// namespace1.cpp

```
// using global variables

/* Include Files */
#include <iostream>
using namespace std;

int myInt=1;  //global
int main()
{
 int myInt=2; // local
 cout << "The local variable myInt is " << myInt << endl;
 cout << "The global variable myInt is " << ::myInt << endl;
 }
```

output

```
The local variable myInt is 2
The global variable myInt is 1
```

// namespace2.cpp

```
// creating namespaces

#include <iostream>

using namespace std;

//defines globalType to be a namespace with four members
namespace globalType
{
        const int n = 10;
        const double rate = 7.50;
        int count = 0;
        void printResult(void);
}
int main()
{

// accessing namespace members

       cout << globalType::rate << endl;

       globalType::printResult( );

       cout << "End of execution" << endl;
       return 0;
 }

 void globalType::printResult( )
 {
 cout << "I'm in the print Result function" << endl;
 }
```

output

```
7.5
```

I'm in the print Result function
End of execution

// namespace3.cpp

```
// illustrates using the namespace std

/* Include Files */
#include <iostream>

int main()
{
		std::cout << "C++ is an improved C" << std::endl;
}
```

Output

C++ is an improved C

// namespace4.cpp

```
// illustrates creating namespaces to scope variables

/* Include Files */
#include <iostream>

namespace firstNameSpace
	{
		int nameSpaceVariable=5;
	}
namespace secondNameSpace
	{
		int nameSpaceVariable=10;
	}


int main()
{
		std::cout << "The value of nameSpaceVariable is " <<
				firstNameSpace::nameSpaceVariable<< std::endl;
		std::cout << "The value of nameSpaceVariable is " <<
				secondNameSpace::nameSpaceVariable<< std::endl;
}
```

output

The value of nameSpaceVariable is 5
The value of nameSpaceVariable is 10

// namespace5.cpp

```
// creating namespaces

#include <iostream>

using namespace std;
```

```
//defines globalType to be a namespace with four members
namespace globalType
{
        const int n = 10;
        const double rate = 7.50;
        int count = 0;
        void printResult(void);
}

using namespace globalType;

int main()
{

// accessing namespace members
// no need to preface members

       cout << rate << endl;

       printResult( );

       cout << "End of execution" << endl;
       return 0;
 }

 void globalType::printResult( )
 {
 cout << "I'm in the print Result function" << endl;
 }
```

output

7.5
I'm in the print Result function
End of execution

```
// namespace6.cpp
// creating namespaces

#include <iostream>

using namespace std;


//defines globalType to be a namespace with four members
namespace globalType
{
        const int n = 10;
        const double rate = 7.50;
        int count = 0;
        void printResult(void);
}
```

```
using globalType::rate;


int main()
{

// accessing namespace members
// no need to preface members

       cout << rate << endl;

       // printResult( );

       cout << "End of execution" << endl;
       return 0;
 }

 void globalType::printResult( )
 {
 cout << "I'm in the print Result function" << endl;
 }
```

output

7.5

I'm in the print Result function

End of execution

//namespace7.cpp

```
// illustrates using the namespace std

/* Include Files */
#include <iostream>
using namespace std;

int main()
{
              cout << "C++ is an improved C" << endl;
}
```

output

C++ is an improved C

//namespace8.cpp

```
// illustrates using the namespace std

/* Include Files */
#include <iostream>
using std::endl;

int main()
{
              std::cout << "C++ is an improved C" << endl;
}
```

output

C++ is an improved C

// namespace9.cpp

```
// illustrates using the namespace std

/* Include Files */
#include <iostream>

int main()

{
              int inputInt=9;
              using std::cout;
              using std::endl;
              cout << "please enter an integer" << endl;
             cin >> inputInt; cin is undefined - compilation error would occur
             return 0;
}
```

}

no output

// namespace10.cpp

```
// illustrates using the namespace std

/* Include Files */
#include <iostream>

using namespace std;

int main()
{

              int inputInt=9;
              cout << "please enter an integer" << endl;
              cin >> inputInt;
              cout << "input allowed inputInt is " << inputInt << endl;
              int cin = 3;
              cout << "please enter another integer" << endl;
              cin >> inputInt; // compilation error
              cout << "cin was redefined - no input allowed" << endl;
              cout << inputInt << endl;
              cout << "please enter still another integer" << endl;
              std::cin >> inputInt;
              cout << "even though cin was redefined -  input was allowed" << endl;
              cout << inputInt << endl;
              return 0;
}
```

Output

No output

// namespace11.cpp

```
// illustrates using the namespaces
#include <iostream>
using namespace std;
int myVariable=7;
namespace myNameSpace
{
	char myVariable='t';
}
using namespace myNameSpace;

int main()
{
	double myVariable=12.0;
	cout << "myVariable defined in main " << myVariable << endl;
	cout << "global myVariable " << ::myVariable << endl;
	cout << "myVariable defined in myNameSpace " <<  myNameSpace::myVariable << endl;
	cout << "End of execution" << endl;
	return 0;
 }
```

output

```
myVariable defined in main 12
global myVariable 7
myVariable defined in myNameSpace t
End of execution
```

pointer1.cpp

```
//   Synopsis  -  Prints the address of a variable.
//
//   Objective -  Demonstrates pointers as addresses in memory.
//             Gives the syntax of declaring a pointer to an
//             integer and one technique for initializing a
//             pointer.

#include <iostream>
using namespace std;

int main()
{
	int myInt=4;
	int* ptrToMyInt;

	// what should the value be???
	cout << "The ptrToMyInt contains "<< ptrToMyInt << endl ;

	ptrToMyInt = &myInt;

	cout << "The address of MyInt is " << &myInt << endl ;
	cout << "The ptrToMyInt contains " << ptrToMyInt << endl ;
	cout << "The address of ptrToMyInt is " << &ptrToMyInt << endl ;
```

```
}
```

output

```
The ptrToMyInt contains 0x28ff68
The address of MyInt is 0x28ff2c
The ptrToMyInt contains 0x28ff2c
The address of ptrToMyInt is 0x28ff28
```

// pointer2.cpp

```
//   Objective -  Illustrates what is meant by a pointer-to-int
//             being a separate data type.

#include <iostream>
using namespace std;

int main()
{
	int firstInt = 10;
	int secondInt = 20;
	int* intPtr;

	intPtr = &firstInt;
	cout << "The value of firstInt is " << firstInt << endl; //direct addressing
	cout << "The value of firstInt is " << *intPtr << endl; //indirect addressing
	cout << "The address of firstInt is " << &firstInt << endl;
	cout << "The address of firstInt is " << intPtr << endl;
	cout << "The address of intPtr is " << &intPtr << endl;
  //   cout << "The value of firstInt is " << *firstInt << endl; //???????????

	*intPtr=30;
	cout << "The value of firstInt is " << firstInt << endl; //direct addressing

	intPtr = &secondInt;
	firstInt = 6* *intPtr;
	cout << "The value of firstInt is " << firstInt << endl; //direct addressing

// use of & and * in the same statement
	// equivalent to firstInt=secondInt
	firstInt = *&secondInt;
	cout << "The value of firstInt is " << firstInt << endl; //direct addressing
}
```

output

```
The value of firstInt is 10
The value of firstInt is 10
The address of firstInt is 0x28ff2c
The address of firstInt is 0x28ff2c
The address of intPtr is 0x28ff24
The value of firstInt is 30
The value of firstInt is 120
The value of firstInt is 20
```

// pointer3.cpp

```
//   Objective - pointer arithmetic
//
```

```cpp
#include <iostream>
using namespace std;

int main()
{
	int myInt;
	int* intPtr = &myInt;
	float myFloat;
	float* floatPtr= &myFloat;
	double  myDouble;
	double* doublePtr = &myDouble;

	cout << "The address of myInt is " << intPtr << endl;
	cout << "The address of next integer is " << ++intPtr  << endl;
	cout << "The address of myFloat is " << floatPtr << endl;
	cout << "The address of next float is " << ++floatPtr  << endl;
	cout << "The address of myDouble is " << doublePtr << endl;
	cout << "The address of next double is " << ++doublePtr  << endl;
}
```

output

```
The address of myInt is 0x28ff20
The address of next integer is 0x28ff24
The address of myFloat is 0x28ff1c
The address of next float is 0x28ff20
The address of myDouble is 0x28ff10
The address of next double is 0x28ff18
```

// pointer4.cpp

```cpp
//more pointers

#include <iostream>
using namespace std;

int main()
{
	char* charPtr;
	char c[]= {'C','S', ' ','1', 'C', '\0'};
	charPtr = c ;

	cout << "charPtr " << charPtr << endl;   // look at pointer3.cpp
	cout << "charPtr+1 " << charPtr+1  << endl;
	cout << "*charPtr " << *charPtr << endl;
	cout << "*charPtr + 1 " << *charPtr+1 << endl;
	cout << "*charPtr + 1 " << char(*charPtr+1) << endl;
	cout << "*(charPtr + 1) " << *(charPtr+1) << endl;
	cout << "*charPtr + 3 " << *charPtr + 3 << endl;
	cout << "*&charPtr " << *&charPtr << endl;
	cout << "&charPtr " << &charPtr << endl;
	cout << "&charPtr + 2 " << &charPtr + 2 << endl; // 32 bit addressing
}
```

output

```
charPtr CS 1C
```

```
charPtr+1 S lC
*charPtr C
*charPtr + 1 68
*charPtr + 1 D
*(charPtr + 1) S
*charPtr + 3 70
*&charPtr CS lC
&charPtr 0x28ff2c
&charPtr + 2 0x28ff34
```

// pointer5.cpp

```
//
//   Synopsis   -  Prints information about the address, the
//       sizeof(), and the contents of an array, using
//              both array and pointer notation.
//
//  Objective -  To illustrate the relationship between pointers
//              and arrays and to demonstrate some of the
//       different methods to access array elements.
//
#include <iostream>
using namespace std;

int main()
{
       char   demoarray[5] = {'D', 'E', 'M', 'O', '!'};
       char   *demoptr = demoarray;
       int    i;

       cout <<  "sizeof( demoarray ) is " << sizeof( demoarray ) << endl;
       cout <<  "sizeof( demoarray[0] ) is " <<      sizeof( demoarray[0] ) << endl
;
       cout <<  "sizeof(demoptr) is " <<  sizeof( demoptr ) << endl ;
       cout <<  "sizeof( *demoptr ) is " << sizeof( *demoptr ) << endl ;

              for ( i = 0; i < 5; i++, demoptr++ )
              {
              cout << "i is " <<  i;
              cout << " demoarray [i] is " << demoarray[i] ;
              cout << " *( demoarray + i ) is " << *( demoarray + i ) ;
              cout << " *demoptr is "  <<  *demoptr << endl;
              }
}
```

output

```
sizeof( demoarray ) is 5
sizeof( demoarray[0] ) is 1
sizeof(demoptr) is 4
sizeof( *demoptr ) is 1
i is 0 demoarray [i] is D *( demoarray + i ) is D *demoptr is D
i is 1 demoarray [i] is E *( demoarray + i ) is E *demoptr is E
```

i is 2 demoarray [i] is M *(demoarray + i) is M *demoptr is M
i is 3 demoarray [i] is O *(demoarray + i) is O *demoptr is O
i is 4 demoarray [i] is ! *(demoarray + i) is ! *demoptr is !

// pointer6.cpp

```
//   Objective -  more pointers and addresses
#include <iostream>
using namespace std;
int main()
{
	int firstInt=10;
	int secondInt=20;
	int* firstPtr=&firstInt;
	int* secondPtr=&secondInt;
	int *intPtr;
	char myChar='Z';
	char * charPtr = &myChar;


	  // firstPtr=firstInt; cannot assign an integer to an integer pointer
	  // firstInt=firstPtr; cannot assign an integer pointer to an integer
	  // intPtr = charPtr;  character pointer cannot point a integer
	  intPtr = (int *)charPtr; // uses a cast
	  cout << "After the cast " << *intPtr << endl;  // need to be careful
	  cout << "After the cast " << char(*intPtr) << endl;

	       cout <<  "The value of firstInt " << **&firstPtr << endl;

	       *&secondInt*=*firstPtr;
	       cout <<  "The value of secondInt is " <<  secondInt << endl;

// NULL pointers
	       firstPtr=NULL; // Null pointer
	       firstPtr=0; // Null and 0 are the only non addresses allowed
	       if (firstPtr == NULL)
	              cout << "NULL " << endl;

// print addresses
	  // print the address of secondInt
	  cout <<  "The address of secondInt is " << secondPtr << endl;
	  // print the address of secondInt
	  cout <<  "The address of secondInt is " << &secondInt << endl;
	  // print the address of secondPtr
	  cout <<  "The address of secondPtr is " << &secondPtr << endl;
}
```

output

```
After the cast 687808602
After the cast Z
The value of firstInt 10
The value of secondInt is 200
NULL
```

```
The address of secondInt is 0x28ff20
The address of secondInt is 0x28ff20
The address of secondPtr is 0x28ff18
```

```
//                    pointer7.cpp
//
//   Synopsis  -  Prints the address of a variable.
//
//   Objective -  Demonstrates pointers as addresses in memory.
//                Gives the syntax of declaring a pointer to an
//                integer and one technique for initializing a
//                pointer.

#include <iostream>
using namespace std;

int main()
{
      int myInt=4;
      int* ptrToMyInt;

      ptrToMyInt = NULL;
      cout << "Deferencing NULL "<< *ptrToMyInt << endl ;
}
```

output

General exception

```
//                    pointer8.cpp
//
//   Synopsis  -  Accepts input of int values (an inventory) from
//                standard input into an array and displays them
//       on standard output.
//
//   Objective -  To illustrate passing an array as a parameter
//       to a function.
//
 #include <iostream>
using namespace std;

#define MAXIMUM     20

void printInventory( int[], int );
int inputInventory( int *, int );

int main()
{
      int inventory[MAXIMUM];
      int numItems;

      cout << "Please enter the number of items in stock" << endl;
```

```
	cout <<  "Enter -1 when you are done" << endl;

	numItems = inputInventory( inventory, MAXIMUM );

	printInventory( inventory, numItems );
}

//*****************************    inputInventory()    *********/
//   Accepts input of MAXIMUM or less values of type int and stores
//   them in an array.  Returns the number of items entered.
//   Checks for overflowing the end of the array.
//
int inputInventory( int *inventory, int maxNum )
{
	int index = -1;

	while ( index < maxNum-1 && *(inventory + index) != -1 )
	{
		index++;
		cin >> *(inventory + index);
	}

	if ( index == maxNum - 1 )
		cout <<  "No room for more items.\n" << endl;
	return (index );
}

//*****************************   printInventory()    *********/
//   Displays the values in an array of ints.  The parameter
//   numItems indicates the number of meaningful items in the
//   array.
//
void printInventory( int inventory[], int numItems )
{
	int index;

	for ( index = 1; index <= numItems; index++ )
	 {
		cout << "Item number " << index << " " << endl;
		cout << "Number on hand "  << inventory[index - 1] << endl;
	 }
}
```

output

Please enter the number of items in stock
Enter -1 when you are done
7
88
66
-1
Item number 1

Number on hand 7
Item number 2
Number on hand 88
Item number 3
Number on hand 66

// pointer9.cpp

```
// comparing passing parameters by copy, reference, and pointers
#include <iostream>
using namespace std;
int testFunction(
	int  &intRef,
	int * intPtr,
	int myInt,
	char arrayOfChars[]);
int main()
{
	char arrayOfChars[] = {'x','y','z'};
	int firstInt=5;
	int secondInt=10;
	int thirdInt=25;
	int & intRef=firstInt;
	int * secondIntPtr=&secondInt;

	cout << testFunction(intRef,secondIntPtr,thirdInt,arrayOfChars) << endl;
	cout << "In main " << firstInt << " " << secondInt <<" " << thirdInt << endl;
	cout << "Cell in the array (main) is " << arrayOfChars[0] << endl;
	return 0;
}
int testFunction(
	int  &intRef,
	int * intPtr,
	int myInt,
	char arrayOfChars[])
{
	myInt=50 ;
	intRef++;
	++*intPtr;
	*arrayOfChars = 'M';
	cout << "In the function " << intRef << " " << *intPtr << " " << myInt <<
endl;
	cout << "Cell in the array (testfunction) is " << *arrayOfChars << endl;
	return    intRef**intPtr;
}
```

output

In the function 6 11 50
Cell in the array (testfunction) is M
66
In main 6 11 25

Cell in the array (main) is M

// pointer10.cpp

```
// comparison of reference and pointer variables
#include <iostream>
using namespace std;

int main()
 {
	int firstInt;
	int secondInt;
	int* intPtr ; // need not be initialized
	int& intReference = secondInt; // must be initialized without &

	intPtr = &firstInt;  // & required
	*intPtr = 8; // * required
	cout << "The value of firstInt is " <<  firstInt << " " << *intPtr
		<< endl;

	intReference = 7; // without the dereference operator (*)
	cout << "The value of secondInt is " <<  secondInt << " " << intReference
		<< endl;
 }
```

output

The value of firstInt is 8 8
The value of secondInt is 7 7

// pointer11.cpp

```
// more comparisons of reference and pointer variables
// reference variable are constant pointers while pointer variables are not
#include <iostream>
using namespace std;

int main()
 {
	int firstInt;
	int secondInt;
	int* intPtr ; // need not be initialized
	int& intReference = secondInt; // must be initialized without &

	intReference = firstInt ; // what's wrong with this statement??
	intReference = 7; // without the dereference operator (*)
	cout << "The value of the integers are " <<  firstInt << " " << secondInt
		<< endl;
```

```
        intPtr = &firstInt;  // & required
        *intPtr = 8; // * required
        cout << "The value of firstInt is " <<  firstInt << " " << *intPtr
                << endl;
        intPtr = &secondInt;  // & required
        *intPtr = 9; // * required
        cout << "The value of the integers are " <<  firstInt << " " << secondInt
                << endl;
 }
```

output

```
The value of the integers are 1439460186 7
The value of firstInt is 8 8
The value of the integers are 8 9
```

//pointer12.h

```
class classExample
{
public:
        void setX(int a);
                //Function to set the value of the data member x
                //Post: x = a;
        void print() const;
                //Function to output the value of x

private:
        int x;
};
```

//pointer12i.cpp

```
#include <iostream>
#include "pointer12.h"

using namespace std;

void classExample::setX(int a)
{
        x = a;
}

void classExample::print() const
{
        cout<<"x = "<<x<<endl;
}
```

//pointer12.cpp

```
#include <iostream>
#include "pointer12.h"

using namespace std;

int main()
{
        classExample *cExpPtr;           //Line 1
        classExample cExpObject;  //Line 2
```

```
        cExpPtr = &cExpObject;          //Line 3

        cExpPtr->setX(5);               //Line 4
        cExpPtr->print();               //Line 5

        return 0;
}
```

output

x = 5

// dynamic1.cpp

```
// dynamic allocation of memory
#include <iostream>
using namespace std;

int main()
{
        int* intArray;
        int*  numberOfCells;
        int sum = 0;
        int index;

        cout <<      "An array will be created dynamically" << endl;
        cout <<      "Input an array size n followed by n integers:  ";

        numberOfCells = new int;
        cin >> *numberOfCells;
        intArray = new int[*numberOfCells];  // get space for n ints

        for (index = 0; index < *numberOfCells; ++index)
        {
                cout << "Please enter an integer\n" ;
                cin >> intArray[index];
        }
        // find the sum of all the elements in the dynamically created array
        for (index = 0; index < *numberOfCells; ++index)
        {
                sum += intArray[index];
        }

        cout << "Number of elements:" <<  *numberOfCells << endl ;
        cout << "Sum of the elements:" <<  sum << endl ;
        return 0;
        // memory is freed upon termination
}
```

output

An array will be created dynamically
Input an array size n followed by n integers: 4
Please enter an integer
5

Please enter an integer
8
Please enter an integer
9
Please enter an integer
12
Number of elements:4
Sum of the elements:34

// dynamic2.cpp

```
// dynamic allocation of memory
#include <iostream>
#include <string.h>
using namespace std;
int main()
{
	char* charArray;
	int*  numberOfCells ;

	cout <<	"An array will be created dynamically" << endl;
	numberOfCells = new int;
	cin >> *numberOfCells;
	charArray = new char[*numberOfCells];

	strcpy (charArray,"This is a test");

	charArray[2]='i'; //can use array notation

	cout << "Number of memory location allocated was " << *numberOfCells << endl;
	cout << charArray << endl;

	delete numberOfCells;
	delete [] charArray;

	// memory was released....

	cout << "Number of memory location allocated was " << *numberOfCells << endl;
	cout << charArray << endl;
}
```

output

```
An array will be created dynamically
23
Number of memory location allocated was 23
This is a test
Number of memory location allocated was 11740256 // Risky
This is a test// Risky
```

// dynamic3.cpp

```
// inaccess1ble memory location
#include <iostream>
```

```
using namespace std;

int main()
 {
       int* intPtr1;
       int* intPtr2;

       intPtr1 = new int;
       *intPtr1 = 8;
       intPtr2 = new int;
       *intPtr2 = 7;
       intPtr1 = intPtr2;      // here the 8 becomes inaccessible
       cout << *intPtr1 << " " << *intPtr2 << endl;
       delete intPtr2;  // intPtr1 is a dangling pointer
 }
```

output

7 7

// dynamic4.cpp

```
// illustrates automatic objects
#include <iostream>
using namespace std;
class AutomaticObject
{
       public:
              AutomaticObject(int objectCount);
              ~AutomaticObject();
 private:
       int objectCount;
} ;
AutomaticObject::AutomaticObject(int count)
       {
                objectCount=count;
                cout << "Start trace " << objectCount << endl;
        }
AutomaticObject::~AutomaticObject()
  {
              cout << "End trace " << objectCount << endl;
  }
void automaticFunction();

int main()
{
       AutomaticObject Object1(1);
       automaticFunction();
       AutomaticObject Object3(3);
}
void automaticFunction()
{
       cout << "Entering Function" << endl;
       AutomaticObject Object2(2);
}
```

output

```
Start trace 1
Entering Function
Start trace 2
End trace 2
Start trace 3
End trace 3
End trace 1
```

// dynamic5.cpp

```
// illustrates static objects
#include <iostream>
using namespace std;
class StaticObject
{
       public:
              StaticObject(int objectCount);
              ~StaticObject();
 private:
       int objectCount;
} ;
StaticObject::StaticObject(int count)
       {
                objectCount=count;
                cout << "Start trace " << objectCount << endl;
        }
StaticObject::~StaticObject()
  {
              cout << "End trace " << objectCount << endl;
  }
void staticFunction(); // prototype
int main()
{
       static StaticObject Object1(1);
       staticFunction();
       StaticObject Object4(4);
       staticFunction();
}
void staticFunction()
{
       cout << "Entering Function" << endl;
       static StaticObject Object2(2);
        StaticObject Object3(3);
}
```

output

```
Start trace 1
Entering Function
Start trace 2
Start trace 3
End trace 3
Start trace 4
Entering Function
Start trace 3
End trace 3
End trace 4
```

```
End trace 2
End trace 1
```

// dynamic6.cpp

```
// illustrates dynamic objects
#include <iostream>
using namespace std;
class DynamicObject
{
	public:
		DynamicObject(int objectCount);
		~DynamicObject();
 private:
	int objectCount;
} ;
DynamicObject::DynamicObject(int count)
	{
		objectCount=count;
		cout << "Start trace " << objectCount << endl;
	 }
DynamicObject::~DynamicObject()
  {
		cout << "End trace " << objectCount << endl;
  }

void objectFunction(); // prototype

int main()
{
	DynamicObject *ObjectPointer = new DynamicObject(1);
	DynamicObject *ObjectPointer2 = new DynamicObject(2);
	DynamicObject Object2(3);
	objectFunction();
	delete ObjectPointer;
	DynamicObject Object4(5);
}
void objectFunction()
{
	cout << "Entering Function" << endl;
	DynamicObject Object3(4);
}
```

output

```
Start trace 1
Start trace 2
Start trace 3
Entering Function
Start trace 4
End trace 4
End trace 1
Start trace 5
End trace 5
End trace 3
```

// dynamic7.cpp

// illustrates dynamic array of objects

```
# #include <iostream>
using namespace std;
class DynamicObject
{
	public:
		DynamicObject();
		DynamicObject(int objectCount);
		void changeObject(int count);
		~DynamicObject();
		void display();
 private:
	int objectCount;
} ;
DynamicObject::DynamicObject()
	{
		objectCount=0;
		cout << "Start trace (default constructor) " << objectCount << endl;
	 }
DynamicObject::DynamicObject(int count)
	{
		objectCount=count;
		cout << "Start trace " << objectCount << endl;
	 }
DynamicObject::~DynamicObject()
  {
		cout << "End trace " << objectCount << endl;
  }

  void DynamicObject::display()
	{
		cout << "display dynamic object " << objectCount << endl;
	 }

	 void DynamicObject::changeObject(int count)
	{
		objectCount=count;
		cout << "change dynamic object " << objectCount << endl;
	 }


int main()
{
	DynamicObject *ObjectPointer = new DynamicObject[5];  //creates 5 objects
	ObjectPointer[2].changeObject(2);
	ObjectPointer[2].display();
	cout << "deleting the array of classes" << endl;
	delete [] ObjectPointer;   // delete's all 5 objects
  DynamicObject ObjectC(3);
}
```

output

```
Start trace (default constructor) 0
```

```
Start trace (default constructor) 0
Start trace (default constructor) 0
Start trace (default constructor) 0
Start trace (default constructor) 0
change dynamic object 2
display dynamic object 2
deleting the array of classes
End trace 0
End trace 0
End trace 2
End trace 0
End trace 0
Start trace 3
End trace 3
```

// cc1.cpp

```
// Illustrates conditional compilation
#ifndef _RATIONAL_H_
#define _RATIONAL_H_
class Rational
{
public:
Rational( int numerator, int denominator );
Rational( int numerator ); // sets denominator to 1
Rational(); // sets numerator to 0, denominator to 1
private:
int n;
int d;
};
#endif
//end file Rational.h
```

// cc2.cpp

```
//   Objective -  To illustrate conditional compilation.

#include <iostream>
using namespace std;

int main()
{
      #define DEBUG 0
            #if DEBUG
                  cout << "debugs on" << endl;
            #endif
            #ifdef DEBUG
                        cout << "debugs are now on"<< endl;
            #endif
            cout << "the value of DEBUG is " << DEBUG << endl;
      }
```

output

```
debugs are now on
```

the value of DEBUG is 0

```
//                  cc3.cpp
//   Objective -  To illustrate conditional compilation.

#include <iostream>
using namespace std;
int main()
{
                    #define UNKNOWN
                    #if 0
                    cout << "can never happen" << endl;
                    #else
                    #ifdef UNKNOWN
                     cout << "UNKNOWN  defined" << endl;
                    #else
                     cout << "last chance"<< endl;
                    #endif
                    #endif
}
```

output

UNKNOWN defined

```
//                  cc4.cpp
//   Objective -  To illustrate conditional compilation.

#include <iostream>
using namespace std;

int main()
{
                    #define z 0
                    #undef z
                    #define Z 1
                    #ifdef z
                            cout << "Hello" << endl;
                    #endif
}
```

No output