Dear All Students
Attached are the lectures. I decided to give you these lectures to help you studying this course. I hope this could help. I apology for any inconvenience to you during this course caused by me or any member of the team that taught you this course.

Dr Ahmed Rafat
Computer Science Dept.
Faculty of Computers and Informatics
Zagazig University

Object-Oriented Programming

Dr Ahmed Rafat Abas

Computer Science Dept, Faculty of Computers and Informatics, Zagazig University

Ahmed_rafat_abas@yahoo.co.uk

Who am I ?

- Name: Ahmed Rafat Abas
- Last Scientific Certificate: Ph.D. in Computer Science, UK, 2005
- Home Town: Zagazig

Who are you ?

- Name:
- Home Town:
- What do you expect from this faculty?
- What do you expect from this course?

Course Outline

1. The Elements of C++
2. Object Oriented Programming Principles
3. Constructors and Destructors
4. Function Overloading
5. Inheritance
6. Designing an Object Oriented Program

Course Outline

7. **Derived Classes**
8. **Pointers and Dynamic Memory allocation**
9. **Polymorphism**
10. **Operator Overloading**
11. **Templates**

Lab Work

Discussing some exercises after each chapter

Evaluation

- Total Degrees: 100
- Final Exam Weight: 75 %
- Lab Work and Reports: 10 %
- Mid-Term Exam: 15 %

Important Tips

- Close mobile phones in lecture time
- No entrance to the lecture hall after the first 15 min of the start of the lecture
- Ask when you do not understand any point
- Take a brake for 15 min after the first hour

List of References

1. C++ and Object-Oriented Programming. Kip R. Irvine, 1997, No. 614.
2. Introduction to Data Structure and Algorithms with C++. Glenn W. Rowe, 2001, No. 908.

Where can I get a copy of the book of this course?

Well, I’d like to remind you that it’s totally optional to buy a book for this course, but if you like you can find it in this place until the end of next week:

المركز الاعلامى لمهمات المكاتب و الطباعة

الزقازيق – المنتزة – ميدان جامع العيداروس

ت : 2329587

Object-Oriented Programming

Dr Ahmed Rafat Abas

Computer Science Dept, Faculty of Computers and Informatics, Zagazig University

Ahmed_rafat_abas@yahoo.co.uk

The Elements of C++

Basic data structures in C++

- **int:** it represents a signed integer data type.
- **short and long:** a short integer is allocated two bytes of storage, while a long integer is allocated four bytes.
- **float and double:** both represent floating point data type. They allow different precision for defined variables.

- **char:** it represents a character data type. The defined variable is allocated one byte of storage for storing the ASCII code corresponding to the stored character.
- **unsigned:** this keyword is used as a prefix for any data type to indicate that all the defined variables take either positive values or zero.

The main() function

This function is executed in the start of the program running. Its structure is as follows:

```
int main()
{
   variable definitions
   statements
}
```

It is a good practice to put this function in a file called *main.cpp*.

Arithmetic operators

=	assignment
+, -, *, /, %	addition, subtraction, multiplication, division and modulus
++, --	increment and decrement
+=, -=, *=, /=	arithmetic assignment

Logical operators

>	greater than
<	less than
>=	greater or equal
<=	less or equal
==	equal
!=	not equal
!	Not
||	Or
&&	And

Conditional operators

- **The if statement**

```
if (condition)
   single_statement

or

if (condition)
{
   block_of_statement
}
```

• The if-else statement

```
if (condition_1)
{
   statements
}
else if (condition_2)
{
   more_statements
}
else if (condition_3)
{
   even_more_statements
}
else
{
   final_set_of_statements
}
```

- **The switch statement**

```
switch(expression)
{
  case value_1:
          statements
          break;
  case value_2:
          statements
          break;
  …
  default:
          statements
}
```

- **The conditional operator ?:**

condition ? expression_1 : expression_2;

Loops

- **The while loop**

```
while(expression)
{
   statements
}

Or

while(expression)
   single statements
```

• The do .. while loop

```
do
{
   statements
} while(expression);
```

• The for loop

```
for(expression1;expression2;expression3)
{
   statements
}
```

expression1: is an initialization expression.
expression2: is the termination test.
expression3: is executed after each iteration of the loop.

Arrays

- **One dimensional arrays**

int IntArray[100];

IntArray can be accessed as follows:
IntArray[0], IntArray[1], …, IntArray[99],

Or

*IntArray, *(IntArray+1), …, *(IntArray+99).

- **Two-dimensional arrays**

int IntArray[100][100];
IntArray can be accessed as follows:
IntArray[0][0], …, IntArray[99][99],

Or

*(IntArray[0]+0), …, *(IntArray[99]+99).

Comments

/* … */ for multi-line comments

// for a single-line comment

Input and output

```
cin >>var1>>var2;
cout <<"message"<<'\n';
cout <<var1<<var2<<endl;
```

Object-Oriented Programming

Dr Ahmed Rafat Abas

Computer Science Dept, Faculty of Computers and Informatics, Zagazig University

Ahmed_rafat_abas@yahoo.co.uk

Object Oriented Programming Principles

Programming approaches can be divided into two main types:

- **procedural approach**
- **object oriented (OO) approach**.

- **In procedural approach**

The first stage in the analysis is to determine the algorithms to be used by the program. Then, data structures are designed and modules are constructed.

- **In OO approach**

The first stage in the analysis is to determine the classes. Once the classes are constructed, the actions for each class are written as individual modules. The overall algorithm is then implemented by arranging for the individual objects to act on each other, using the actions associated with each class.

Basic definitions

- A **class** is a collection of attributes (properties and actions) which collectively describe an entity.
- An **object** is a particular instance of a class.
- **Inheritance** is an OOP principle that allows objects of different classes to be related to each other by common properties and actions.
- A **base class** contains characteristics that are shared by all derived classes from it.

- A **derived class** is a superset of the base class, which contains additional properties and actions not found in the base class.

- **Encapsulation** is the process of hiding implementation details.

A class in C++

A class can be defined as follows:

```
class classname
{
  private:
    statements
  public:
    statements
};
```

The above class declaration can be stored in a header file named as *name.h*. This header file should be included in the file *name.cpp* that contains all definitions of class functions using *#include* statement.

Definition of A class function

```
return_data_type classname::functionname(arguments)
:initializationlist
{
statements
}
```

Definition of A Global Variable

To define a global variable, which can be accessed by all functions of the program, put the statement that define this variable in the file containing the main() function outside any function definition. Then, use the *extern* statement to declare this variable in each other file, in which this variable is accessed as follows:

extern datatype variablename;

Constructors and Destructors

Constructors and destructors are member functions of any class that are automatically called when a class object is created or destroyed, respectively.

Constructor functions

The constructor function is executed when a class object is defined. It can be argumentless or with arguments. It should have the following:

- No return data type,
- The same name as the class name,
- An optional initialization list may follow the name and arguments of the constructor,
- Public declaration within the class.

Example:

```
class name
{
  private:
    variable declarations
  public:
    name();     // constructor
…
};

…
name::name(arguments):initialization list
{
  statements
}
```

Example:

```
account();     //declaration

account::account():Balance(0.0), Accountnumber(-1)
{}   //definition
```

Example:

```
account(float newBalance=0.0, int newAccount=-1);
     // declaration

account::account(float newBalance, int newAccount):
Balance(newBalance), AccountNumber(newAccount)
{}   // definition
```

Example

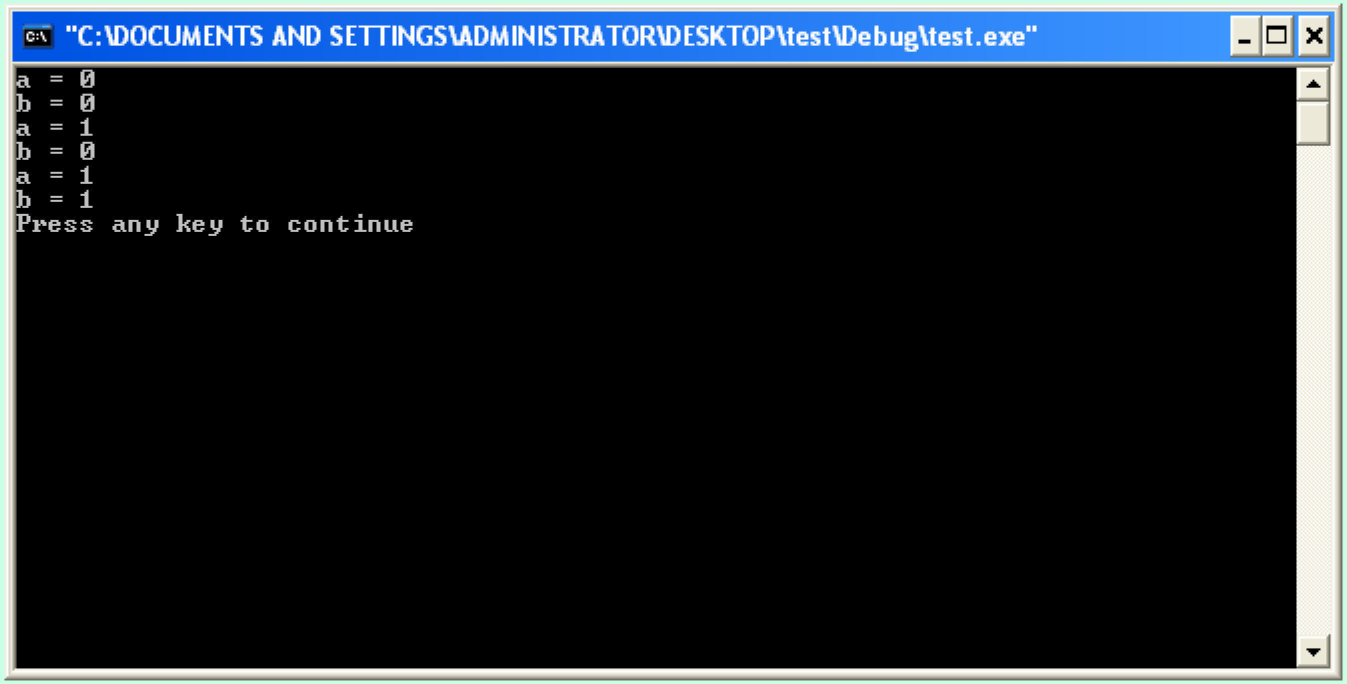
```
class test {         // test.h
private:
     float a;
     float b;
public:
     test(float a=0,float b=0);
};

#include "test.h“   // test.cpp
#include <iostream.h>

test::test(float c, float d):a(c),b(d)
{
     cout << "a = "<<a<<endl;
     cout << "b = "<<b<<endl;
}
```

```
#include "test.h“   // main.cpp
#include <iostream.h>

main()
{
    test obj1;
    test obj2(1);
    test obj3(1,1);
    return;
}
```



"C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\DESKTOP\test\Debug\test.exe"

```
a = 0
b = 0
a = 1
b = 0
a = 1
b = 1
Press any key to continue
```

Dynamic memory allocation

- **Pointers in C++**

Example:

```
int * IntPointer;        // definition of a pointer to int
int Integer_variable;
Integer_variable=5;
IntegerPointer=& Integer_variable;
cout << "the address of integer variable is "<< InPointer<<
   "but the quantity of this variable is "<<*IntPointer;
```

- The *new* operator

The way of declaring and assigning value and memory locations to pointers in the previous example is static. To declare and assign values and memory locations to pointers dynamically the *new* operator should be used.

Example:

```
float * RealPointer = new float;
int * InPointer = new int(24);
*RealPointer = 3.34;
cout << "pointer: "<< RealPointer << " ; value: "<<
    *RealPointer << endl;
cout << " Pointer: "<< IntPointer << " ; value: "<<
    *IntPointer <<endl;
```

In this example, RealPointer is assigned an address of a location in memory that is reserved for a float number. Also, IntPointer is assigned an address of a location in memory that is reserved for an int number whose value is 24.

• Dynamic allocation of arrays

```
int * IntArray = new int[100];
```

This statement reserves a memory space for 100 int values and returns the address of the first memory location to be assigned to IntArray. It is equivalent to:

```
int IntArray[100];    // static definition
```

but the size of the array can be a variable determined during program running in the dynamic memory assignment.

However, arrays can be initialized statically but not dynamically.

```
int Num[10]={1,2,3,4,5,6,7,8,9,10};
```

Argument passing in function calls

- **Passing by value**

```
int IntFunction(int Arg1, float Arg2) {…}
/*definition of  a function that receives  its
arguments by value.*/
…
ReturnInt=IntFunction(Int1,Float2);
/*function calling for passing variables by
value.*/
```

- **Passing by reference**

There are two ways for passing arguments by reference:

int RefFunction(**int &**Arg1, **float &**Arg2){…}

…

ReturnInt=RefFunction(Int1, Float2);

Or

int PointerFunction(**int** *Arg1, **float** *Arg2){…}

…

ReturnInt=PointerFunction(**&**Int1, **&**Float2);

The second way is clearer and more understandable from code. Passing variable by reference saves memory and time but requires attention about accessing the variables.

- **The *const* keyword**

const int Num = 10;   //define a constant int variable.

int ConstArgs(**const int &**Arg1,**const float &**Arg2){…}

/*variables Arg1 and Arg2 are read only variables in this function. */

Object-Oriented Programming

Dr Ahmed Rafat Abas

Computer Science Dept, Faculty of Computers and Informatics, Zagazig University

Ahmed_rafat_abas@yahoo.co.uk

Destructors

It is necessary to release any memory allocated using the new operator. This can be done using the C++ delete keyword. This process could be done in the destructor function of every class.

```
class Atm
{
private:
    variable declarations
public:
    atm();
~atm();
…
};
atm::atm()
{
int * intpointer=new int[10];
}
atm::~atm()
{
delete [] intpointer;
}
```

It is worth mentioning that A destructor function:

- has no return type,
- takes no arguments,
- Also, the operand of the delete keyword must be a pointer.

Text File Streams

- **Reading from an Input File Stream**

If a program requires a large amount of input, it is necessary to store the input data in a text file and let the program get its input from there. An input stream can be read from a text file by:

- declaring an **ifstrearm** object,
- passing a filename to its constructor,
- checking the stream variable for a non-zero value, which

indicates that the file is opened successfully.

Once a file stream is opened, it behaves exactly the same as **cin**, the standard input stream.

Both **iostrearm.h** and **fstrearm.h** should be included when declaring a file stream.

Example:

```
ifstream infile ("payroll.dat");
```

Example:

```
#include <iostream.h>
#include <fstream.h>
int main ()
{
long employeeId;
int hoursWorked;
float payRate;
ifstream infile ("payroll. dat") ;
if( !infile )
 cerr « "Error: cannot open input file.\n";
else
 infile » employee Id » hoursWorked » payRate;
return 0;
}
```

- The expression !infile calls a function in infile's class that returns 1 if the file is not open.
- When infile is destroyed, the **ifstream** destructor closes the input file. In this example, this happens when **main ()** terminates.
- A file stream can be closed by calling the close member function as follows:

```
infile.close();
```

- A stream can also be opened for input by calling the open function and passing the **ios::in** flag as an argument:

```
infile.open("payroll.dat", ios::in );
```

- It is necessary to check for end of file using the **eof** function when reading multiple records from a file stream.

Example:

Assuming that infile has already been opened, the following loop reads all of its records:

```
while (!infile.eof())
infile » employeeId » hoursWorked » payRate;
```

Moving the Input File Position

Every file stream keeps track of its current position within the file. You might use the seekg stream function to manipulate the input position. This is useful for reading a certain record in a file with fixed-length records.

Example:

Suppose a file is created such that each record was exactly ten bytes long (including the two-byte end of line sequence used by our computer system):

```
11110000
22220000
33330000
44440000
55550000
```

To move to the third record, we would call seekg to set the file position at 20 bytes from the beginning of the file as follows:

```
recordLen = 10;
recNum = 3;
offset = (recNum - 1) * recordLen;
infile.seekg( offset, ios::beg );
char buf [10];
infile.get( buf, 10 );
cout « buf « endl;
```

Note: Random access with text files has a problem that is computer systems vary in the way they implement newline. Some use a single character, others use two. The offset argument passed to the seekg function has three options as follows:

ios::beg	Offset is from the beginning of the stream
ios::end	Offset is from the end of the stream
ios::cur	Offset is from the current position

The offset can be either positive or negative.

Example:

- The following statement moves the pointer

ten bytes prior to the end of the file:

```
infile.seekg( -10, ios::end );
```

- The following moves the position backward one byte from the current position:

```
infile.seekg( -1, ios::cur );
```

The tellg function returns the current file position. Its return value is of type streampos, which is usually a long integer.

Example:

```
streampos p = infile.tellg();
```

Writing to an Output File Stream

An output file stream is created in a way approximately similar to an input file stream. To create an output file stream, you should define an **ofstream** object and pass it a filename.

Example:

The ofile is opened for output as follows:

```
ofstream ofile ( "payroll.dat", ios::out) ;
if ( !ofile )
 cerr « "Error: cannot create the output file.\n";
```

The expression !ofile returns 1 if the output stream could not be created. It is important to make this test before writing to the output file.

Example:

This example creates a file called salary.dat and writes three numbers to it, separated by spaces:

```
#include <iostream.h>
#include <fstream.h>
long employeeId;
float hoursWorked;
float payRate;
ofstream ofile ( "salary.dat", ios::out) ;
if( !ofile )
 cerr « "Error: cannot create output file.\n";
else
  ofile «employeeId « ' ' « hoursWorked «' ' « payRate « endl;
```

As with input file streams, the output file is automatically closed when the stream object ofile is destroyed.

Example:

An output file stream can be explicitly opened with the following statement:

```
ofile.open( "payroll.dat", ios::out );
```

By default, a file stream opened for output destroys any existing file by the same name.

Example:

You can open a file in the append mode by using the **app** flag as follows:

```
ofile.open ( "payroll. dat", ios:: app );
```

No error will be generated if the file does not exist.

Object-Oriented Programming

Dr Ahmed Rafat Abas

Computer Science Dept, Faculty of Computers and Informatics, Zagazig University

Ahmed_rafat_abas@yahoo.co.uk

Function Overloading

Function overloading allows multiple functions, which generally do the same task but on different data types, to have the same name.

Example:

In sorting applications, the swap function is used to exchange the values of two objects. The swap function is overloaded to work with different argument types as follows:

```
void swap ( unsigned long &, unsigned long & );
void swap ( double &, double & );
void swap ( char *, char * );
void swap ( Point &, Point & );
```

Function overloading allows the same function name to be reused, as long as each function has a different signature. A function signature consists of a function's name, as well as the order and type of its parameters. It does not include the function's return type.

Overloading Constructors

Function overloading is commonly used for class constructors to provide alternate ways of constructing objects.

Example:

The Figure class contains a default constructor, a constructor taking a single Point, and a constructor taking an array of Point objects and a counter as follows:

```
class Point {
public:
   Point( int x = 0, int y = 0 );
} ;
class Figure {
public:
   Figure() ;
   Figure (const Point & center);
   Figure (const Point vertices[], int count);
} ;
```

These constructors allow one to create an array of Figures, a Figure specified by its center point, and a Figure specified by an array of vertices as follows:

```
Figure figl[50];
Point center( 25, 50 );
Figure fig2( center );
const int VCount = 5;
Point verts[VCount];
Figure fig3( verts,VCount );
```

Inheritance

To make use of the already written code, the inheritance mechanism in OOP allows the programmer to derive new classes that inherit all the fields (properties and actions) from another class called a base class.

Inheritance in C++

```
class DerivedClass : public BaseClass
{
   statements
};
```

The colon means inherit. The keyword public means that all public fields in the BaseClass are also public in the DerivedClass. If the keyword private is used instead of public, this means that the public fields in the BaseClass are private in the DerivedClass.

Example:

```
class account
{
protected:
    statements
public:
    statements
};
class chequing : public account
{
private:
    statements
public:
    chequing(float overdraft=0.0);
    statements
};
```

The keyword protected means that all these fields are private in this class and in the derived classes from it.

Constructors and inheritance

Both the base class and the derived class can have their own constructor functions. When an object of the derived class is defined, the base class constructor is called first, followed by the constructor of the derived class.

Example:

The constructor of class chequing can be defined as follows:

```
Chequing::chequing(float overdraft) :Overdraft(overdraft){}
```

When an object of class chequing is defined, the default argumentless constructor of class account is called first, and then this constructor is called.

note:

- If you have not defined an argumentless constructor for class account but instead you have defined a constructor with arguments, an error occurs.
- The compiler defines an argumentless constructor for a class if you do not define any constructor for this class.
- The initialization list of the derived class constructor is executed before the base class constructor is called.

To overcome the above problem, the constructor of the chequing class should call the constructor of the account as follows:

```
chequing::chequing(float overdraft, float balance, int
    account_No) : Overdraft(overdraft),
    account(balance,account_No) {}
```

In the class chequing, the constructor is declared as follows:

```
chequing(float overdraft = 0.0, float balance = 0.0, int
    accunt_No = -1);
```

Function overloading in derived classes

C++ allows the functions in the base class to be overloaded in the derived classes. The functions in the derived classes are allowed to have the same signature as some functions in the base class. This means that a function in the base class and a function in a derived class can have the same name and arguments types but do different things.

Pointers and virtual functions

```
account *Accountptr;
```

- Accountptr is a pointer to a class object.
- This pointer accesses fields of the class object using the -> notation instead of the . notation.
- A pointer to a class object can be assigned an address of memory location for an object defined on this base class or any derived class from it.

Example:

```
account *Accountptr[3];
accountptr[0]=new account(50,12);
accountptr[1]= new chequing(0,25.50,15);
accountptr[2]= new savings;
```

For accessing overloaded functions in the derived classes using a pointer to an object defined on the base class, these functions should be declared in the base class as virtual. Definitions of these functions are not affected.

Example:

```
class account
{
protected:
    statements
public:
    virtual int withdraw(float amount);
    statements
};
Accountptr[0] -> withdraw(10);
Accountptr[1] -> withdraw(10);
Accountptr[2] -> withdraw(10);
```

In this code, the withdraw() function belonging to the three classes are executed in the right way.

Destructors and virtual destructors

- Where objects of the derived class are declared directly (without using pointers), the destructors are called in reverse order to the constructors (that is, derived class first, then base class).

- If dynamic allocation is used to create derived objects, virtual destructors should be used to ensure that the correct destructors are called.

Example:

```
class account
{
protected:
     statements
public:
     account(float balance=0.0, int accountnumber=-1);
     virtual ~account();
     virtual int withdraw(float amount);
…
};
class chequing:public account
{
private:
     statements
public:
     ~chequing();
     chequing(float overdraft=0.0, float balance=0.0,float accountnumber=-1,
     float admincharge=0.50);
     int withdraw(float amount);
};
account *chequeAcct=new chequing(0,25,15);
delete chequeAcct;
```

- A statement delete chequeAcct results in the chequing destructor being called first, followed by the account destructor.
- If the account destructor had not been declared virtual, it would have been the only destructor called.

Inheritance in OO design

- Inheritance is an important feature in OO design.
- To write proper OO programs, begin your analysis of the problem to be solved by listing all classes in the model.
- Then, go over your list attempting to identify those classes which are "special cases" of a more general class.
- All such classes may be derived from a base class, which contains these properties and functions common to all members of the group.
- Certain actions may have variations in each derived class, but this can be handled using the virtual function mechanism.