Computer Programming Lab # 10

Pointers and Dynamic Memory Management II

Pointers to Objects:

```
#include <iostream>
using namespace std;
class Distance
{
private:
	int feet;
	float inches;
public:
	void getdist()
	{
		cout << "\nEnter feet : "; cin >> feet;
		cout << "Enter inches : "; cin >> inches;
	}
	void showdist() //display distance
	{
		cout << feet << "\' - " << inches << "\"";
	}
};
int main()
{
	Distance dist; //define a named Distance object
	dist.getdist(); //access object members
	dist.showdist(); // with dot operator
	Distance* distptr; //pointer to Distance
	distptr = new Distance; //points to new Distance object
	distptr->getdist(); //access object members with -> operator
	distptr->showdist();
	cout << endl;
	return 0;
}
```

Referring to Members

The question is, how do we refer to the member functions in the object pointed to by distptr? You might guess that we would use the dot (.) membership-access operator, as in
distptr.getdist();   // won’t work; distptr is not a variable
but this won’t work. The dot operator requires the identifier on its left to be a variable. Since distptr is a pointer to a variable, we need another syntax. One approach is to dereference (get the contents of the variable pointed to by) the pointer:
(*distptr).getdist(); // ok but inelegant

However, this is slightly awkward because of the parentheses. (The parentheses are necessary because the dot operator (.) has higher precedence than the dereference operator (*). An equivalent but more concise approach is furnished by the membership-access operator, which consists of a hyphen and a greater-than sign:

distptr->getdist(); // better approach

Another Approach to new:

```
int main()
{
	Distance dist = *(new Distance); // create Distance object
	dist.getdist(); //access object members
	dist.showdist(); // with dot operator

	Distance* distptr; //pointer to Distance
	distptr = new Distance; //points to new Distance object
	distptr->getdist(); //access object members with -> operator
	distptr->showdist();
	cout << endl;
	return 0;
}
```

Pointers to Pointers:

```
#include <iostream>
#include <string>
using namespace std;
class person //class of persons
{
	protected:
		string name; //person’s name
	public:
		void setName() //set the name
		{	cout << "Enter name: "; cin >> name; }

		void printName() //display the name
		{ cout << name << endl; }

		string getName() //return the name
		{	return name; }
};

void fun(person**, int); //prototype
int main() {

	int number;
	cout << "Enter the number of Person"; cin >> number;
	person** persPtr = new person*[number]; //array of pointers to persons
```

Remaining part of the code

```
int n = 0; //number of persons in array
       char choice; //input char
       do {
                     //put persons in array
                     persPtr[n] = new person; //make new object
                     persPtr[n]->setName(); //set person’s name
                     n++; //count new person
                     cout << "Enter another (y/n)? "; //enter another
                     cin >> choice; // person?
              } while (choice != 'n');

       fun(persPtr, n);
       return 0;
}
void fun(person** pp, int n) //sort pointers to persons
{
       cout << "FAST NUCES";
}
```

Deleting an Array of Pointers to Objects

At the end of the program the destructor must delete the objects, which it obtained with new in its constructor. Notice that we can’t just say delete[] hArray;

This deletes the array of pointers, but not what the pointers point to. Instead we must go through the array element by element, and delete each object individually:

```
for(int j=0; j<number; j++)
        delete persPtr[j];

delete [] persPtr;
```

Dynamic Arrays and specifying arrays’ length on run time:

One Dimensional Dynamic Array:

The statement:

int *p; declares p to be a pointer variable of type int. The statement:

```
p = new int[10];
```

One Dimensional Dynamic Array:

```
  int *intList;  int arraySize;
  cout << "Enter array size: ";  cin >> arraySize;
  intList = new int[arraySize];
```

Two Dimensional Dynamic Array:

```
int **Pointer;

Pointer=new int*[5];

for (int i=0;i<5;i++)
     Pointer[i]=new int[10];
```

Shallow versus Deep Copy and Pointers:

1. Shallow Copy

Consider the following statements:

```
int *first; int *second; first =
new int[10]; second = first; delete
[] second;
```

2. Deep Copy

```
second = new int[10]; for
(int j = 0; j < 10; j++)
second[j] = first[j];
```

Copy Constructor:

```
class Point
{
       int x, y;
       public:
              Point() {     }
              Point(const Point& p);   // copy constructor
};
Point::Point(const Point& p)  { x = p.x;     y = p.y; }
void main()
{
       Point p;             // calls default constructor
       Point s = p;         // calls copy constructor.
       p = s;               // assignment, not copy constructor.
}
```

When copy constructor is called?

The copy constructor automatically executes in three situations (the first two are described previously).

• When an object is declared and initialized by using the value of another object
• When, as a parameter, an object is passed by value
• When the return value of a function is an object

Virtual Functions and Virtual Destructor and Polymorphism:

Example??