EE 441 Data Structures

Lecture 1
ADMINISTRATIVE

Administrative Details

- Instructor: Ece Güran Schmidt
- Office: A-402
- Email: eguran@metu.edu.tr
- Section 1 Schedule:
  - Tue. 9:40-11:30@A-206
  - Thu. 9:40-10:30@A-206

Administrative Details

- **Follow**
  - https://odtuclass.metu.edu.tr for lecture slides, extra material and announcements
  - Your e123456@metu.edu.tr email
  - Class notes and discussions
  - Syllabus is posted on https://odtusyllabus.metu.edu.tr/
- Office Hours:
  - Preferred communication mean: E-MAIL
  - Send with subject **including** ee441 (no guarantee of reply otherwise)
  - I will answer your questions any time if you come by my office provided that I am not busy at the moment

Required

- **Pre-requisite:** CENG 230 or equivalent.
- **Some references:**
- Preiss, B.R., Data Structures and Algorithms with Object-Oriented Design Patterns in C++, Wiley, 1999
- Ford&Topp, Data Structures with C++, Prentice-Hall, 1996;
- Shaffer, C., Data Structures & Algorithm Analysis in C++, Dover Publications, 2012 (http://www.e-booksdirectory.com/details.php?ebook=7307).

Course Objective (Why should you take this course?)

- You will work with software and hardware systems
- Software specific gains:
  - How to organize data, how to design algorithms
- Useful for both hardware and software:
  - Modular system design
  - Interfaces between modules
  - Complexity and design trade offs (space, time, cost)

Course Outline

- Introduction to OOP
- Abstract Data Types, Classes & Objects
- Arrays, Pointers
- Algorithm and Problem Complexity
- Stacks
- Queues
- Dynamic memory management
- Linked Lists
- Trees, B-Trees
- Graphs
- Sorting and Hashing Algorithms

Grading and Policies

- Grading:
  - ONE Midterm: 30%
  - Final: 40%
  - Programming assignments + quizzes + attendance: 30%
- Course Policies
  - Late submissions of assignments will be penalized according to the following policy:
  - It is **NOT** allowed to prapare homeworks as groups. METU honor code is essential.
  - To **COPY** or **BEING COPIED** will result in grade **ZERO**.

Grading and Policies

- Make-ups are to be given to those having medical report approved by METU medical center.
- Students who miss all the exams, or who do not submit any HW will be graded as NA.
- It is not allowed:
  - to use calculators, cell phones or other electronic devices
  - going outside

during exams.

EE 441 Data Structures

Lecture 1
Introduction

Data Structures

- A systematic way of organizing and accessing data so that it can be used efficiently.
  - Examples: queue, stack, linked list, tree
- Associated algorithms to perform operations that maintain the properties of the data structure.
  - Examples: search, insert, balance,
- A well-designed data structure allows a variety of critical operations to be performed on using as little resources, both execution time and memory space, as possible.

Different ways of programming: Unstructured Programming

- One main program
- Data is global throughout the whole program
- Simple for small projects
- Problems:
  - If the same statement sequence is needed at different locations within the program, the sequence must be copied.
  - Disadvantageous once the program gets sufficiently large

Example

- Unstructured

```
main();
{
int a,b,temp,result;
a=5;
b=6;
temp=(a+b)/2;
result=temp*temp;
int c=8;
int d=10;
temp=(c+d)/2;
result=temp*temp;
};
```

Different ways of programming: Procedural Programming

- Programs are divided into pieces which can be combined later
- These pieces are written by programmers
- Other users construct their own programs using these pieces
- Abstraction: separates what the user needs to know and the programmer needs to know
  - users can think in high-level terms
  - users don't need low-level details about the piece implementations

Example

- Unstructured

```
main();
{
int a,b,temp,result;
a=5;
b=6;
temp=(a+b)/2;
result=temp*temp;
int c=8;
int d=10;
temp=(c+d)/2;
result=temp*temp;
};
```

- Procedural

```
int avgsq(int x,
int y)
{int t=(x+y)/2;
return t*t;}

main();
{
int a,b,result;
a=5;
b=6;
result=avgsq(a,b);
int c=8;
int d=10;
result=avgsq(c,d)

};
```

Different ways of programming: Procedure-oriented Programming (POP)

- POP: procedural abstractions:
  - Ignore the implementation of the procedure
  - Focus on arguments and return values

```
int avgsq(int x, int y);

main();
{
int a,b,result;
a=5;
b=6;
result=avgsq(a,b);
int c=8;
int d=10;
result=avgsq(c,d);
};
```


Different ways of programming: Object-oriented Programming (OOP)

- OOP: procedural abstractions, data abstractions and encapsulation
  - Ignore the way data is represented in memory
  - Focus on operations that can be performed on data
  - Encapsulation aids the software designer by enforcing *information hiding*.
  - The implementation details are *hidden* from the user of that object.

Abstract Data Type

- Before you program you sit down and design the data with pencil and paper.
- A model used to understand the design of a data structure.
- Implementation independent data description
- Specifies:
  - contents
  - type of data stored
  - the legal operations on the data
- Viewing a data structure as an ADT allows a programmer to focus on an idealized model of the data and its operations.

ADT Format

- Name
  - Description of the data structure
- Operations
  - Construction operations
    - Initial values
    - Initialization processes
  - Other operations

Designing an ADT

- Example: A calendar software
- What kind of data organization do we need?
- What kind of procedures do we need to manipulate this data?
- We need to:
  - Represent dates in the computer
  - Print dates on the screen
  - Update dates

ADT Example

ADT Date
Data
  1≤d ≤31 (day)
  1≤m ≤12 (month)
  1900≤y ≤2100 (year)
Operations
  Constructor:
    Input :month, day, year;
    Preconditions: none;
    Process: Assign initial values to d, m, y ;
    Output: None
    Postconditions: None
  PrintDate:
    Input: none;
    Preconditions: none;
    Process: Print formatted on screen
    Output: none
    Postconditions: none

JumpYear:
  Input: year jump (j);
  Preconditions: j ≤ 100, y ≤2000
  Process: JY = y+j
  Output: JY
  Postconditions: none;

SetDate:
  Input :new month, new day, new year;
  Preconditions: (only basic check)
  1≤new day ≤31
  1≤new month ≤12 (month)
  1900≤new year ≤2100 (year)
  Process: update month day year
  Output: none
  Postconditions: none;

End ADT Date;

ADT Operation Description

- Name of the operation
- Input: External data that comes from the user of this data
- Preconditions: Necessary state of the system before executing this operation
- Process: Actions performed by the operation on the data
- Output: Data returned to client
- Post conditions: state of the system after executing this operation

Classes and Objects

- A class:
  - an actual representation of an ADT.
  - provides implementation details for the data structure used and operations
  - Members:
    - variables to store data
    - operations (methods) for data handling

Class Example

Written in C++ syntax

```
class Date
{
private:
            int month, day, year;
public:
            Date (int m=1, int d=1, int y=1900);
             void PrintDate( );
             int JumpYear(int j) const;
             void SetDate(int m, int d, int y);

        };
```


Objects

- Variables of the class type (Instances of classes)

```
int x=10;
float y;
```

```
Date Today(10,2,2018);
```

Objects

- Variables of the class type (Instances of classes)
- A class is a blueprint, or prototype that defines properties and behavior of sets of objects.
- An object:
  - a self-contained entity that consists of data
  - methods to manipulate the object’s data are defined by the object’s class
  - can be uniquely identified by its name and it defines a state which is represented by the values of its attributes at a particular time.

Object Example

class Date is declared.

```
class Date
{
private:
        int month, day, year;
public:
        Date (int m=1, int d=1, int y=1900);
        void PrintDate( );
        int JumpYear(int j) const;
        void SetDate(int m, int d, int y);

    };
```

```
Date Today(10,2,2018);
Date Tomorrow(10,3,2018);
```

C++ Classes

- Class declaration
  - Member variables
  - Member function prototypes
- Class implementation
  - Member function definitions

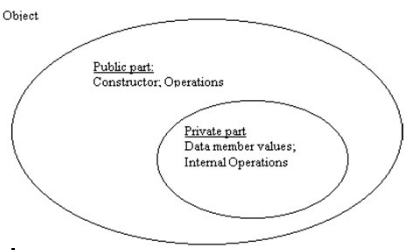
C++ Classes: Class Declaration

```
class <class_name>
{
private:
      <private data declarations>
      <private method declarations
          (prototypes)>
public:
      <public data declarations>
      <public method declarations
          (prototypes)>
};
```

```
class Date
{
private:
            int month, day, year;
public:
   Date (int m=1, int d=1, int
   y=1900);
   void PrintDate( );
   int JumpYear(int j) const;
   void SetDate(int m, int d, int
   y);

};
```

C++ Classes: Controlling Access to Members



- Members: variables and operations (methods) for data handling
- classes can protect their member variables and methods from access by other objects.
- The public and private sections in a class declaration allow program statements outside the class different access to the class members.

class
private:
Data members: $value_1$ $value_2$
Internal operations
public:
Constructor
Operation 1
Operation 2
Ooperation 3

```
class Date
{
private:
        int month, day, year;
public:
   Date (int m=1, int d=1, int
   y=1900);
   void PrintDate( );
   int JumpYear(int j) const;
   void SetDate(int m, int d, int y)};
```


Date Class Implementation

```
/* Class implementation */
/* constructor */
Date::Date(int m, int d, int y )
{
   month=m ;
   day=d;
   year=y;
}

int Date::JumpYear (int j) const
{
     return  year+j;
}
void Date::PrintDate()
{
cout<<"Month:"<<month<<" Day:"<<day<<"
  Year:"<<year<<"\n";
};
```

Date Class Method Calls

```
Date::Date(int m, int d, int y )
{
   month=m ;
       day=d;
       year=y;
}
```

Constructor:Creating an object (instance of the class) Initializing the object

MUST BE PUBLIC can be called by the main or any function that is not class member

```
Date Today(10,2,2018);
```

Object declaration creates an instance of a class. Initializing the object

Date Class Method Calls

```
int Date::JumpYear (int j) const
{
     return  year+j;
}
```

Computation with private members without changing them

```
int graduation_year=Today.JumpYear(1);
```

`graduation_year` is 2019 after this statement

Date Class Method Calls

```
void Date::PrintDate()
{
cout<<"Month:"<<month<<"
Day:"<<day<<"
Year:"<<year<<"\n";
};
```

Controlled access to private members

```
Today.PrintDate();
```

Screen:
Month:10 Day:2 Year:2018

`cout<<Today.day;` Will not work, `day` is private

Use of Classes

- Classes are designed and implemented by designers for certain purposes
- The users (clients) reuse the classes in their own code without redesigning them
- Example:
  - Ahmet designs and implements `Date` Class
  - Mehmet uses `Date` Class in his Calendar software

Access Control: Private and Public Members

```
class Date
{
private:
            int month, day, year;
public:
           Date (int m=1, int d=1, int y=1900);
           void PrintDate( );
           int JumpYear(int j) const;
           void SetDate(int m, int d, int y);

      };
```


C++ Classes: Controlling Access to Members: Private Members

- Data and internal operations necessary to implement the class
- The most restrictive access level
- Private data members and operations can be accessed only by the methods in the class.
- Use this access to declare members that should only be used by the class.
- Example:
  - Variables: that contain information that if accessed by an outsider could put the object in an inconsistent state
  - Methods: if invoked by an outsider, could jeopardize the state of the object or the program in which it's running.

C++ Classes: Controlling Access to Members: Public Members

- Operations available to clients (who do not need to know anything about the private parts)
- Clients can only access the public part
- Interface of the object to the program.
- Any statement in a program block that declares an object can access a public member of the object
- The public parts hide information encapsulated in the private parts to:
  - Protect data integrity
  - Enhance portability
  - Facilitate software reuse

Example for Controlled Access

```
class Date
{
private:
            int month, day, year;
public:
           Date (int m=1, int d=1, int y=1900);
            void PrintDate( );
            int JumpYear(int j) const;
            void SetDate(int m, int d, int y);
       };
```

```
void Date::PrintDate()
{
cout<<"Month:"<<month<<"
Day:"<<day<<"
Year:"<<year<<"\n";
};
```

```
#include "Date.h"
int main( )
{ Date Today(10,7,2016);
Today.PrintDate();
cout<<Today.day<<"\n";
\\ERROR!!
}
```

Why do we need access control

- Large programs involving more than one programmer.
- A class can be very complex, with many member functions and data members.
- One programmer creates a class
  - Knows all details
- Other programmers use the class in their parts
  - Only need to know how to use it
  - Only know the public functions

Practice

- Make data members private.
- Member functions which must be called from outside the class should be public.
- Member functions which are only called from within the class (also known as "helper functions") should probably be private.

C++ Classes

- Class declaration
  - Member variables
  - Member function prototypes
- Class implementation
  - Member function definitions

Example: Date Class Declaration

```
class Date
{
private:
        int month, day, year;

public:
        Date (int m=1, int d=1, int y=1900);
         void PrintDate( );
         int JumpYear(int j) const;
         void SetDate(int m, int d, int y);

    };
```

Prototypes for member functions

Function Prototype

```
int JumpYear (int j) const;
```

Result type | Function name | Parameter type | Formal parameter: place holder to stand for the actual parameter

- Describes how the function is called
- Tells everything you need to know to make a function call
- Terminates with semi-colon
- Lets the compiler know that we intend to call this function.
- Lets the compiler generate the correct code for calling the function
- Enables the compiler to check up on our code (by making sure, for example, that we pass the correct number of arguments to each function we call).

Function Definition and Scope

```
int Date::JumpYear (int j) const
{
      return  year+j ;
}
```

- `<ReturnValueType><ClassName>::FunctionName(parameters)`
- Function returns data of type `int`
- Declaring a member function with the `const` keyword specifies that the function is a "read-only" function that does not modify the object for which it is called.
- :: **scope resolution operator:** shows that the function `JumpYear` is in the scope of `Date` class → `JumpYear` belongs to `Date` class → `JumpYear` can access private members (accesses year)
- **scope:** The range of reference for an object or variable.

Alternative Constructors

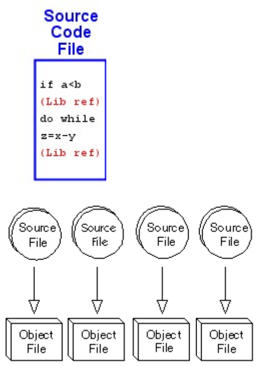
```
# include  <string.h>
class Date
{
private:
            int month, day, year;
public:
            Date (int m=1, int d=1, int y=1900);
            Date (char *dstr);
// Other methods

};
```

- Two different constructors are defined
- The compiler will select the appropriate constructor according to the call parameters during object creation
- Constructor cannot be private
- Why?

Compiling

- Source code: human-readable text file format for the computer program
- The compiler program reads the text source code file as input and generates a binary file called an "object" file.
- Object file: a binary (machine-readable) version of the programmer's soure code file, complete with those references to library routines.

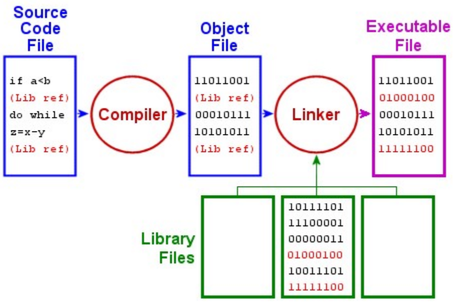


Compile each source file

Linking

- The linker is another program
  - Reads in the object file that was generated by the compiler and one or more library files.
  - Every time the linker finds a reference to a library routine in the object file, it reads the library files and finds that routine
  - Library files:

  #include "stdio.h"



  - It then replaces the programmer's reference with the code for the routine from the library file.
  - Generates the executable binary file.

Inline Definition

```
# include  <string.h>
class Date
{
private:
            int month, day, year;
public:
            Date (int m=1, int d=1, int y=1900);
            Date (char *dstr);
            void PrintDate(void);
void SetDate(int m, int d, int y)
{//only does basic check
if(m>=1&&m<=12)
month=m;
if(d>=1&&d<=31)
day=d;
if(y>=0)
year=y;
}

};
```

INLINE DEFINITION:
Compiler inserts the complete body of the function wherever it is called instead of a jump instruction to the function definition
Faster BUT makes the code larger

Inheritance in OOP

- Example: People database program
- Classes such as `Parent`, `Student`, `Worker`
- All these data types have overlapping features because they all describe some `Person` with more specific properties
- Idea: Define `Person` first and then extend it to make it more specific

Inheritance: Example

```
enum Gender{male, female};

class Person
{

protected://new access control level used for inheritance
Gender gender;
int age;
public:
   void Info();
   Person (int a=0, Gender g=male);

};

Person::Person (int a, Gender g):age(a), gender(g)
{}
//same as:
//Person::Person (int a, Gender g)
//{age=a;
//gender= g;}

void Person::Info()
{
   cout<<"Gender:"<<gender<<" Age:"<<age<<"\n";
}
```

Inheritance: Example

```
class Parent:public Person//Derived Class
{

private:
int children;
public:
Parent (int c=0);
void Info();
void update();
};

Parent::Parent (int c): children(c)//ADD-ON
{}
void Parent::Info()//OVERWRITE
{
   cout<<"Gender:"<<gender<<" Age:"<<age<<" Number of
   Children:"<<children<<"\n";
}
void Parent::update( )//BRAND NEW
{
   cout<<"age:";
   cin>>age;
   cout<<"children:";
   cin>>children;
}
```


Inheritance: Sample run

```
void main()
{
Parent p;
Person q;
cout<<"parent info:";
p.Info();
cout<<"person info:";
q.Info();
cout<<"change:\n";
p.update();
p.Info();}
```

parent info:Gender:0 Age:0 Number of Children:0

person info:Gender:0 Age:0

change:

age:45
children:3
updated parent info:Gender:0 Age:45 Number of Children:3

Inheritance and access control

```
class B { ... };
class D1 : public B {
... };
class D2 : private B {
... };
```

- B is a public base class of D1.
  - private members of B cannot be accessed by the derived class
  - public members of B are also public in D1
  - protected members of B are also protected in D1
- B is a private base class of D2.
  - private members of B cannot be accessed by the derived class
  - public and protected members of B are private in D2

```
class Person
{

protected:
Gender gender;
int age;
public:
     void Info();
     Person (int a=0, Gender g=male);

};

class Parent:public Person//Derived Class
{

private:
int children;
public:
Parent (int c=0);
void Info();
void update();
};


void Parent::update( )//BRAND NEW
{
     cout<<"age:";
     cin>>age;
     cout<<"children:";
     cin>>children;


}
```


Inheritance and access control

```
class Person
{
private:
Gender gender;
int age;
public:
  void Info();
  Person (int a=0,
  Gender g=male);

};
```

- Will not compile because the private members of the base class are not accessible
- Errors such as:
  - 'gender' : cannot access private member declared in class 'Person'
  - 'age' : cannot access private member declared in class 'Person'

Creating objects

- When a derived class object is created:
  - Base class constructor is first called and initializes the members from the base class
  - Derived constructor is called next to initialize the new members of the derived class or overwrite the base initialization as required

Abstract classes and Polymorphism

- An abstract class:
  - Only specifies an interface.
  - typically has one or more pure virtual member functions .
- A pure virtual member function declares an interface only:
  - specifies the set of operations
  - there is no implementation defined
- It is not possible to create object instances of abstract classes.

Abstract classes and Polymorphism

- Abstract class is a base class from which other classes are derived
- Declaring the member functions virtual makes it possible to access the implementations provided by the derived classes through the base-class interface.
- We don't need to know
  - how a particular object instance is implemented,
  - of which derived class a particular object is an instance.
- This design pattern uses the idea of **polymorphism**.

Example

```
class Polygon
{
protected:
int width, height;
public:
Polygon(int w=0,
  int h=0);
void
  set_values(int
  w, int h);
};
```

```
class Rectangle:public
   Polygon
{
public:
//inherits the constructor
int Area()
{return width*height;}
};
class Triangle:public
   Polygon
{
public:
//inherits the constructor
int Area()
{return width*height/2;}
};
```

Example

- The computation of the area (Area implementation) will be different among polygons
- BUT
- Any class derived from Polygon would have some Area method.

Virtual functions

```
class Polygon
{
protected:
int width, height;
public:
Polygon(int w=0, int
  h=0);
void set_values(int
  w, int h);
virtual int Area()
{return (0);}
};
```

- Virtual function `Area` is not implemented for the base class
- If not redefined in the derived class, it returns 0
- Provides interface and ensures the existence of `Area` function for the derived classes

Example

```
class Polygon
{
protected:
int width, height;
public:
Polygon(int w=0, int
  h=0);
void set_values(int
  w, int h);
virtual int Area()
{return (0);}
};
```

```
class Rectangle:public
   Polygon
{
public:
//inherits the constructor
int Area()
{return width*height;}
};
class Triangle:public
   Polygon
{
public:
//inherits the constructor
int Area()
{return width*height/2;}
};
```


EE 441 Data Structures

Lecture 1

Introduction