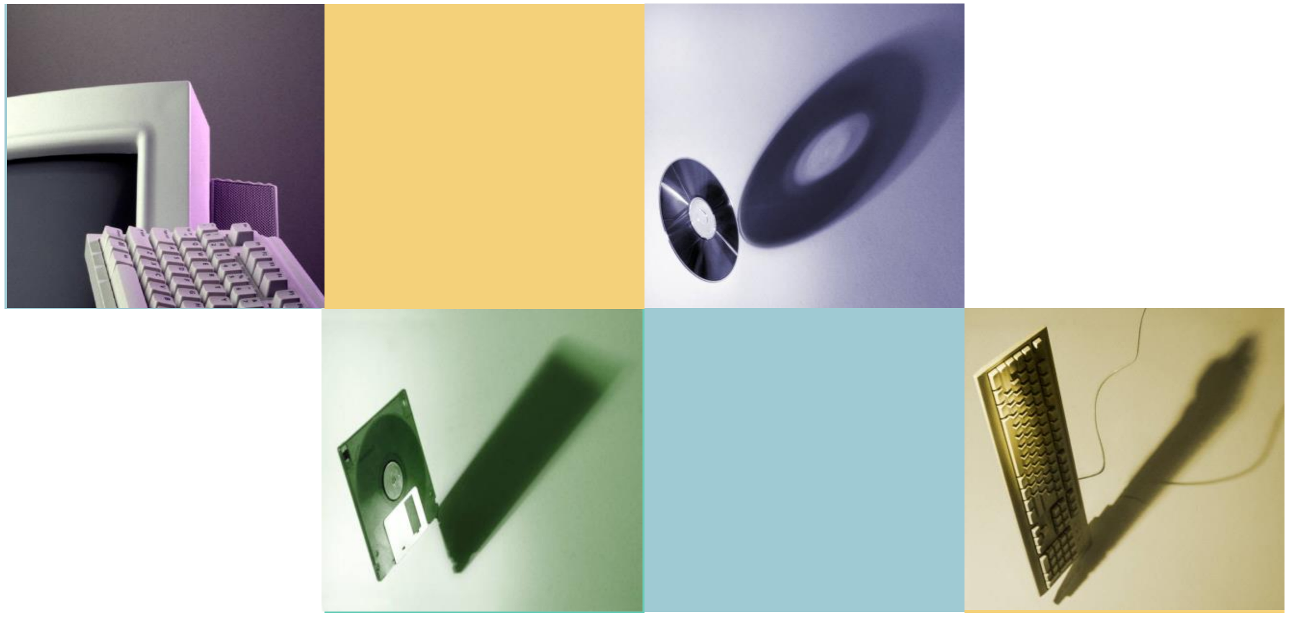
Object-Oriented Programming

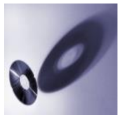


Chuan-Kang Ting

Dept of Computer Science and Information Engineering
National Chung Cheng University

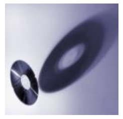
Chapter 7

Constructors and Other Tools



Outline

- **Constructors**
- **More tools**
  - The `const` parameter modifier
  - Inline functions
  - Static members
- **Vectors**

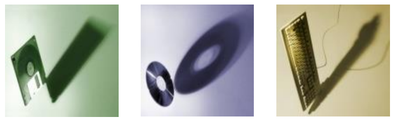


Constructors (1)

- **(Class) constructors**
  - Is a member function of a class that has **the same name** as the class
  - Is automatically called when an object of the class is *declared*
  - Is used to **initialize** objects, that is, to initialize the values of some or all member variables or other initialization jobs
    - Initialization → constructor
    - Assignment → assignment operator

Constructors (2)

- **Constructor definitions**
  - Same as definition of any member function
  - Except
    - Have the same name as the class
    - CanNOT return a value
      - No type, not even `void`, can be given at the start of function declaration or function header



Constructors (3)

- **An example**

```
class DayOfYear
{
  public:
    DayOfYear(int monthValue, int dayValue);
    //Constructor initializes month and day
    void input();
    void output();
    void set(int newMonth, int newDay);
    void set(int newMonth);
    int getMonthNumber();
    int getDay();
 private:
    int month;
    int day;
}
```

the same name

public section

no return-type

Constructors (4)

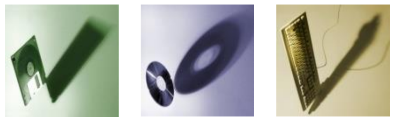
- **Calling constructors**
  - When declaring objects of the class
  - e.g.

```
DayOfYear date1(7,4), date2(5,5);
```

    - The constructor `DayOfYear` is called with the two arguments 7 (monthValue) and 4 (dayValue)
    - *Conceptually* equivalent to

```
DayOfYear date1, date2;
~~date1.DayOfYear(7,4);~~       //very illegal
~~date2.DayOfYear(5,5);~~       //very illegal
```

A constructor **CANNOT** be called in the same way as an ordinary member function is called



Constructor Definition (1)

- **Constructor definition**
  - Like any member function
  - Class name occurs twice in the function heading
  - No return type!

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
{
  month = monthValue;
  day = dayValue;
}
```

Note:
Constructor is used to **initialize** objects

Constructor Definition (2)

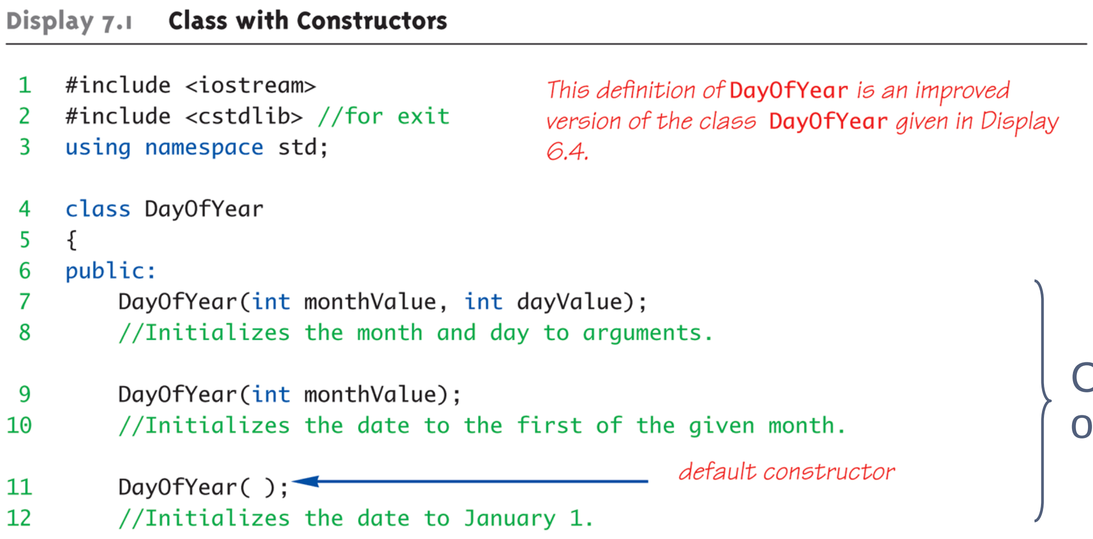
- **An alternative way**
  - Preferable to use
  - Initialization section
    - Colon
    - A list of some or all the member variables separated by commas
    - Syntax: *member_variable(initializing_value)*

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
                    : month(monthValue), day(dayValue)
{

  //empty or checks or ...

}
```

Constructor Definition (3)

- **Overloading a constructor**
  - Allowable and common (cf. function overloading)

    → Objects can be initialized in more than one way
  - Default constructor:
    - constructor with *no* arguments



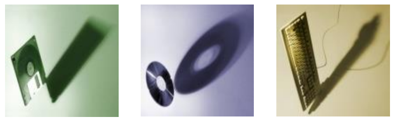
Display 7.1 **Class with Constructors**

```
#include <iostream>
#include <cstdlib> //for exit
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    //Initializes the month and day to arguments.

    DayOfYear(int monthValue);
    //Initializes the date to the first of the given month.

    DayOfYear( );
    //Initializes the date to January 1.
```

This definition of **DayOfYear** *is an improved version of the class* **DayOfYear** *given in Display 6.4.*

default constructor

Constructor overloading

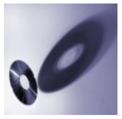


Default Constructor

- **Auto-generated?**
  - **Yes**, automatically
    - If you define a class including **no constructors** of any kind
    - This automatically-created constructor does nothing
  - **No**, manually
    - If you define a class including **one or more** constructors of any kind
- **Why?** (hint: What happen if there is no default constructor)

```
DayOfYear date3;            //illegal when no default constructor exists
```

→ Always include a default constructor



Explicit Constructor Calls

- **Invocation of constructors**
  - Implicit: Whenever your **declare** an object of the class type
  - Explicit: **After** the object has been declared
    - Creates an *anonymous object*
    - Convenient way to set members of an object!
    - In action: `DayOfYear(3, 21)`
      - Explicit constructor call
      - Returns new *anonymous object*
      - **Assigned** back to current object

Say goodbye to **`set`**(…)?

```
DayOfYear date3;
                    //object date3 has been declared
date3 = DayOfYear(3, 21);
date3 = DayOfYear(1, 27);
```

⇔

```
int month;
                    //month has been declared
month = 3;          //=int(3)
month = 1;          //=int(1)
```

Example of Constructors (1)

Display 7.1 **Class with Constructors**

```
#include <iostream>
#include <cstdlib> //for exit
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    //Initializes the month and day to arguments.

    DayOfYear(int monthValue);
    //Initializes the date to the first of the given month.

    DayOfYear( ); ←——————————— default constructor
    //Initializes the date to January 1.

    void input( );
    void output( );
    int getMonthNumber( );
    //Returns 1 for January, 2 for February, etc.
```

This definition of `DayOfYear` *is an improved version of the class* `DayOfYear` *given in Display 6.4.*

Have removed the member function **`set`**

→ replaced with constructor definitions

Example of Constructors (2)

```cpp
        int getDay( );
    private:
        int month;
        int day;
        void testDate( );
    };

    int main( )
    {
        DayOfYear date1(2, 21), date2(5), date3;
        cout << "Initialized dates:\n";
        date1.output( ); cout << endl;
        date2.output( ); cout << endl;
        date3.output( ); cout << endl;

        date1 = DayOfYear(10, 31);
        cout << "date1 reset to the following:\n";
        date1.output( ); cout << endl;
        return 0;
    }

    DayOfYear::DayOfYear(int monthValue, int dayValue)
                              : month(monthValue), day(dayValue)
    {
        testDate( );
    }
```

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

an explicit call to the constructor `DayOfYear::DayOfYear`

→ replace member function **`set`**

Example of Constructors (3)

```cpp
DayOfYear::DayOfYear(int monthValue) : month(monthValue), day(1)
{
    testDate( );
}

DayOfYear::DayOfYear( ) : month(1), day(1)
{/*Body intentionally empty.*/}

//uses iostream and cstdlib:
void DayOfYear::testDate( )
{
    if ((month < 1) || (month > 12))
    {
        cout << "Illegal month value!\n";
        exit(1);
    }
    if ((day < 1) || (day > 31))
    {
        cout << "Illegal day value!\n";
        exit(1);
    }
}
```

<Definitions of the other member functions are the same as in Display 6.4.>

Sample Dialogue

```
Initialized dates:
February 21
May 1
January 1
date1 reset to the following:
October 31
```

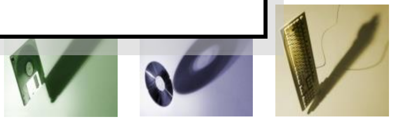
Class Type Member Variables

- **Class member variables**
  - Can be **any type**, including another class
  - Member objects or member classes?

```
class Holiday
{
public:
    Holiday( );//Initializes to January 1 with no parking enforcement
    Holiday(int month, int day, bool theEnforcement);
    void output( );
private:
    DayOfYear date;
    bool parkingEnforcement;//true
};
```

```
class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    DayOfYear(int monthValue);
    DayOfYear( );
    void input( );
    void output( );
    int getMonthNumber( );
    int getDay( );
private:
    int month;
    int day;
    void testDate( );
};
```

The class DayOfYear
Display 7.1, but we hav
details you need for t



Class Type Member Variables (cont'd)

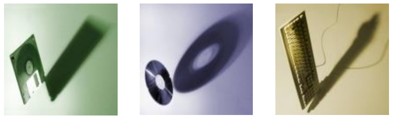
- **Need special notation for constructors**
  - So they can call "back" to the constructor of member object

```
int main( )
{
    Holiday h(2, 14, true);
    cout << "Testing the class Holiday.\n";
    h.output( );

    return 0;
}

Holiday::Holiday( ) : date(1, 1), parkingEnforcement(false)
{/*Intentionally empty*/}

Holiday::Holiday(int month, int day, bool theEnforcement)
                      : date(month, day), parkingEnforcement(theEnfor
{/*Intentionally empty*/
```

Invocations of constructors from the class `DayOfYear`.

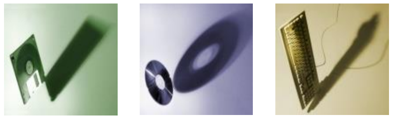
```
class Holiday
{
public:
    Holiday( );//Initializes
    Holiday(int month, int d
    void output( );
private:
    DayOfYear date;
    bool parkingEnforcement;
};
```

- Invocation of **constructor** from class `DayOfYear`
- To initialize the member variables of **object** `date`



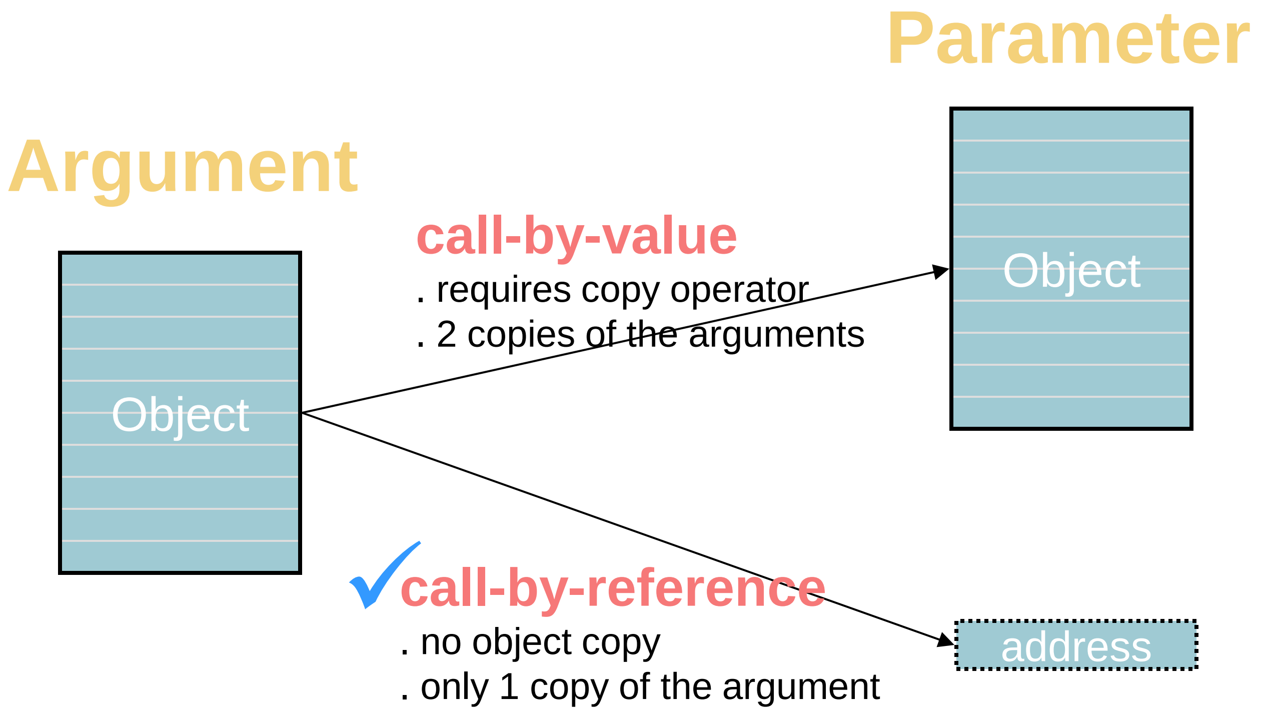
Outline

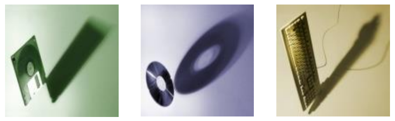
- Constructors
- **More tools**
  - The `const` parameter modifier
  - Inline functions
  - Static members
- Vectors



The `const` parameter modifier (1)

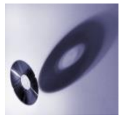
- **Which one is more efficient?**





The `const` parameter modifier (2)

- **Call-by-reference parameter**
  - Is preferable for large data type parameters (**class**, **array**, etc.)
  - Protect arguments → constant parameter
    - Place modifier `const` before data type (class)
    - Make it “read-only”
    - Automatic error checking by compiler
- **Calling objects**
  - Protect calling objects
    - e.g. member function `output` should not change the values of the calling object’s member variables
    - Place modifier `const` **at the end** of function declaration (just before semicolon)



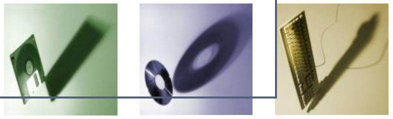
The const parameter modifier (3)

```
class BankAccount
{
  public:
    BankAccount(int dollars, double rate);
    BankAccount();
    void input();
    void output() const;
    int getDollars() const;
    double getRate() const;
  private:
    int accountDollars;
    double rate;
    int round(double number) const;
}

bool isLarger(const BankAccount& account1, const BankAccount& account2);

int main()
{
  ...
}

bool isLarger(const BankAccount& account1, const BankAccount& account2)
{
  return(account1.getDollars() > account2.getDollars());
}

void bankAccount::output() const
{
  ...
}
```

Place **const** modifier in both *declaration* and *definition*

Protect calling object

Protect arguments



The const parameter modifier (4)

- **Use of const**
  - All-or-nothing!
  - → You should tell compiler ***wherever*** not to change the parameters
    - By default, compiler assumes the calling object will be changed

```
class BankAccount
{
  public:
    ...
    void output() const;
    ...
}

void welcome(const BankAccount& yourAccount)
{
  cout << “Welcome to our bank. \n”
       << “The status of you account is: \n”;
  yourAccount.output();       //Error if no const for output()
}
```

protect **argument**

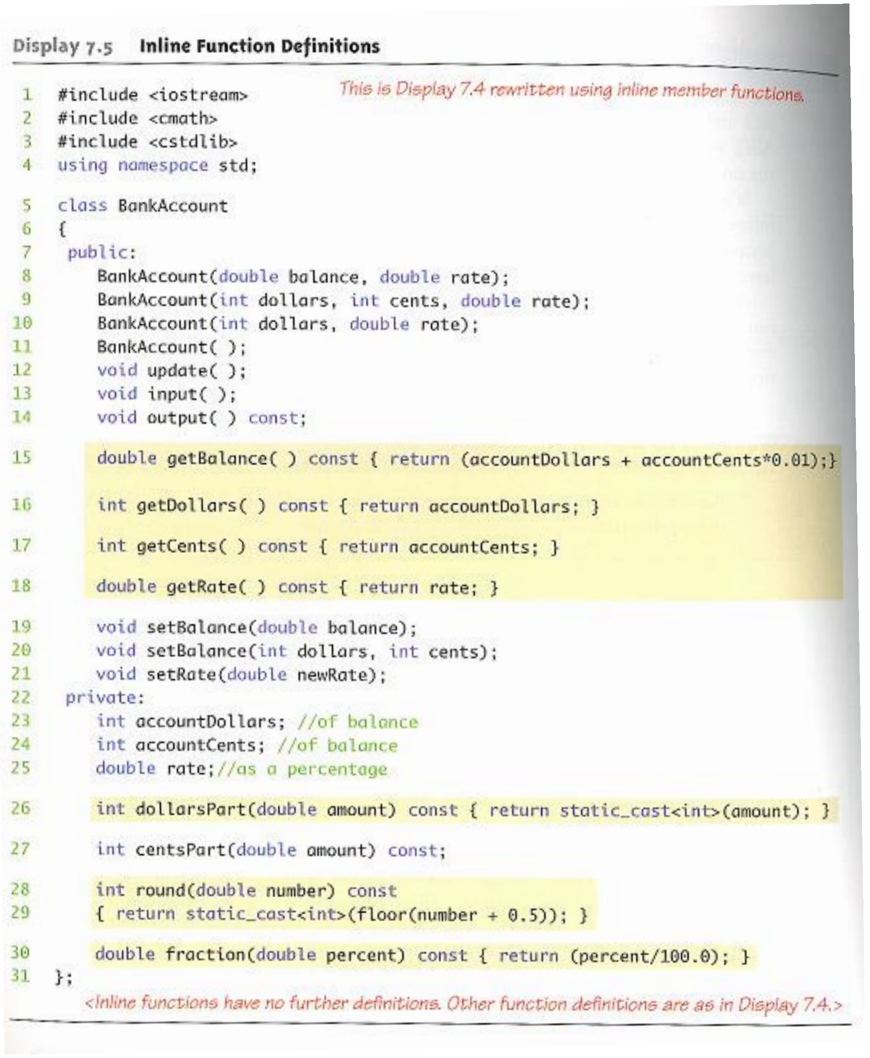
need protection → protect **calling object** (yourAccount)

Inline Functions (1)

- **For non-member functions**
  - Use keyword **`inline`** in function declaration and function heading
- **For member functions**
  - Defining a member function **within** the definition of its class → automatically inline
- **Use of Inline Functions**
  - Only for short functions
  - Code is literally copied and inserted in place of function invocation

    (Recall: **`#define`** statement in C language)

Inline Functions (2)


Display 7.5 **Inline Function Definitions**

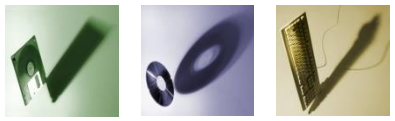
This is Display 7.4 rewritten using inline member functions.

```cpp
#include <iostream>
#include <cmath>
#include <cstdlib>
using namespace std;

class BankAccount
{
 public:
    BankAccount(double balance, double rate);
    BankAccount(int dollars, int cents, double rate);
    BankAccount(int dollars, double rate);
    BankAccount( );
    void update( );
    void input( );
    void output( ) const;

    double getBalance( ) const { return (accountDollars + accountCents*0.01);}

    int getDollars( ) const { return accountDollars; }

    int getCents( ) const { return accountCents; }

    double getRate( ) const { return rate; }

    void setBalance(double balance);
    void setBalance(int dollars, int cents);
    void setRate(double newRate);
 private:
    int accountDollars; //of balance
    int accountCents; //of balance
    double rate;//as a percentage

    int dollarsPart(double amount) const { return static_cast<int>(amount); }

    int centsPart(double amount) const;

    int round(double number) const
    { return static_cast<int>(floor(number + 0.5)); }

    double fraction(double percent) const { return (percent/100.0); }
};
```

<Inline functions have no further definitions. Other function definitions are as in Display 7.4.>

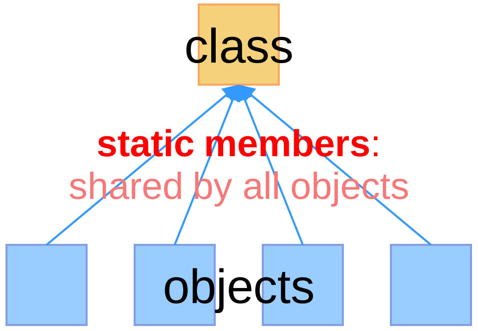
Inline Functions (3)

- **Pros**
  - Eliminates overhead
  - More efficient, but only when short
- **Cons**
  - Go against the principle of *encapsulation*, because of mixing the interface and implementation of a class
  - Less efficient for long function definitions, since a large piece of code is repeated frequently



Static Members (1)

- **Static variables**
  - Variables that are **shared** by all the *objects* of a class
  - One object changes → All objects know it
  - Used for objects of the class
    - To communicate with each other
    - To coordinate their actions
    - Have the advantages of global variables without opening the flood gates to abuses
  - Can be *private* → only objects of the class can directly access it



(have their own members)

Static Members (2)

- **Static variables** (cont'd)
  - Place keyword `static` before type
  - CanNOT be initialized more than once
  - Must be initialized ***outside*** the class definition (why?)

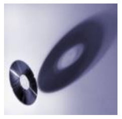
```cpp
#include <iostream>
using namespace std;

class Server
{
public:
    Server(char letterName);
    static int getTurn( );
    void serveOne( );
    static bool stillOpen( );
private:
    static int turn;
    static int lastServed;
    static bool nowOpen;
    char name;
};

int Server:: turn = 0;
int Server:: lastServed = 0;
bool Server::nowOpen = true;
```

Initialization (just once)

Contrary to "private"?

The author of a class is expected to do the initialization in the same file as the class definition

→ no programmer who uses the class by including it can initialize static (cannot twice!)



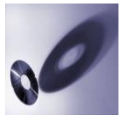
Static Members (3)

- **Static functions**
  - Does not access the data of any *object*
  - A member of the class

  → Functions that deal with class-level matters
    - Cannot use anything that depends on a calling object
    - Use only
      - static variables
      - static member functions
      - local variables (local objects)

Static Members (4)

- **Static functions** (cont'd)
  - Place keyword `static` only in ***declaration***, but NOT in *definition*
  - Function call outside class
    - Common way (nothing with object)
      - e.g. `Server::getTurn();`
    - Using a calling object
      - e.g. `myObject.getTurn();`



Example of Static Members (1)

Display 7.6 Static Members

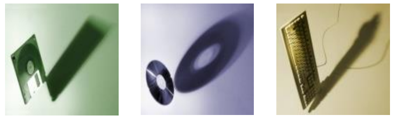
```
#include <iostream>
using namespace std;

class Server
{
public:
    Server(char letterName);
    static int getTurn( );
    void serveOne( );
    static bool stillOpen( );
private:
    static int turn;
    static int lastServed;
    static bool nowOpen;
    char name;
};

int Server:: turn = 0;
int Server:: lastServed = 0;
bool Server::nowOpen = true;
```

static functions

static variables

initialization

```
int main( )
{
    Server s1('A'), s2('B');
    int number, count;
    do
    {
        cout << "How many in your group? ";
        cin >> number;
        cout << "Your turns are: ";
        for (count = 0; count < number; count++)
            cout << Server::getTurn( ) << ' ';
        cout << endl;
        s1.serveOne( );
        s2.serveOne( );
    } while (Server::stillOpen( ));

    cout << "Now closing service.\n";

    return 0;
}
```



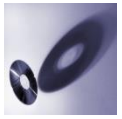
Example of Static Members (2)

```
Server::Server(char letterName) : name(letterName)
{/*Intentionally empty*/}

int Server::getTurn( )
{
    turn++;
    return turn;
}
bool Server::stillOpen( )
{
    return nowOpen;
}

void Server::serveOne( )
{
    if (nowOpen && lastServed < turn)
    {
        lastServed++;
        cout << "Server " << name
            << " now serving " << lastServed << endl;
     }
    if (lastServed >= turn) //Everyone served
        nowOpen = false;
}
```

Since `getTurn` *is static, only static members can be referenced in here.*

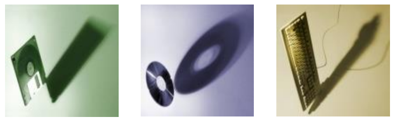
Sample Dialogue

```
How many in your group? 3
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? 2
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? 0
Your turns are:
Server A now serving 5
Now closing service.
```



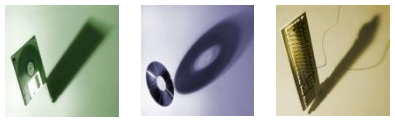
Outline

- Constructors
- More tools
  - The `const` parameter modifier
  - Inline functions
  - Static members
- Vectors



Vectors

- **Arrays that can grow and shrink**
  - Changeable length while program is running
  - Arrays: fixed size
- **Formed from Standard Template Library (STL)**
  - Template class
    - Can be plugged in any data type



Vector Basics (1)

- **A vector**
  - Has a base type
  - Stores a collection of values of its base type
  - Syntax:

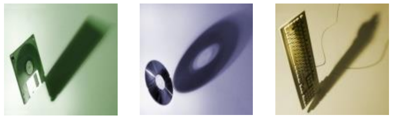
```
vector<Base_Type> Vec_Name;
```

    - Different from the syntax for arrays
    - Template class → a class for vectors with Base_type
    - e.g.

```
vector<int> v;
```

class name
- includes base type
- creates a vector object that is empty

vector object
v is a vector of type `int`



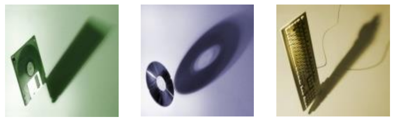
Vector Basics (2)

- **Use an element** – same as arrays
  - Index starts with 0
  - Square bracket notation to read or change

```
v[i] = 42;
cout << “The answer is ” << v[i];
```

- **Add an element**
  - Member function `push_back`

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.2);
sample.push_back(7.5);
```



Vector Basics (3)

- **Member function `size()`**
  - Returns the current number of elements
  - Type: `unsigned int`

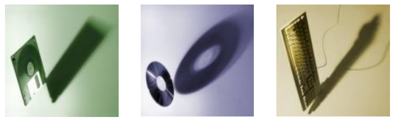
- **Initialization**
  - Vectors with *predefined type*

    ```
    vector<int> v(10);
    ```

    - Initializes the first 10 elements to 0
    - v.size() returns 10
  - Vectors with *class type*

    ```
    vector<DayOfYear> v(10);
    ```

    - Initializes the first 10 elements by the default constructor
    - Actually, default constructor `int()` returns 0



Example of Vectors

Display 7.7 **Using a Vector**

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> next;
    }
    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

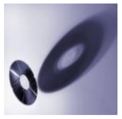
SAMPLE DIALOGUE

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size = 1
4 added. v.size = 2
6 added. v.size = 3
8 added. v.size = 4
You entered:
2 4 6 8
```

Efficiency Issues

- **`Capacity()` vs. `Size()`**
  - Size: the number of elements in a vector
  - Capacity: the number of elements that a vector has memory allocated
- **Capacity**
  - Capacity ≥ Size
  - Is automatically increased (typically, double it)
  - Efficiency → Manage capacity yourself
    - Member function **`reserve()`**
    - e.g.

```
v.reserve(32);
v.reserve(v.size() +10);
```



Vector Assignment

- **Well-Behaved**
  - The assignment operator (=) with vectors does an element-by-element assignment
  - The left-hand side
    - Increases capacity if needed
    - Resets the size of the vector
  - To produce a totally independent copy?
    - **Depends on** the assignment operator of the base type

Summary (1)

- **Constructors**
  - A member function of a class that is called automatically when an object is declared
    - Automatic initialization of class data
    - Have the same names as the class
  - Default constructor
    - A constructor with no parameters
    - Always define a default constructor
  - Constructor Invocation
    - Whenever your declare an object of the class type
    - After the object has been declared → Explicit constructor calls

Summary (2)

- **More tools**
  - The `const` parameter modifier
    - Call-by-reference is more efficient
    - Protect argument
    - Protect calling object
  - Inline functions
    - Efficient for short code
  - Static members
    - Static member variables
      - Variables that are shared by all objects of a class
    - Static member functions

Summary (3)

- **Vectors**
  - Like “arrays that can grow and shrink in length”
  - Template class for vector objects with base type