objC.show(); //ambiguous--will not compile

 objC.A::show(); //OK

 objC.B::show(); //OK

 return 0;

 }

**Aggregation: Classes within classes**

// containership with employees and degrees

#include <iostream>

#include <string>

using namespace std;

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

class student //educational background

 {

 private:

 string school; //name of school or university

 string degree; //highest degree earned

 public:

 void getedu()

 {

 cout << " Enter name of school or university: ";

 cin >> school;

 cout << " Enter highest degree earned \n";

 cout << " (Highschool, Bachelor's, Master's, PhD): ";

 cin >> degree;

 }

 void putedu() const

 {

 cout << "\n School or university: " << school;

 cout << "\n Highest degree earned: " << degree;

 }

 };

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

class employee

 {

 private:

 string name; //employee name

 unsigned long number; //employee number

 public:

 void getdata()

 {

 cout << "\n Enter last name: "; cin >> name;

 cout << " Enter number: "; cin >> number;

 }

 void putdata() const

 {

 cout << "\n Name: " << name;

 cout << "\n Number: " << number;

 }

 };

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

class manager //management

 {

 private:

 string title; //"vice-president" etc.

 double dues; //golf club dues

 employee emp; //object of class employee

 student stu; //object of class student

 public:

 void getdata()

 {

 emp.getdata();

 cout << " Enter title: "; cin >> title;

 cout << " Enter golf club dues: "; cin >> dues;

 stu.getedu();

 }

 void putdata() const

 {

 emp.putdata();

 cout << "\n Title: " << title;

 cout << "\n Golf club dues: " << dues;

 stu.putedu();

 }

 };

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

class scientist //scientist

 {

 private:

 int pubs; //number of publications

 employee emp; //object of class employee

 student stu; //object of class student

 public:

 void getdata()

 {

 emp.getdata();

 cout << " Enter number of pubs: "; cin >> pubs;

 stu.getedu();

 }

 void putdata() const

 {

 emp.putdata();

 cout << "\n Number of publications: " << pubs;

 stu.putedu();

 }

 };

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

class laborer //laborer

 {

 private:

 employee emp; //object of class employee

 public:

 void getdata()

 { emp.getdata(); }

 void putdata() const

 { emp.putdata(); }

 };

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

int main()

 {

 manager m1;

 scientist s1, s2;

 laborer l1;

 cout << endl;

 cout << "\nEnter data for manager 1"; //get data for

 m1.getdata(); //several employees

 cout << "\nEnter data for scientist 1";

 s1.getdata();

 cout << "\nEnter data for scientist 2";

 s2.getdata();

 cout << "\nEnter data for laborer 1";

 l1.getdata();

 cout << "\nData on manager 1"; //display data for

 m1.putdata(); //several employees

 cout << "\nData on scientist 1";

 s1.putdata();

 cout << "\nData on scientist 2";

 s2.putdata();

 cout << "\nData on laborer 1";

 l1.putdata();

 cout << endl;

 return 0;

 }

**Virtual functions**

**Normal functions access with pointers (early binding)**

// notvirt.cpp

// normal functions accessed from pointer

#include <iostream>

using namespace std;

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

class Base //base class

 {

 public:

 void show() //normal function

 { cout << "Base\n"; }

 };

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

class Derv1 : public Base //derived class 1

 {

 public:

 void show()

 { cout << "Derv1\n"; }

 };

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

class Derv2 : public Base //derived class 2

 {

 public:

 void show()

 { cout << "Derv2\n"; }

 };

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

int main()

 {

 Derv1 dv1; //object of derived class 1

 Derv2 dv2; //object of derived class 2

 Base\* ptr; //pointer to base class

 ptr = &dv1; //put address of dv1 in pointer

 ptr->show(); //execute show()

 ptr = &dv2; //put address of dv2 in pointer

 ptr->show(); //execute show()

 return 0;

 }

Output
Base
Base

**Virtual functions accessed with pointers (late binding)**

// virt.cpp

// virtual functions accessed from pointer

#include <iostream>

using namespace std;

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

class Base //base class

 {

 public:

 virtual void show() //virtual function

 { cout << "Base\n"; }

 };

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

class Derv1 : public Base //derived class 1

 {

 public:

 void show()

 { cout << "Derv1\n"; }

 };

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

class Derv2 : public Base //derived class 2

 {

 public:

 void show()

 { cout << "Derv2\n"; }

 };

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

int main()

 {

 Derv1 dv1; //object of derived class 1

 Derv2 dv2; //object of derived class 2

 Base\* ptr; //pointer to base class

 ptr = &dv1; //put address of dv1 in pointer

 ptr->show(); //execute show()

 ptr = &dv2; //put address of dv2 in pointer

 ptr->show(); //execute show()

 return 0;

 }

Output

Derv1
Derv2

**Abstract class and pure virtual functions**

// virtpure.cpp

// pure virtual function

#include <iostream>

using namespace std;

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

class Base //base class

 {

 public:

 virtual void show() = 0; //**pure virtual function**

 };

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

class Derv1 : public Base //derived class 1

 {

 public:

 void show()

 { cout << "Derv1\n"; }

 };

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

class Derv2 : public Base //derived class 2

 {

 public:

 void show()

 { cout << "Derv2\n"; }

 };

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

int main()

 {

// Base bad; //can't make object from abstract class

 Base\* arr[2]; //array of pointers to base class

 Derv1 dv1; //object of derived class 1

 Derv2 dv2; //object of derived class 2

 arr[0] = &dv1; //put address of dv1 in array

 arr[1] = &dv2; //put address of dv2 in array

 arr[0]->show(); //execute show() in both objects

 arr[1]->show();

 return 0;

 }

**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;

 }

**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;

 }

**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;

 }

**Changing pointer type 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; }

 };

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

int main()

 {

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

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

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

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

 }

**friend Functions and friend Classes**

**friend Function**

// friend.cpp

// friend functions

#include <iostream>

using namespace std;

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

class beta; //needed for frifunc declaration

class alpha

 {

 private:

 int data;

 public:

 alpha() : data(3) { } //no-arg constructor

 friend int frifunc(alpha, beta); //friend function

 };

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

class beta

 {

 private:

 int data;

 public:

 beta() : data(7) { } //no-arg constructor

 friend int frifunc(alpha, beta); //friend function

 };

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

int frifunc(alpha a, beta b) //function definition

 {

 return( a.data + b.data );

 }

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

int main()

 {

 alpha aa;

 beta bb;

 cout << frifunc(aa, bb) << endl; //call the function

 return 0;

 }

// nofri.cpp

// limitation to overloaded + operator

#include <iostream>

using namespace std;

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

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public:

 Distance() : feet(0), inches(0.0) //constructor (no args)

 { } //constructor (one arg)

 Distance(float fltfeet) //convert float to Distance

 { //feet is integer part

 feet = static\_cast<int>(fltfeet);

 inches = 12\*(fltfeet-feet); //inches is what's left

 }

 Distance(int ft, float in) //constructor (two args)

 { feet = ft; inches = in; }

 void showdist() //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 Distance operator + (Distance);

 };

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

 //add this distance to d2

Distance Distance::operator + (Distance d2) //return the sum

 {

 int f = feet + d2.feet; //add the feet

 float i = inches + d2.inches; //add the inches

 if(i >= 12.0) //if total exceeds 12.0,

 { i -= 12.0; f++; } //less 12 inches, plus 1 foot

 return Distance(f,i); //return new Distance with sum

 }

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

int main()

 {

 Distance d1 = 2.5; //constructor converts

 Distance d2 = 1.25; //float feet to Distance

 Distance d3;

 cout << "\nd1 = "; d1.showdist();

 cout << "\nd2 = "; d2.showdist();

 d3 = d1 + 10.0; //distance + float: OK

 cout << "\nd3 = "; d3.showdist();

// d3 = 10.0 + d1; //float + Distance: ERROR

// cout << "\nd3 = "; d3.showdist();

 cout << endl;

 return 0;

 }

// frengl.cpp

// friend overloaded + operator

#include <iostream>

using namespace std;

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

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public:

 Distance() //constructor (no args)

 { feet = 0; inches = 0.0; }

 Distance( float fltfeet ) //constructor (one arg)

 { //convert float to Distance

 feet = int(fltfeet); //feet is integer part

 inches = 12\*(fltfeet-feet); //inches is what's left

 }

 Distance(int ft, float in) //constructor (two args)

 { feet = ft; inches = in; }

 void showdist() //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 friend Distance operator + (Distance, Distance); //friend

 };

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

Distance operator + (Distance d1, Distance d2) //add D1 to d2

 {

 int f = d1.feet + d2.feet; //add the feet

 float i = d1.inches + d2.inches; //add the inches

 if(i >= 12.0) //if inches exceeds 12.0,

 { i -= 12.0; f++; } //less 12 inches, plus 1 foot

 return Distance(f,i); //return new Distance with sum

 }

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

int main()

 {

 Distance d1 = 2.5; //constructor converts

 Distance d2 = 1.25; //float-feet to Distance

 Distance d3;

 cout << "\nd1 = "; d1.showdist();

 cout << "\nd2 = "; d2.showdist();

 d3 = d1 + 10.0; //distance + float: OK

 cout << "\nd3 = "; d3.showdist();

 d3 = 10.0 + d1; //float + Distance: OK

 cout << "\nd3 = "; d3.showdist();

 cout << endl;

 return 0;

 }

**friends for functional notation**

// misq.cpp

// member square() function for Distance

#include <iostream>

using namespace std;

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

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void showdist() //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 float square(); //member function

 };

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

float Distance::square() //return square of

 { //this Distance

 float fltfeet = feet + inches/12; //convert to float

 float feetsqrd = fltfeet \* fltfeet; //find the square

 return feetsqrd; //return square feet

 }

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

int main()

 {

 Distance dist(3, 6.0); //two-arg constructor (3'-6")

 float sqft;

 sqft = dist.square(); //return square of dist

 //display distance and square

 cout << "\nDistance = "; dist.showdist();

 cout << "\nSquare = " << sqft << " square feet\n";

 return 0;

 }

**friend class**// friclass.cpp

// friend classes

#include <iostream>

using namespace std;

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

class alpha

 {

 private:

 int data1;

 public:

 alpha() : data1(99) { } //constructor

 friend class beta; //beta is a friend class

 };

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

class beta

 { //all member functions can

 public: //access private alpha data

 void func1(alpha a) { cout << "\ndata1=" << a.data1; }

 void func2(alpha a) { cout << "\ndata1=" << a.data1; }

 };

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

int main()

 {

 alpha a;

 beta b;

 b.func1(a);

 b.func2(a);

 cout << endl;

 return 0;

 }

**Overloading operators**

**Overloading Binary Operators
Arithmetic operators**// englplus.cpp

// overloaded '+' operator adds two Distances

#include <iostream>

using namespace std;

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

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void getdist() //get length from user

 {

 cout << "\nEnter feet: "; cin >> feet;

 cout << "Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 Distance operator + ( Distance ) const; //add 2 distances

 };

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

 //add this distance to d2

Distance Distance::operator + (Distance d2) const //return sum

 {

 int f = feet + d2.feet; //add the feet

 float i = inches + d2.inches; //add the inches

 if(i >= 12.0) //if total exceeds 12.0,

 { //then decrease inches

 i -= 12.0; //by 12.0 and

 f++; //increase feet by 1

 } //return a temporary Distance

 return Distance(f,i); //initialized to sum

 }

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

int main()

 {

 Distance dist1, dist3, dist4; //define distances

 dist1.getdist(); //get dist1 from user

 Distance dist2(11, 6.25); //define, initialize dist2

 dist3 = dist1 + dist2; //single '+' operator

 dist4 = dist1 + dist2 + dist3; //multiple '+' operators

 //display all lengths

 cout << "dist1 = "; dist1.showdist(); cout << endl;

 cout << "dist2 = "; dist2.showdist(); cout << endl;

 cout << "dist3 = "; dist3.showdist(); cout << endl;

 cout << "dist4 = "; dist4.showdist(); cout << endl;

 return 0;

 }

**Unary operators**

// countpp1.cpp

// increment counter variable with ++ operator

#include <iostream>

using namespace std;

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

class Counter

 {

 private:

 unsigned int count; //count

 public:

 Counter() : count(0) //constructor

 { }

 unsigned int get\_count() //return count

 { return count; }

 void operator ++ () //increment (prefix)

 {

 ++count;

 }

 };

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

int main()

 {

 Counter c1, c2; //define and initialize

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 ++c1; //increment c1

 ++c2; //increment c2

 ++c2; //increment c2

 cout << "\nc1=" << c1.get\_count(); //display again

 cout << "\nc2=" << c2.get\_count() << endl;

 return 0;

 }

// countpp2.cpp

// increment counter variable with ++ operator, return value

#include <iostream>

using namespace std;

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

class Counter

 {

 private:

 unsigned int count; //count

 public:

 Counter() : count(0) //constructor

 { }

 unsigned int get\_count() //return count

 { return count; }

 Counter operator ++ () //increment count

 {

 ++count; //increment count

 Counter temp; //make a temporary Counter

 temp.count = count; //give it same value as this obj

 return temp; //return the copy

 }

 };

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

int main()

 {

 Counter c1, c2; //c1=0, c2=0

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 ++c1; //c1=1

 c2 = ++c1; //c1=2, c2=2

 cout << "\nc1=" << c1.get\_count(); //display again

 cout << "\nc2=" << c2.get\_count() << endl;

 return 0;

 }

// countpp3.cpp

// increment counter variable with ++ operator

// uses unnamed temporary object

#include <iostream>

using namespace std;

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

class Counter

 {

 private:

 unsigned int count; //count

 public:

 Counter() : count(0) //constructor no args

 { }

 Counter(int c) : count(c) //constructor, one arg

 { }

 int get\_count() //return count

 { return count; }

 Counter operator ++ () //increment count

 {

 ++count; //increment count, then return

 return Counter(count); // an unnamed temporary object

 } // initialized to this count

 };

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

int main()

 {

 Counter c1, c2; //c1=0, c2=0

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 ++c1; //c1=1

 c2 = ++c1; //c1=2, c2=2

 cout << "\nc1=" << c1.get\_count(); //display again

 cout << "\nc2=" << c2.get\_count() << endl;

 return 0;

 }

**Posfix Notation**

// postfix.cpp

// overloaded ++ operator in both prefix and postfix

#include <iostream>

using namespace std;

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

class Counter

 {

 private:

 unsigned int count; //count

 public:

 Counter() : count(0) //constructor no args

 { }

 Counter(int c) : count(c) //constructor, one arg

 { }

 unsigned int get\_count() const //return count

 { return count; }

 Counter operator ++ () //increment count (prefix)

 { //increment count, then return

 return Counter(++count); //an unnamed temporary object

 } //initialized to this count

 Counter operator ++ (int) //increment count (postfix)

 { //return an unnamed temporary

 return Counter(count++); //object initialized to this

 } //count, then increment count

 };

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

int main()

 {

 Counter c1, c2; //c1=0, c2=0

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 ++c1; //c1=1

 c2 = ++c1; //c1=2, c2=2 (prefix)

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 c2 = c1++; //c1=3, c2=2 (postfix)

 cout << "\nc1=" << c1.get\_count(); //display again

 cout << "\nc2=" << c2.get\_count() << endl;

 return 0;

 }

**Comparison Operators**

// engless.cpp

// overloaded '<' operator compares two Distances

#include <iostream>

using namespace std;

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

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void getdist() //get length from user

 {

 cout << "\nEnter feet: "; cin >> feet;

 cout << "Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 bool operator < (Distance) const; //compare distances

 };

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

 //compare this distance with d2

bool Distance::operator < (Distance d2) const //return the sum

 {

 float bf1 = feet + inches/12;

 float bf2 = d2.feet + d2.inches/12;

 return (bf1 < bf2) ? true : false;

 }

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

int main()

 {

 Distance dist1; //define Distance dist1

 dist1.getdist(); //get dist1 from user

 Distance dist2(6, 2.5); //define and initialize dist2

 //display distances

 cout << "\ndist1 = "; dist1.showdist();

 cout << "\ndist2 = "; dist2.showdist();

 if( dist1 < dist2 ) //overloaded '<' operator

 cout << "\ndist1 is less than dist2";

 else

 cout << "\ndist1 is greater than (or equal to) dist2";

 cout << endl;

 return 0;

 }

**Arithmetic Assignment operators**

// englpleq.cpp

// overloaded '+=' assignment operator

#include <iostream>

using namespace std;

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

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void getdist() //get length from user

 {

 cout << "\nEnter feet: "; cin >> feet;

 cout << "Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 void operator += ( Distance );

 };

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

 //add distance to this one

void Distance::operator += (Distance d2)

 {

 feet += d2.feet; //add the feet

 inches += d2.inches; //add the inches

 if(inches >= 12.0) //if total exceeds 12.0,

 { //then decrease inches

 inches -= 12.0; //by 12.0 and

 feet++; //increase feet

 } //by 1

 }

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

int main()

 {

 Distance dist1; //define dist1

 dist1.getdist(); //get dist1 from user

 cout << "\ndist1 = "; dist1.showdist();

 Distance dist2(11, 6.25); //define, initialize dist2

 cout << "\ndist2 = "; dist2.showdist();

 dist1 += dist2; //dist1 = dist1 + dist2

 cout << "\nAfter addition,";

 cout << "\ndist1 = "; dist1.showdist();

 cout << endl;

 return 0;

 }

**Data Conversion**

Type conversion

|  |  |  |
| --- | --- | --- |
| **Conversion** | **Routine in Destination** | **Routine in source** |
| Basic to basic(float to int) | Built in | Built in |
| Basic to class(int to obj) | Constructor |   |
| Class to Basic(obj to int) |   | Operator function |
| Class to class(obj to otherObj | Constructor | Operator function |

**Conversion between Class and Basic Types**

// englconv.cpp

// conversions: Distance to meters, meters to Distance

#include <iostream>

using namespace std;

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

class Distance //English Distance class

 {

 private:

 const float MTF; //meters to feet

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0), MTF(3.280833F)

 { } //constructor (one arg)

 Distance(float meters) : MTF(3.280833F)

 { //convert meters to Distance

 float fltfeet = MTF \* meters; //convert to float feet

 feet = int(fltfeet); //feet is integer part

 inches = 12\*(fltfeet-feet); //inches is what's left

 } //constructor (two args)

 Distance(int ft, float in) : feet(ft),

 inches(in), MTF(3.280833F)

 { }

 void getdist() //get length from user

 {

 cout << "\nEnter feet: "; cin >> feet;

 cout << "Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 operator float() const //conversion operator

 { //converts Distance to meters

 float fracfeet = inches/12; //convert the inches

 fracfeet += static\_cast<float>(feet); //add the feet

 return fracfeet/MTF; //convert to meters

 }

 };

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

int main()

 {

 float mtrs;

 Distance dist1 = 2.35F; //uses 1-arg constructor to

 //convert meters to Distance

 cout << "\ndist1 = "; dist1.showdist();

 mtrs = static\_cast<float>(dist1); //uses conversion operator

 //for Distance to meters

 cout << "\ndist1 = " << mtrs << " meters\n";

 Distance dist2(5, 10.25); //uses 2-arg constructor

 mtrs = dist2; //also uses conversion op

 cout << "\ndist2 = " << mtrs << " meters\n";

// dist2 = mtrs; //error, = won't convert

 return 0;

 }

**Conversion between Objects of Different Classes**

**class Cartesian**

{

double x;

double y;

public:

Cartesian()

{x=0,y=0)

Cartesian(doubly x, double y)

{

this.x=x

this.y=y

}

//added constructor

Cartesian(Polar p)

{

double r=P.getRadius();

double a=p.getAngle();

x=r\*cos(a)

y=r\*cos(a)

}

};

**class Polar**

{

double radius;

double angle;

public:

Polar()

{

radius=0;

angle=0;

}

Polar (double r, double a)

{

radius=r;

angle=a;

}

operator Cartesian()

{

double x=Radius\*cos(angle);

double y=radius\*sin(angle);

return cartesian(x,y)

}

};

**In main**

Polar P(10,.5)

Cartesian c;

c=p