# Classes (Part1)

Prof. Seokin Hong

SUNGKYUNKWAN UNIVERSITY

# Agenda

- Basics
- Private VS Public
- Constructors
- Abstract Data Types

# Basics

# What Is a Class?

- **A class is a data type whose variables are objects**
- **Some pre-defined data types you have used are**
  - int, char
- **A pre-defined class you have used is**
  - ifstream

- **A class definition includes**
  - A description of the member variables
  - A description of the member functions

# A Class Example

- **To create a new type named DayOfYear as a class**
  - Decide the values to represent
    - Example: **dates** such as July 4
      - Member variable ***month*** is an int (Jan = 1, Feb = 2, etc.)
      - Member variable ***day*** is an int
  - Decide the member functions needed

```
class DayOfYear
{
    public:
            int month;
            int day;
            void output( );
};
```

Member Function **Declaration**

# Defining a Member Function

- **Member functions are declared in the class definition**
- **Member function definitions need to identify the class in which the function is a member**
  - **Syntax:**
    Returned_Type **Class_Name::Function_Name**(Parameter_List)
    {
    Function Body Statements
    }

Class name

```
void DayOfYear::output()
  {
        cout << "month = " << month
                << ",  day = " << day
                << endl;
  }
```

# The '::' Operator

- **'::' is the scope resolution operator**
  - Tells the class a member function is a member of
  - void **DayOfYear::output( )** indicates that function output is a member of the DayOfYear class

```
void DayOfYear::output()
  {
        cout << "month = " << month
             <<  ",  day = " << day
             << endl;
   }
```

# ‘::’ and ‘.’

- **‘::’ is used with classes to identify a member**

```
void DayOfYear::output( )
{
    // function body
}
```

- **‘.’ is used with variables (Object) to identify a member**

```
DayOfYear birthday;
birthday.output( );
```

# Example

```
//Program to demonstrate a very simple example of a class.
//A better version of the class DayOfYear will be given in Display 10.4.
#include <iostream>

class DayOfYear
{
public:
    void output( );
    int month;
    int day;
};

int main( )
{   using namespace std;
    DayOfYear today, birthday;

    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;

    cout << "Today's date is ";
    today.output( );
    cout << "Your birthday is ";
    birthday.output( );

    if (today.month == birthday.month
        && today.day == birthday.day)
        cout << "Happy Birthday!\n";
    else
        cout << "Happy Unbirthday!\n";

    return 0;
}
//Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

Member function declaration

Calls to the member function **output**

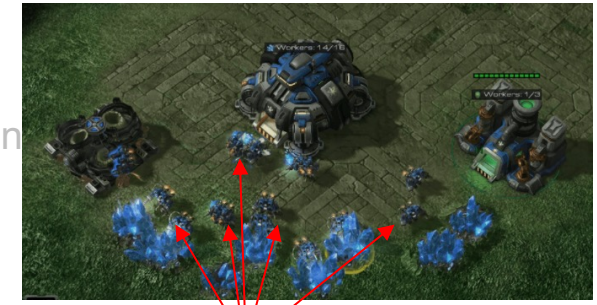
Member function definition

# Encapsulation

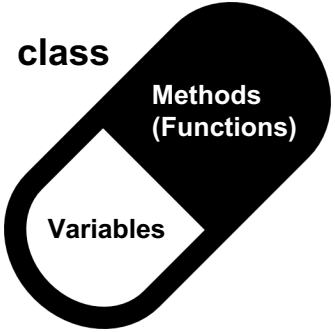
- **Encapsulation**
  - You can organize related variables and functions into one bundle (called class)

- **Inheritance**
  - You can create a class that adds other information to an already existed class

- **Polymorphism**
  - You can give the same name to fun



**Objects**
**(Data, Operations)**



# Problems With DayOfYear

- **Changing how the month is stored in the class DayOfYear requires changes to the program!**

- **If we decide to store the month as three characters (JAN, FEB, etc.) instead of an int**
  - cin >> today.month will no longer work
  - if(today.month == birthday.month) will no longer work

- **An ideal class definition of DayOfYear could be changed without requiring changes to the program that uses DayOfYear!**

# Fixing DayOfYear

- **To fix DayOfYear**
  - We need to add member functions to use when changing or accessing the member variables
  - If the program never directly references the member variables, changing how the variables are stored will not require changing the program
  - We need to be sure that the program does not ever directly reference the member variables

# Public Or Private?

- **C++ helps us restrict the program from directly referencing member variables**
  - **Private** members of a class can only be referenced within the definitions of member functions
    - If the program tries to access a private member, the compiler gives an error message
  - Private members can be variables or functions

# Private Variables

- **Private variables cannot be accessed directly by the program**
  - Changing their values requires the use of public member functions of the class
  - Example

```
void DayOfYear::set(int new_month, int new_day)
 {
        month = new_month;
        day = new_day;
 }
```

# Public or Private Members

- **The keyword private identifies the members of a class that can be accessed only by member functions of the class**
  - Members that follow the keyword private are private members of the class

- **The keyword public identifies the members of a class that can be accessed from outside the class**
  - Members that follow the keyword public are public members of the class

# A New DayOfYear

```
//Program to demonstrate the class DayOfYear.
#include <iostream>

class DayOfYear
{
public:
    void input( );
    void output( );

    void set(int newMonth, int newDay);
    //Precondition: newMonth and newDay form a possible date.
    //Postcondition: The date is reset according to the arguments.

    int getMonth( );
    //Returns the month, 1 for January, 2 for February, etc.

    int getDay( );
    //Returns the day of the month.
private:
    void checkDate( );
    int month;
    int day;
};
int main( )
{   using namespace std;
    DayOfYear today, bachBirthday;
    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );

    bachBirthday.set(3, 21);
    cout << "J. S. Bach's birthday is ";
    bachBirthday.output( );

    if (today.getMonth( ) == bachBirthday.getMonth( ) &&
        today.getDay( ) == bachBirthday.getDay( ) )
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
//Uses iostream:
void DayOfYear::input( )
{   using namespace std;
    cout << "Enter the month as a number: ";
```

*Private member function*

*Private member variables*

```
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
    checkDate( );
}

void DayOfYear::output( )
```

*Private members may be used in member function definitions (but not elsewhere).*

//The definition of output() is the same as in the previous example.

```
void DayOfYear::set(int newMonth, int newDay)
{
    month = newMonth;
    day =  newDay;
    checkDate();
}

void DayOfYear::checkDate( )
{   using namespace std;
    if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);
    }
}

int DayOfYear::getMonth( )
{
    return month;
}

int DayOfYear::getDay( )
{
    return day;
}
```

# Using Private Variables

- **It is normal to make all member variables private**
- **Private variables require member functions to perform all changing and retrieving of values**
  - **Accessor (getter) functions** allow you to obtain the values of member variables
    - Example: getDay() in class DayOfYear
  - **Mutator (setter) functions** allow you to change the values of member variables
    - Example: set() in class DayOfYear

# General Class Definitions

- **The syntax for a class definition is**

```
class Class_Name
{
        public:
                Member_Specification_1
                Member_Specification_2
                …
                Member_Specification_3
          private:
                Member_Specification_n+1
                Member_Specification_n+2
                …
};
```

# Declaring an Object

- **Once a class is defined, an object of the class is declared just as variables of any other type**

Example: To create two objects of type Bicycle:

```
class Bicycle
{
    // class definition lines
};

Bicycle myBike,  yourBike;
```

# The Assignment Operator

- **Objects and structures can be assigned values with the assignment operator (=)**
  - Example:

```
DayOfYear  dueDate, tomorrow;

tomorrow.set(11, 19);

dueDate = tomorrow;
```

  - If the class contains pointers, a deep copy is required to copy what the pointers refer to
    → Will be discussed later

# Program Example

- **BankAccount Class**
  - Withdrawal of money at any time
  - Storing an account balance
  - Storing the account's interest rate

```
//Program to demonstrate the class BankAccount.
#include <iostream>
using namespace std;

//Class for a bank account:
class BankAccount
{
public:
    void set(int dollars, int cents, double rate);
    //Postcondition: The account balance has been set to $dollars.cents;
    //The interest rate has been set to rate percent.

    void set(int dollars, double rate);
    //Postcondition: The account balance has been set to $dollars.00.
    //The interest rate has been set to rate percent.

    void update( );
    //Postcondition: One year of simple interest has been
    //added to the account balance.

    double getBalance( );
    //Returns the current account balance.

    double getRate( );
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate have
    //been written to the stream outs.
private:
    double balance;
    double interestRate;

    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3)
    //returns 0.503.
};
int main( )
{
    BankAccount account1, account2;
    cout << "Start of Test:\n";
    account1.set(123, 99, 3.0);
    cout << "account1 initial statement:\n";
    account1.output(cout);
    account1.set(100, 5.0);
    cout << "account1 with new setup:\n";
    account1.output(cout);
```

The member function set is overloaded.

Calls to the overloaded member function set

# Program Example

- **BankAccount Class**
  - Withdrawal of money at any time
  - Storing an account balance
  - Storing the account's interest rate

```
    account1.update( );
    cout << "account1 after update:\n";
    account1.output(cout);

    account2 = account1;
    cout << "account2:\n";
    account2.output(cout);
    return 0;
}

void BankAccount::set(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        return;
    }

    balance = dollars + 0.01*cents;
    interestRate = rate;
}

void BankAccount::set(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        return;
    }

    balance = dollars;
    interestRate = rate;
}

void BankAccount::update( )
{
    balance = balance + fraction(interestRate)*balance;
}

double BankAccount::fraction(double percentValue)
{
    return (percentValue / 100.0);
}

double BankAccount::getBalance( )
{
    return balance;
}
```

*Definitions of overloaded member function* set

*In the definition of a member function, you call another member function like this.*

# Constructors

# Constructors

- **A constructor can be used to initialize member variables when an object is declared**
  - A constructor is a member function that is usually public
  - A constructor is automatically called when an object of the class is declared
  - A constructor's name must be the name of the class
  - A constructor cannot return a value

# Constructor Declaration

- **A constructor for the BankAccount class could be declared as:**

```
class BankAccount
{
    public:
            BankAccount(int dollars, int cents, double rate);
             //initializes the balance to $dollars.cents
             //initializes the interest rate to rate percent


};
```

# Constructor Definition

- **The constructor for the BankAccount class could be defined as**

```
BankAccount::BankAccount(int dollars, int cents, double rate)
 {
   if ((dollars < 0) || (cents < 0) || ( rate < 0 ))
    {
        cout << “Illegal values for money or rate\n”;
        exit(1);
    }
  balance = dollars + 0.01 * cents;
  interestRate = rate;
 }
```

Note that the class name and function name are the same

# Calling A Constructor

- **A constructor is not called like a normal member function:**

```
BankAccount  account1;

account1.BankAccount(10, 50, 2.0);
```

- **A constructor is called in the object declaration**

```
BankAccount account1(10, 50, 2.0);
```

  - Creates a BankAccount object and calls the constructor to initialize the member variables

# Overloading Constructors

- **Constructors can be overloaded by defining constructors with different parameter lists**
  - Other possible constructors for the BankAccount class might be

```
BankAccount (double balance, double interestRate);
BankAccount (double balance);
BankAccount ( );
```

# The Default Constructor

- **A default constructor uses no parameters**

**Declaration example**

```
class BankAccount
{
        public:
                BankAccount( );
                // initializes balance  to $0.00
                // initializes rate to 0.0%

};
```

**Definition example**

```
 BankAccount::BankAccount( )
{
                balance = 0;
                rate = 0.0;
}
```

# The Default Constructor (Cont.)

- **The default constructor is called during declaration of an object**
  - An argument list is not used

```
BankAccount  account1;
// uses the default BankAccount constructor
```

# Example

```cpp
//Program to demonstrate the class BankAccount.
#include <iostream>
using namespace std;

//Class for a bank account:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.

    BankAccount( );
    //Initializes the account balance to $0.00
    //and the interest rate to 0.0%.

    void set(int dollars, int cents, double rate);
    //Postcondition: The account balance has been set to $dollars.cents;
    //The interest rate has been set to rate percent.

    void set(int dollars, double rate);
    //Postcondition: The account balance has been set to $dollars.00.
    //The interest rate has been set to rate percent.

    void update( );

    //Postcondition: One year of simple interest has been added
    //to the account balance.

    double getBalance( );
    //Returns the current account balance.

    double getRate( );
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate
    //have been written to the stream outs.
private:
    double balance;
    double interestRate;

    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3)
    //returns 0.503.
};
```

```cpp
int main( )
{
    BankAccount account1(100, 2.3), account2;

    cout << "account1 initialized as follows:\n";
    account1.output(cout);
    cout << "account2 initialized as follows:\n";
    account2.output(cout);

    account1 = BankAccount(999, 99, 5.5);
    cout << "account1 reset to the following:\n";
    account1.output(cout);
    return 0;
}
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        return;
    }
    balance = dollars + 0.01 * cents;
    interestRate = rate;
}
BankAccount::BankAccount(int dollars, double rate)
{

    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        return;
    }
    balance = dollars;
    interestRate = rate;
}
BankAccount::BankAccount( ) : balance(0), interestRate(0.0)
{
    //Body intentionally empty
}
```

*This declaration causes a call to the default constructor. Notice that there are no parentheses.*

*An explicit call to the constructor* BankAccount::BankAccount

# Initialization Sections

- **An initialization section in a function definition provides an alternative way to initialize member variables**

```
BankAccount::BankAccount( ): balance(0), interestRate(0.0)
{
   // No code needed in this example
}
```

  - The values in parenthesis are the initial values for the member variables listed

# Parameters and Initialization

- **An initialization section in a function definition provides an alternative way to initialize member variables**

```
BankAccount::BankAccount(int dollars, int cents, double rate)
                                : balance (dollars + 0.01 * cents), interestRate(rate)
{
        if (( dollars < 0) || (cents < 0) || (rate < 0))
       {
           cout << “Illegal values for money or rate\n”;
           exit(1);
        }
}
```

# Member Initializers

- **C++11 supports a feature called member initialization**
  - Simply set member variables in the class
  - Ex:

```
class Coordinate
{
        private:
                int x=1;
                int y=2;
                ...
};
```

Creating a Coordinate object will initialize its x variable to 1 and y to 2

# Constructor Delegation

- **C++11 also supports constructor delegation.**
  - This lets you have a constructor invoke another constructor in the initialization section

- **For example, make the default constructor call a second constructor that sets X to 99 and Y to 99:**

```
Coordinate::Coordinate() : Coordinate(99,99)
{ }
```

# Abstract Data Types

# Abstract Data Type

- **ADT** is a kind of user-defined data type composed of
  - A collection of values
  - A set of operations (function) on the values
- **ADT** specifies what the operation do, not how to implement them
- So, **ADT** hides the internal structure and design of data types from the user
- **ADT** specification consists of
  - Names of every function, the type of its arguments, type of its result

- **A data type is an Abstract Data Type (ADT) if programmers using the type do not have access to the details of how the values and operations are implemented**

# Classes To Produce ADTs

- **To define a class so it is an ADT**
  - Separate the specification of how the type is used by a programmer from the details of how the type is implemented
  - Make all member variables **private** members
  - Basic operations a programmer needs should be public member functions
  - Fully specify how to use each public function
  - Helper functions should be private members

# ADT Interface

- **The ADT interface tells how to use the ADT in a program**
  - The **interface** consists of
    - The public member functions
    - The comments that explain how to use the functions
  - The interface should be all that is needed to know how to use the ADT in a program

# ADT Implementation

- **The ADT implementation tells how the interface is realized in C++**
  - The implementation consists of
    - The private members of the class
    - The definitions of public and private member functions
  - The implementation is not needed to write the main part of a program or any non-member functions

# ADT Benefits

- **Changing an ADT implementation does require changing a program that uses the ADT!**

- **Make it easier to divide work among different programmers**
  - One or more can write the ADT
  - One or more can write code that uses the ADT

# Example: The BankAccount ADT

- **In this version of the BankAccount ADT**
  - Data is stored as three member variables
    - The dollars part of the account balance
    - The cents part of the account balance
    - The interest rate
  - The public portion of the class definition remains unchanged from the previous version

```cpp
//Demonstrates an alternative implementation of the class BankAccount.
#include <iostream>
#include <cmath>
using namespace std;

//Class for a bank account:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.

    BankAccount( );
    //Initializes the account balance to $0.00 and the
    //interest rate to 0.0%.

    void set(int dollars, int cents, double rate);
    //Postcondition: The account balance has been set to $dollars.cents;
    //The interest rate has been set to rate percent.

    void set(int dollars, double rate);
    //Postcondition: The account balance has been set to $dollars.00.
    //The interest rate has been set to rate percent.

    void update( );
    //Postcondition: One year of simple interest has been
    //added to the account balance.

    double getBalance( );
    //Returns the current account balance.

    double getRate( );
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate
    //have been written to the stream outs.
private:
    int dollarsPart;
    int centsPart;
    double interestRate;
    //Expressed as a fraction, for example, 0.057 for 5.7%
    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3)
    //returns 0.503.

    double percent(double fractionValue);
    //Converts a fraction to a percentage. For example, percent(0.503)
    //returns 50.3.
};

int main( )
{
    BankAccount account1(100, 2.3), account2;

    cout << "account1 initialized as follows:\n";
    account1.output(cout);
    cout << "account2 initialized as follows:\n";
    account2.output(cout);

    account1 = BankAccount(999, 99, 5.5);
    cout << "account1 reset to the following:\n";
    account1.output(cout);
    return 0;
}
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    dollarsPart = dollars;
    centsPart = cents;
    interestRate = fraction(rate);
}
BankAccount::BankAccount(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    dollarsPart = dollars;
    centsPart = 0;
    interestRate = fraction(rate);
}
```

*In the old implementation of this ADT, the private member function* `fraction` *was used in the definition of* `update`*. In this implementation,* `fraction` *is instead used in the definition of constructors and in the set function.*

```cpp
BankAccount::BankAccount( ) : dollarsPart(0), centsPart(0), interestRate(0.0)

{
    //Body intentionally empty.
}
double BankAccount::fraction(double percentValue)
{
    return (percentValue/100.0);
}
//Uses cmath:
void BankAccount::update( )
{
    double balance = getBalance( );
    balance = balance + interestRate * balance;
    dollarsPart = staticCast<int>(floor(balance));
    centsPart = staticCast<int>(floor((balance - dollarsPart)*100));
}
double BankAccount::getBalance( )
{
    return (dollarsPart + 0.01 * centsPart);
}
double BankAccount::percent(double fractionValue)
{
    return (fractionValue * 100);
}
double BankAccount::getRate( )
{
    return percent(interestRate);
}
  //Uses iostream:
  void BankAccount::output(ostream& outs)
{
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << getBalance( ) << endl;
    outs << "Interest rate "<< getRate( ) << "%" << endl;
}
```

```cpp
void BankAccount::set(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        return;
    }
    dollarsPart = dollars;
    centsPart = cents;
    interestRate = fraction(rate);
}
void BankAccount::set(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        return;
    }
    dollarsPart = dollars;
    interestRate = fraction(rate);
}
```

# Interface Preservation

- **To preserve the interface of an ADT so that programs using it do not need to be changed**
  - Public member declarations cannot be changed
  - Public member definitions can be changed
  - Private member functions can be added, deleted, or changed

→ It is necessary to carefully determine the public members and their declaration

# Copyright Notice

- The contents of this slide deck are taken from the textbook (Problem Solving with C++, Walter Savitch).
- See your textbook for more details.
- Redistribution of this slide deck is not permitted.

# NEXT ?

## Classes (Part2)

Friends, Overloaded Operators