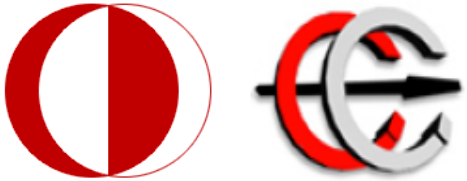


EE 441 Data Structures

Chapter 1: Object-Oriented Programming, Classes

Serkan SARITAŞ
Uğur HALICI
Klaus Werner SCHMIDT

Abstract Data Type (ADT)

- Model used to understand the design of a data structure
- Implementation independent data description

→ Design data with pencil and paper before programming

- ADT specifies
  - Contents
  - Type of data stored
  - Legal operations on the data
- Benefit: 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
    - Modification of values
    - Computations
    - etc.

Designing an ADT

- Example: A calendar software
- Questions:
  - What kind of data organization do we need?
  - What kind of procedures do we need to manipulate this data?
- For a calendar software, 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: day, month, 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 day month 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

Object Oriented Programming: Classes and Objects

- A class
  - Is an actual representation of an ADT
  - Provides implementation details for the data structure used
  - Provides implementation details for the operations
  - Has members
    - Variables to store data
    - Operations (methods) for data handling

Class Example

- Class example in C++ syntax

```
class Date {
        private:
                int day; // Data representation of day
                int month; // Data representation of month
                int year; // Data representation of year
        public:
                // Constructor
                Date (int d=1, int m=1, int y=1900);
                // Method to print the current date
                void PrintDate(void);
                // Method to modify the year by j
                int JumpYear(int j) const;
                // Method to directly set the date
                void SetDate(int d, int m, int y);
};
```

Objects

- An object
  - is a self-contained entity that consists of data
  - has methods to manipulate the object's data as defined by the object's class
  - can be uniquely identified by its name
  - defines a state which is represented by the values of its data at a particular time
  - is also denoted as an instance of a class

    → A class is a blueprint, or prototype that defines properties and behavior of sets of objects

Object Example

- class Date is declared

```
class Date {
        private:
                int day; // Data representation of day
                int month; // Data representation of month
                int year; // Data representation of year
        public:
                // Constructor
                Date (int d=1, int m=1, int y=1900);
                // Method to print the current date
                void PrintDate(void);
                // Method to modify the year by j
                int JumpYear(int j) const;
                // Method to directly set the date
                void SetDate(int d, int m, int y);
};
```

- Objects (instances) of class Date

```
Date Today(3,10,2022); // Object that holds today's date
Date Tomorrow(4,10,2022); // Object that holds tomorrow's date
```

C++ Classes

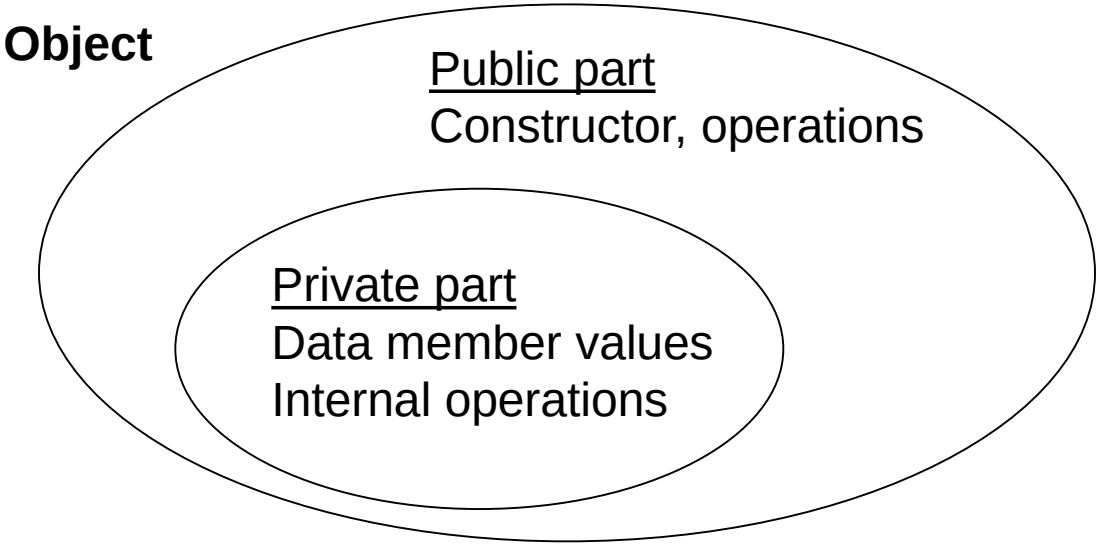
- Class declaration
  - Member variables
  - Member function prototypes
- Class implementation
  - Member function definitions

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

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

C++ Classes

- Members are variables and methods for data handling
- Classes can protect members from access by other objects
- Public and private sections in a class declaration allow program statements outside the class different access to the class members



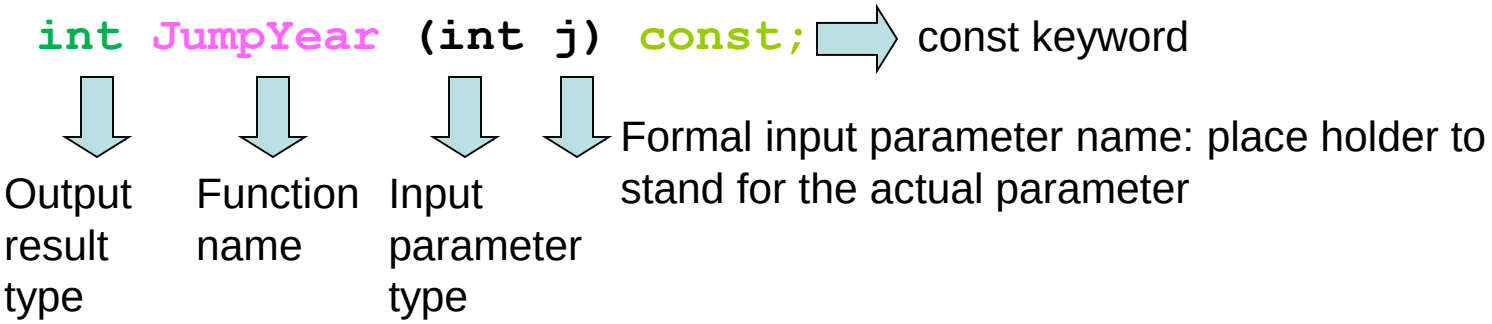
C++ Classes

- Members are variables and methods for data handling
- Classes can protect members from access by other objects
- Public and private sections in a class declaration allow program statements outside the class different access to the class members

```
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 d=1, int m=1, int y=1900);
      void PrintDate( );
      int JumpYear(int j) const;
      void SetDate(int d, int m, int y);
};
```

Function Prototype



- Prototype 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
  (for example, it makes sure that we pass the correct number of arguments to each function we call).

Function Definition and Scope

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


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

Constructor:

- Creates an object (instance of the class)
- Initializes the object

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

Member method “JumpYear”:
Computation with private member “year”
Keyword “const” requires that no member is changed by the operation

Date Class Implementation

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

<ReturnValueType><ClassName>::FunctionName(parameters)

- Example 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

Date Class Implementation

```
void Date::PrintDate() const{
    std::cout << "Day: " << day << " Month: "
    << month << "Year: " << year << std::endl;
};
```

Member method “PrintDate”:
Controlled access to private members

No member changes → “const”

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

Member method “SetDate”:
Modify member variables

Keyword “const” cannot be used

Date Class Method Calls

- Example main program

```
int main()
{
    std::cout << "Let's use the Date class" << std::endl;
    Date Today(3,10,2022); // Construct object initialized with today's date
    Today.PrintDate(); // Print today's date using member method
    Date Tomorrow; // Construct default object
    Tomorrow.SetDate(4,10,2022); // Set the date of Tomorrow
    Tomorrow.PrintDate(); // print tomorrow's date
    std::cout << "Graduation Year: " << Today.JumpYear(1) << std::endl;
    return 0;
}
```

Date Class Method Calls

- Console output in CodeBlocks 20.03

```
Let's use the Date class
Day: 3 Month: 10 Year: 2022
Day: 4 Month: 10 Year: 2022
Graduation Year: 2023

Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

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

- Example

```
class Date {
    private:
        int day; // Data representation of day
        int month; // Data representation of month
        int year; // Data representation of year
    public:
        // Constructor
        Date (int d=1, int m=1, int y=1900);
        // Method to print the current date
        void PrintDate(void);
        // Method to modify the year by j
        int JumpYear(int j) const;
        // Method to directly set the date
        void SetDate(int d, int m, int y);
};
```

private

public

C++ Classes: Private Members

- Most restrictive access level
- Data and internal methods needed to implement the class
- Private data members and methods can be accessed only by the methods of 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 violate security or put the object in an inconsistent state
  - Methods that, if invoked by an outsider, could jeopardize the state of the object or the program in which it's running

C++ Classes: 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 d=1, int m=1, int y=1900);
        void PrintDate( );
        int JumpYear(int j) const;
        void SetDate(int d, int m, int y);
};
```

```
void Date::PrintDate() const{
    std::cout << "Day: " << day << " Month: "
    << month << "Year: " << year << std::endl;
};
```

```
In function 'int main()':
error: 'int Date::day' is private within this context
note: declared private here
```

```
int main()
{
    Date Today(3,10,2022); // Today's date
    Today.PrintDate(); // print today's date
    std::cout << "Day: " << Today.day << std::endl;
    return 0;
}
```

Compiler error since „day“ is a private member variable

Why do we need access control

- Large programs involving more than one programmer
- A class can be very complex
  - Many member methods
  - Many data members
- One programmer creates a class

  → Knows all details
- Other programmers use the class in their code

  → Only need to know how to use it

  → Only know the public functions

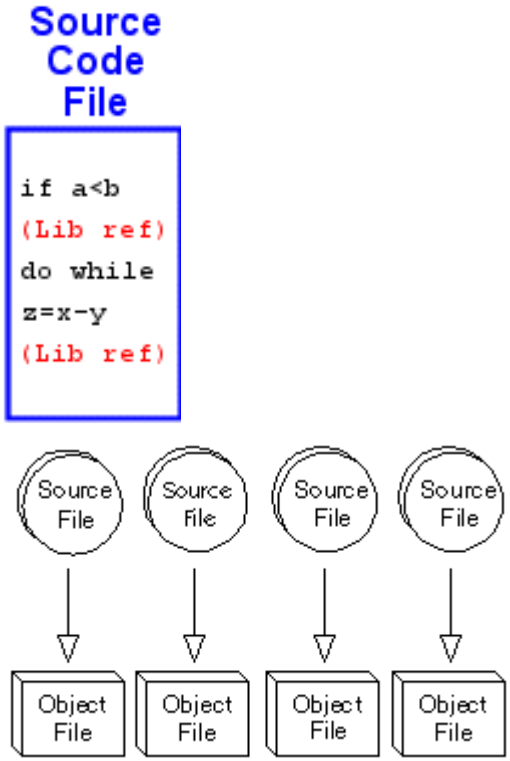
Alternative Constructors

```
# include  <string>
class Date{
        private:
                int day, month, year;
        public:
                Date (int d = 1, int m = 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
- Recall: Constructor cannot be private
- Why?

Compiling

- Source code: human-readable text file format for the computer program
- Compiler program: reads the text source code file as input and generates a binary file called an "object" file
- Object file: binary (machine-readable) version of the programmer's source code file including references to library routines



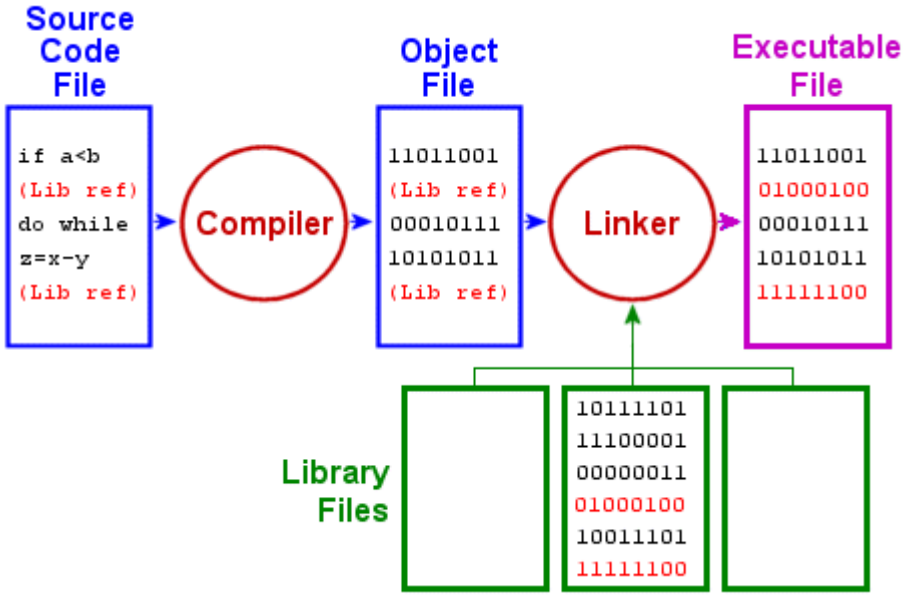
Compile each source file

Linking

- The linker is a 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 <iostream>

    #include <math.h>



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

Inline Definition

```
# include  <iostream>
# include  <string.h>
class Date {
   private:
      int day, month, year;
   public:
      Date (int d=1, int m=1, int y=1900);
      Date (char *dstr);
      void PrintDate(void);
      void SetDate(int d, int m, 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 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 Object Oriented Programming

- Example: People database program
- Classes such as “Parent”, “Student”, “Worker”
- Observation: All these data types have common features

  → They all describe a Person with more specific properties
- Idea
  - Define a general “Person” first
  - Extend it to make it more specific

Inheritance: Example

- Class Declaration (Base Class)

```
enum Gender{male, female};

class Person{
    protected: //new access control level used for inheritance
        Gender gender; // either male or female
        int age; // Age of the person
    public:
        Person (int a = 0, Gender g = male); // Constructor
        void Info() const; // Print info about person
};
```

Inheritance: Example

- Implementation (Base Class)

```
Person::Person (int a, Gender g): age(a), gender(g){
}
// more general than:
//Person::Person(int a, Gender g){
//   age=a;
//   gender= g;
//}
void Person::Info() const{
    std::cout << " Age: " << age;
        std::cout << ", Gender: ";
        if(gender == male)
         std::cout << "male";
    else
         std::cout << "female";
    std::cout << std::endl;
}
```

Member initializer list

Constructor body initialization

Inheritance: Example

- Declaration (Inherited Class)

```
class Parent:public Person { //Derived Class
   private:
      int children; // Additional member variable
   public:
      Parent(int a = 20, Gender g = female, int c = 0); //Constructor
      void Info() const; //Overwrite member method Info
      void update(); //New public member method
};
```

Derived from Person class

Inheritance: Example

- Implementation (Inherited Class)

```
Parent::Parent(int a, Gender g, int c):Person(a,g),children(c){
}
```

Member initializer: Constructor of Base class

Member initializer for Additional member variable

```
void Parent::Info() const{ //OVERWRITE
    Person::Info(); // Call function from Base class
    std::cout << ", Number of Children: " << children << std::endl;
}
```

Inheritance: Example

- Implementation (Inherited Class)

```
void Parent::update( ){ //BRAND NEW
        std::cout << "Age: ";
        std::cin >> age;
        int gender_input;
        std::cout << "Gender (male=0, female=1): ";
        std::cin >> gender_input;
        if(gender_input == 0)
            gender = male;
         else
            gender = female;
        std::cout << "Number of Children: ";
        std::cin >> children;
}
```

Inheritance: Example

- Main Function

```
#include "Person.hpp"

int main(){
    Parent p;
    Person q;
    std::cout << "parent info:";
    p.Info(); // Show default parent info
    std::cout << "person info:";
    q.Info(); // Show default person info
    std::cout << "change:" << std::endl;
    p.update(); // Update parent info
    p.Info(); // Show updated parent info
    return 0;
}
```

```
parent info:
Age: 20, Gender: female
Number of Children: 0
person info:
Age: 0, Gender: male
change:
Age: 33
Gender (male=0, female=1): 1
Number of Children: 2
Age: 33, Gender: female
Number of Children: 2
```

Construct with default member variable values

Inheritance and Access Control

- Case 1: Person is a public base class of Parent
  - Private members of Person cannot be accessed by Parent
  - Public members of Person are also public in Parent
  - Protected members of Person are also protected in Parent

```
class Person{
    private: //
        Gender gender;
        int age;
};
```

```
class Parent:public Person {
   private:
      int children;
   public:
      Parent(int a = 20, Gender
         g = female, int c = 0);
      void Info() const;
};
```

- Compiler error since Parent tries to access age and gender

```
error: 'int Person::age' is private within this context
```

Inheritance and Access Control

- Case 2: Person is a protected base class of Parent

```
class Parent:protected Person
```

  - Private members of Person cannot be accessed by Parent
  - Public members of Person are protected in Parent
  - Protected members of Person are also protected in Parent
- Case 3: Person is a private base class of Parent

```
class Parent:private Person
```

  - Private members of Person cannot be accessed by Parent
  - Public and protected members of Person are private in Parent

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
- Example: Call constructor of Base class

```
Parent::Parent(int a, Gender g, int c):children(c){}
```

```
parent info:
Age: 0, Gender: male, Number of Children: 0
```

Default values of Base class

```
Parent::Parent(int a, Gender g, int c):Person(a,g), children(c){}
```

```
parent info:
Age: 20, Gender: female, Number of Children: 0
```

Overwrite default values of Base class

Abstract Classes and Polymorphism

- 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 a member function 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

Polymorphism Example

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

```
class Rectangle:public Polygon{
    public:
        Rectangle(int w = 0, int
        h = 0):Polygon(w,h){}
    int Area() const{
        return width*height;
    }
};

class Triangle:public Polygon{
    public:
        Triangle(int w = 0, int
       h = 0):Polygon(w,h){}
    int Area() const{
        return width*height/2;
    }
};
```

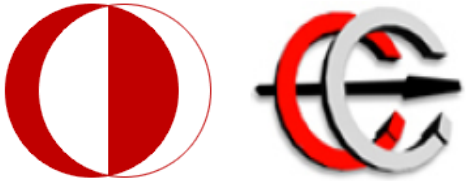
virtual member method with default implementation

Implementation in derived class

Polymorphism Example

```
int main(){
    Rectangle r(4,5);
    Triangle t(7,8);
    std::cout << "Rectangle Area: " << r.Area() << std::endl;
    std::cout << "Triangle Area: " << t.Area() << std::endl;
    // Make use of polymorphism
    Polygon *p_poly; // Pointer to object of Polygon class
    p_poly = &r; // Assign address of r to p_polygon
    std::cout << "Access member method Area from Base class" << std::endl;
    // Access member method Area from Base class
    std::cout << "Rectangle Area: " << p_poly->Area() << std::endl;
    p_poly = &t; // Assign address of t to p_polygon
    // Access member method Area from Base class
    std::cout << "Triangle Area: " << p_poly->Area() << std::endl;
    return 0;
}
```

```
Rectangle Area: 20
Triangle Area: 28
Access member method Area from Base class
Rectangle Area: 20
Triangle Area: 28
```



EE 441 Data Structures

Chapter 1: Object-Oriented Programming, Classes

Serkan SARITAŞ
Uğur HALICI
Klaus Werner SCHMIDT