```
// dynamic8.h
//SPECIFICATION FILE
class  Date
{
public :
        void print() const;
        Date ( int  initMonth, int  initDay,  int  initYear, char* initMessage );
//  constructor
        ~Date ( ) ;  //  default destructor
        void copyFrom (Date otherDate);  // deep copy operation
        Date (const Date& otherDate); // copy constructor

private :
        int   month;
        int   day;
        int    year ;
        char* message;
} ;
//  Implementation File

#include "dynamic8.h"
#include <string.h>
#include <iostream>
using namespace std;

        // print function
void Date::print () const
        {
                cout << endl
                       << "Month is " << month
                    << " Day is " << day
                    << " Year is " << year
                    << " " << message << endl;

        }

                    // a constructor
        Date::Date ( int  initMonth, int  initDay,  int  initYear, char* initMessage)
        {
                month = initMonth;
                day = initDay;
                year = initYear;

                message = new char [strlen(initMessage)+1];
                strcpy(message,initMessage);
        }
                // destructor
        Date :: ~ Date()
        {
        cout << "destructor called" << endl;
                delete [ ] message ;
        }

        Date :: Date(const Date& otherDate)
```

```
       {
             month = otherDate.month ;
             day = otherDate.day ;
             year = otherDate.year ;

             message = new char [strlen(otherDate.message)+1];
             strcpy(message,otherDate.message);
                cout << "copy constructor is called" << endl;
       }

void Date::copyFrom (Date otherDate)
       {
             month = otherDate.month ;
             day = otherDate.day ;
             year = otherDate.year ;

             delete [] message;

             message = new char [strlen(otherDate.message)+1];
             strcpy(message,otherDate.message);
       }
```

/// dynamic8.cpp

```
//client

#include "dynamic8.h"
#include <string>
#include <iostream>
using namespace std;

void dateFunction1(Date date1);
void dateFunction2(Date& date2);

       int main()
       {
       Date JulyDate(07,04,2017,"Happy fourth of July"); // constructor is called
       JulyDate.print();
       cout << "july object" << endl;
       Date OctoberDate(10,31,2017,"Happy Halloween"); // constructor is called
       OctoberDate.print();
       cout << "october1 object" << endl;
       Date AnotherDate = OctoberDate; // copy constructor is called
       AnotherDate.print();
       cout << "october2 object" << endl;
       AnotherDate.copyFrom(JulyDate); // copyFrom  is called
       cout << "october3 object" << endl;
       AnotherDate.print();
       cout << "july object" << endl;
       dateFunction1(JulyDate);
       dateFunction2(JulyDate);
       }

       void dateFunction1(Date date1)
       {
             date1.print();
```

```
	}

	void dateFunction2(Date& date2)
	{
		date2.print();
	}
```

## output

```
Month is 7 Day is 4 Year is 2017 Happy fourth of July
july object

Month is 10 Day is 31 Year is 2017 Happy Halloween
october1 object
copy constructor is called

Month is 10 Day is 31 Year is 2017 Happy Halloween
october2 object
copy constructor is called
destructor called
october3 object

Month is 7 Day is 4 Year is 2017 Happy fourth of July
july object
copy constructor is called

Month is 7 Day is 4 Year is 2017 Happy fourth of July
destructor called

Month is 7 Day is 4 Year is 2017 Happy fourth of July
destructor called
destructor called
destructor called
```

```
 //dynamic9.h
class DynArray  {
public:
	DynArray( /* in */  int arrSize );
	 // Constructor.
		// PRE:  arrSize is assigned
		// POST:  IF arrSize >= 1 && enough memory THEN
		//       Array of size arrSize is created with     //       all
elements == 0  ELSE error message.

	DynArray( const DynArray& otherArr );
		// Copy constructor.
		// POST: this DynArray is a deep copy of otherArr
		// Is implicitly called for initialization.

	~DynArray( );
```

```
             // Destructor.
             // POST: Memory for dynamic array deallocated.

       int  ValueAt ( /* in */ int i )  const;
             // PRE:  i is assigned.
             // POST: IF 0 <= i < size of this array THEN
             //        FCTVAL == value of array element at index i
             //        ELSE error message.

       void  Store ( /* in */ int val,  /* in */ int i ) ;
             // PRE:  val and i are assigned
             // POST: IF 0 <= i < size of this array THEN
             //        val is stored in array element i
             //        ELSE error message.

       void  CopyFrom ( /* in */ DynArray otherArr);
             // POST:  IF enough memory THEN
       //            new array created (as deep copy)
       //            with size and contents
       //            same as otherArr
             //        ELSE error message.

private:
       int*  arr ;
       int   size ;
```

// dynamic9i.cpp

```
// Implementation FILE
#include "dynamic9.h"
#include <cstdlib>
#include <iostream>

DynArray::DynArray( /* in */  int arrSize )
              // Constructor.
             // PRE:  arrSize is assigned
             // POST:  IF arrSize >= 1 && enough memory THEN
             //        Array of size arrSize is created with      //         all
elements == 0  ELSE error message.
{
       int i;
       if ( arrSize < 1 ) {
             std::cerr << "DynArray constructor - invalid size: "
                      << arrSize << std::endl;
             exit(1);
       }

       arr = new  int[arrSize] ;   // allocate memory

       size = arrSize;

       for (i = 0; i < size;  i++)
             arr[i] = 0;
}

void  DynArray::Store ( /* in */ int val,  /* in */ int i )
```

```
                // PRE:  val and i are assigned
                // POST: IF 0 <= i < size of this array THEN
                //        arr[i] == val
                //        ELSE error message.
 {

        if ( i < 0 || i >= size ) {
                std::cerr << "Store - invalid index : " << i << std::endl;
                //exit(1) ;
        }

        arr[i] = val ;

}

int  DynArray::ValueAt ( /* in */ int i )  const
                // PRE:  i is assigned.
                // POST: IF 0 <= i < size THEN
                //        FCTVAL == arr[i]
                //        ELSE halt with error message.
{
        if ( i < 0 || i >= size ) {
                std::cerr << "ValueAt - invalid index : " << i << std::endl;
                //exit(1) ;
        }

        return  arr[i];
}

DynArray::~DynArray( )
                // Destructor.
                // POST: Memory for dynamic array deallocated.
{
        delete [ ] arr ;
}

DynArray::DynArray( const DynArray& otherArr )
                // Copy constructor
                // Implicitly called for deep copy in initializations.
                // POST:  If room on free store THEN
                //    new array of size otherArr.size is created
                //    on free store && arr == its base address
                //     && size == otherArr.size
                 //   && arr[0..size-1] == otherArr.arr[0..size-1]
                //    ELSE error message.
{
        int i ;
        size = otherArr.size ;
        arr = new  int[size] ;      // allocate memory for copy

        for ( i = 0; i< size ; i++ )
                arr[i] = otherArr.arr[i] ;        // copies array

}
```

```
void  DynArray::CopyFrom ( /* in */ DynArray  otherArr )
             // Creates a deep copy of otherArr.
             // POST:  Array pointed to by arr@entry deallocated
             //   &&  IF room on free store
             //      THEN new array is created on free store
             //          && arr == its base address
             //          && size == otherArr.size
             //          && arr[0..size-1] == otherArr[0..size-1]
             //        ELSE halts with error message.
{
      int i ;
      delete [ ]  arr ;                  // delete current array
      size = otherArr.size ;
      arr = new int [size] ;             // allocate new array
             for ( i = 0; i< size ; i++ )    // deep copy array
             arr[i] = otherArr.arr[i] ;
             std::cout << "copyfrom function is called" << std::endl;
}
```

```
//    dynamic9.cpp
// client file

#include "dynamic9.h"
#include <iostream>
using namespace std;
```

Write a client to product the following output

output

```
0 0
1 10
2 20
3 30
4 40
5 50
6 60
7 70
8 80
9 90
copyfrom is called
(Note: print the new array)
0 0
1 10
2 20
3 30
4 40
5 50
```

```
6 60
7 70
8 80
9 90

Store - invalid index : 11
ValueAt - invalid index : 11
```

```
****************************************************************************
//*********************************************************
//  SPECIFICATION FILE              ( inherit1.h )

class  Time
{

public :

        void Set ( int  hours , int  minutes , int  seconds ) ;
        void Increment ( ) ;
        void Write ( )  const ;  // virtual has been removed
        Time ( int  initHrs, int  initMins,  int  initSecs ) ;   //  constructor
        Time ( ) ;                                     //  default constructor

 private :

        int             hrs ;
        int             mins ;
        int              secs ;
} ;

//*********************************************************************
// virtual1.cpp
// client of the ExtTime class
#include  "inherit1e.h"
#include <iostream>
using namespace std;

void Print (/*in */ Time someTime); // can handle the base class and all
descendents
```

```
 int main()
 {
         Time startTime (8,45,0);
         ExtTime    thisTime ( 8, 35, 0, PST ) ;

         Print(startTime);
         Print(thisTime);
 }

 void Print (/*in */ Time someTime)
        {
                cout << "Time is ";
                //invokes the Write() function for the parent and all
descendants
                someTime.Write();
                cout << endl;
        }
```

output

```
Time is 08:45:00
Time is 08:35:00
```

Note: the timezone is not printed

```
// virtual2.cpp
// client of the ExtTime class
#include  "inherit1e.h"
#include <iostream>
using namespace std;


void Print (/*in */ Time& someTime); // can handle the base class and all
descendents

 int main()
 {
         Time startTime (8,45,0);
         ExtTime    thisTime ( 8, 35, 0, EST) ;

         Print(startTime);
         Print(thisTime);
 }

 void Print (/*in */ Time&  someTime)   // pass by reference
        {
                cout << "Time is ";
                //invokes the Write() function for the parent and all
descendants
                someTime.Write();
                cout << endl;
        }
```

output

```
Time is 08:45:00
Time is 08:35:00 EST
```

```
// virtual 3

// without virtual functions
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat };  .

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};
```

```
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Calls the base class
} ///:~c
```

Output

```
Instrument::play
```

//virtual4

```
// Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};
```

```
void tune(Instrument& i) {
  // ...
  i.play(middleC); // Calls the derived class
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

Output

```
Wind::play
```

```
//cast1
//This program illustrates how explicit type conversion works.

#include <iostream>

using namespace std;

int main()
{
   cout << "static_cast<int>(7.9) = " << static_cast<int>(7.9)
       << endl;
   cout << "static_cast<int>(3.3) = " << static_cast<int>(3.3)
       << endl;
   cout << "static_cast<double>(25) = " << static_cast<double>(25)
       << endl;
   cout << "static_cast<double>(5 + 3) = "
       << static_cast<double>(5 + 3)
       << endl;
   cout << "static_cast<double>(15) / 2 = "
       << static_cast<double>(15) / 2
       << endl;
   cout << "static_cast<double>(15 / 2) = "
       << static_cast<double>(15 / 2)
       << endl;
   cout << "static_cast<int>(7.8 + static_cast<double>(15) / 2) = "
       << static_cast<int>(7.8 + static_cast<double>(15) / 2)
       << endl;
   cout << "static_cast<int>(7.8 + static_cast<double>(15 / 2)) = "
```

```
        << static_cast<int>(7.8 + static_cast<double>(15 / 2))
        << endl;
return 0;
}
```

**Output**

```
static_cast<int>(7.9) = 7
static_cast<int>(3.3) = 3
static_cast<double>(25) = 25
static_cast<double>(5 + 3) = 8
static_cast<double>(15) / 2 = 7.5
static_cast<double>(15 / 2) = 7
static_cast<int>(7.8 + static_cast<double>(15) / 2) = 15
static_cast<int>(7.8 + static_cast<double>(15 / 2)) = 14
```

```
// cast2.cpp
#include <iostream>
using namespace std;
class Parent {
public:
  void sleep() {cout << "go to sleep" << endl;}
};

class Child: public Parent {
public:
  void gotoSchool(){cout << "go to school" << endl;}
};

int main( )
{
  Parent parent;
  Child child;

  // upcast - implicit type cast allowed
  Parent *pParent = &child;

  // downcast - explicit type case required
  Child *pChild =  (Child *) &parent;

  pParent -> sleep();
  pChild -> gotoSchool();

  return 0;
}
```

**Output**

```
go to sleep
go to school
```

// cast 3

```
#include <string>

class Parent {
public:
  void sleep() {
  }
};

class Child: public Parent {
private:
  std::string classes[10];
public:
  void gotoSchool(){}
};

int main( )
{
  Parent *pParent = new Parent;
  Parent *pChild = new Child;

  Child *p1 = (Child *) pParent;  // type cast #1
  Parent *p2 = (Child *) pChild;  // #type cast 2
  return 0;
}
```

Type cast #1 is not safe because it assigns the address of a base-class object (**Parent**) to a derived class (**Child**) pointer. So, the code would expect the base-class object to have derived class properties such as **gotoSchool()** method, and that is false. Also, **Child** object, for example, has a member **classes** that a **Parent** object is lacking.

Type case #2, however, is safe because it assigns the address of a derived-class object to a base-class pointer. In other words, public derivation promises that a **Child** object is also a **Parent** object.