Parameter Passing Methods

◆Efficiency of parameter passing

- Call-by-value
  - Requires ***copy*** be made → *Overhead*
- Call-by-reference
  - Placeholder for actual argument
  - *Most efficient method*
- Negligible difference for simple types
- For class types → clear advantage

◆Call-by-reference desirable

- Especially for "large" data, like class types

The const Parameter Modifier

Large data types (typically classes)

- Desirable to use pass-by-reference
- Even if function will not make modifications

Protect argument

- Use constant parameter
  - Also called constant call-by-reference parameter
- Place keyword *const* before type
- Makes parameter "read-only"
- Attempts to modify result in compiler error

Use of const

- **All-or-nothing**
- **If no need for function modifications**
  - Protect parameter with const
  - Protect ALL such parameters
- **This includes class member function parameters**

```
// Display 7.4 The const Parameter Modifier
#include <iostream>
#include <cmath>
#include <cstdlib>
using namespace std;

//Data consists of two items, an amount of money for the account balance
//and a percent for the interest rate.
class BankAccount
{
public:
    BankAccount(double balance, double rate);
    //Initializes balance and rate according to arguments.

    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents. For a negative balance both
    //dollars and cents must be negative. 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 update( );
    //Postcondition: One year of simple interest has been added to the account.
    void input( );
    void output( ) const;
    double getBalance( ) const;
    int getDollars( ) const;
    int getCents( ) const;
    double getRate( ) const;//Returns interest rate as a percent.

    void setBalance(double balance);
    void setBalance(int dollars, int cents);
    //checks that arguments are both nonnegative or both nonpositive

    void setRate(double newRate);
    //If newRate is nonnegative, it becomes the new rate. Otherwise abort program.

private:
    //A negative amount is represented as negative dollars and negative cents.
    //For example, negative $4.50 sets accountDollars to -4 and accountCents to -50.
    int accountDollars; //of balance
    int accountCents; //of balance
    double rate; //as a percent

    int dollarsPart(double amount) const;
    int centsPart(double amount) const;
    int round(double number) const;

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

```
//Returns true if the balance in account1 is greater than that
//in account2. Otherwise returns false.
bool isLarger(const BankAccount& account1, const BankAccount& account2);

void welcome(const BankAccount& yourAccount);

int main( )
{
   BankAccount account1(6543.21, 4.5), account2;
   welcome(account1);
   cout << "Enter data for account 2:₩n";
   account2.input( );
   if (isLarger(account1, account2))
      cout << "account1 is larger.₩n";
   else
      cout << "account2 is at least as large as account1.₩n";

      return 0;
}

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

void welcome(const BankAccount& yourAccount)
{
   cout << "Welcome to our bank.₩n"
        << "The status of your account is:₩n";
   yourAccount.output( );
}
```

```
//Uses iostream and cstdlib:
void BankAccount::output( ) const
{
   int absDollars = abs(accountDollars);
   int absCents = abs(accountCents);
   cout << "Account balance: $";
   if (accountDollars < 0)
      cout << "-";
   cout << absDollars;
   if (absCents >= 10)
      cout << "." << absCents << endl;
   else
      cout << "." << '0' << absCents << endl;

   cout << "Rate: " << rate << "%₩n";
}

BankAccount::BankAccount(double balance, double rate)
 : accountDollars(dollarsPart(balance)), accountCents(centsPart(balance))
{
   setRate(rate);
}

BankAccount::BankAccount(int dollars, int cents, double rate)
{
   setBalance(dollars, cents);
   setRate(rate);
}

BankAccount::BankAccount(int dollars, double rate)
                        : accountDollars(dollars), accountCents(0)
{
   setRate(rate);
}
```

```
BankAccount::BankAccount( ): accountDollars(0), accountCents(0), rate(0.0)
{/*Body intentionally empty.*/}

void BankAccount::update( )
{
   double balance = accountDollars + accountCents*0.01;
   balance = balance + fraction(rate)*balance;
   accountDollars = dollarsPart(balance);
   accountCents = centsPart(balance);
}

//Uses iostream:
void BankAccount::input( )
{
   double balanceAsDouble;
   cout << "Enter account balance $";
   cin >> balanceAsDouble;
   accountDollars = dollarsPart(balanceAsDouble);
   accountCents = centsPart(balanceAsDouble);
   cout << "Enter interest rate (NO percent sign): ";
   cin >> rate;
   setRate(rate);
}

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

int BankAccount::getDollars( ) const
{
   return accountDollars;
}
```

```
int BankAccount::getCents( ) const
{
   return accountCents;
}

void BankAccount::setBalance(double balance)
{
   accountDollars = dollarsPart(balance);
   accountCents = centsPart(balance);
}

//Uses cstdlib:
void BankAccount::setBalance(int dollars, int cents)
{
   if ((dollars < 0 && cents > 0) || (dollars > 0 && cents < 0))
   {
      cout << "Inconsistent account data.₩n";
      exit(1);
   }
   accountDollars = dollars;
   accountCents = cents;
}

double BankAccount::getRate( ) const
{
   return rate;
}

//Uses cstdlib:
void BankAccount::setRate(double newRate)
{
   if (newRate >= 0.0)
      rate = newRate;
   else
   {
      cout << "Cannot have a negative interest rate.₩n";
      exit(1);
   }
}
```

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

//Uses cmath:
int BankAccount::centsPart(double amount) const
{
   double doubleCents = amount*100;
   int intCents = (round(fabs(doubleCents)))%100;//% can misbehave on negatives
   if (amount < 0)
      intCents = -intCents;
   return intCents;
}

//Uses cmath:
int BankAccount::round(double number) const
{
   return floor(number + 0.5);
}

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

Inline Functions

- **For non-member functions:**
  - Use keyword ***inline*** in function declaration and function heading
- **For class member functions:**
  - Place implementation (code) for function IN class definition → automatically inline
- **Use for very short functions only**
- **Code actually inserted in place of call**
  - Eliminates overhead
  - More efficient, but only when short!

Inline Member Functions

- ◆ **Member function definitions**
  - Typically defined separately, in different file
  - Can be defined IN class definition
    - Makes function "in-line"
- ◆ **Again: use for very short functions only**
- ◆ **More efficient**
  - If too long → actually less efficient!

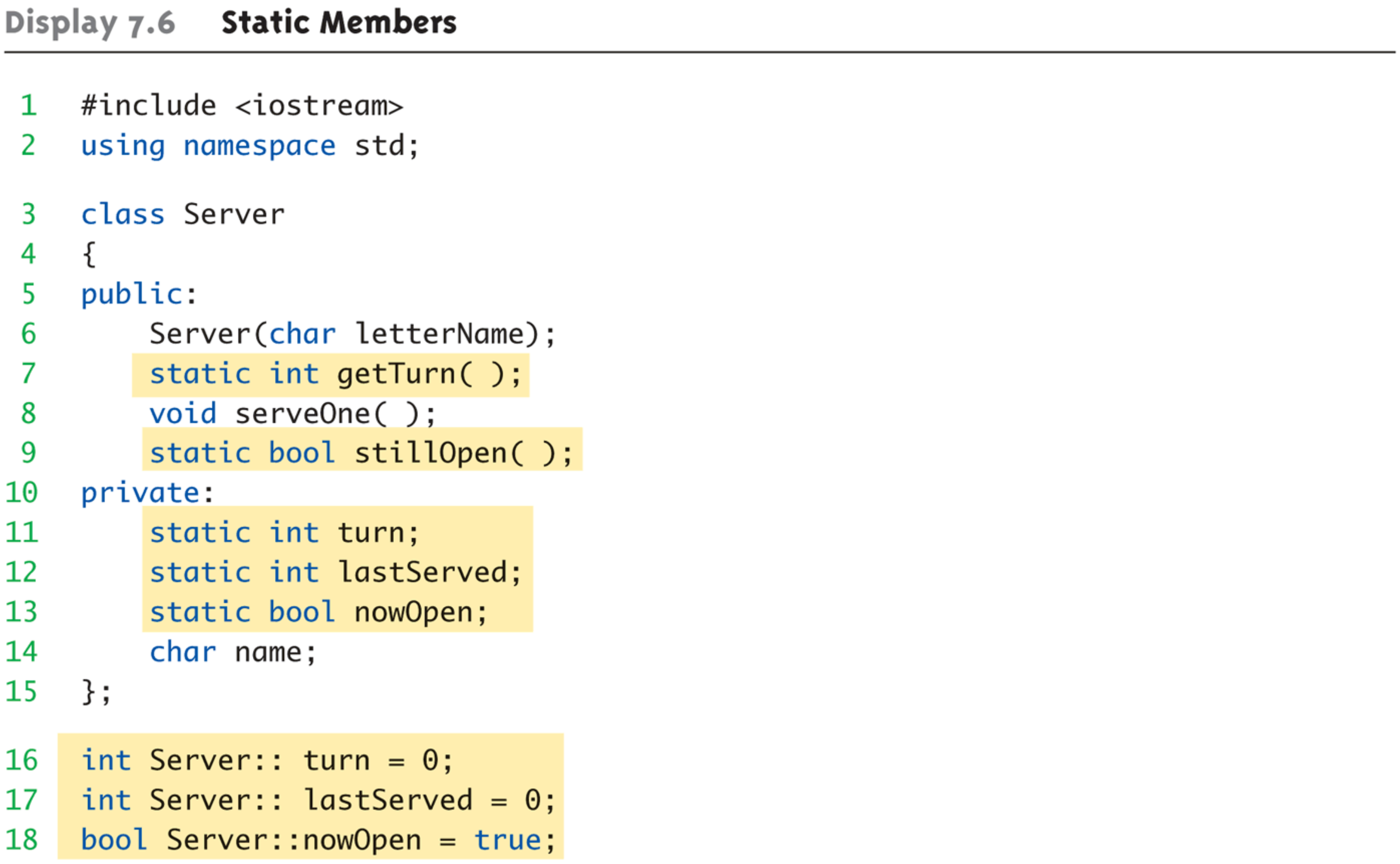
Static Members

- **Static member variables**
  - All objects of class "share" one copy
  - One object changes it → all see change
- **Useful for "tracking"**
  - How often a member function is called
  - How many objects exist at given time
- **Place keyword *static* before type**

Static Functions

- **Member functions can be static**
  - If no access to object data needed
  - And still "must" be member of the class
  - Make it a static function
- **Can then be called outside class**
  - From non-class objects:
    - E.g., Server::getTurn();
  - As well as via class objects
    - Standard method: myObject.getTurn();
- **Can only use static data, functions!**

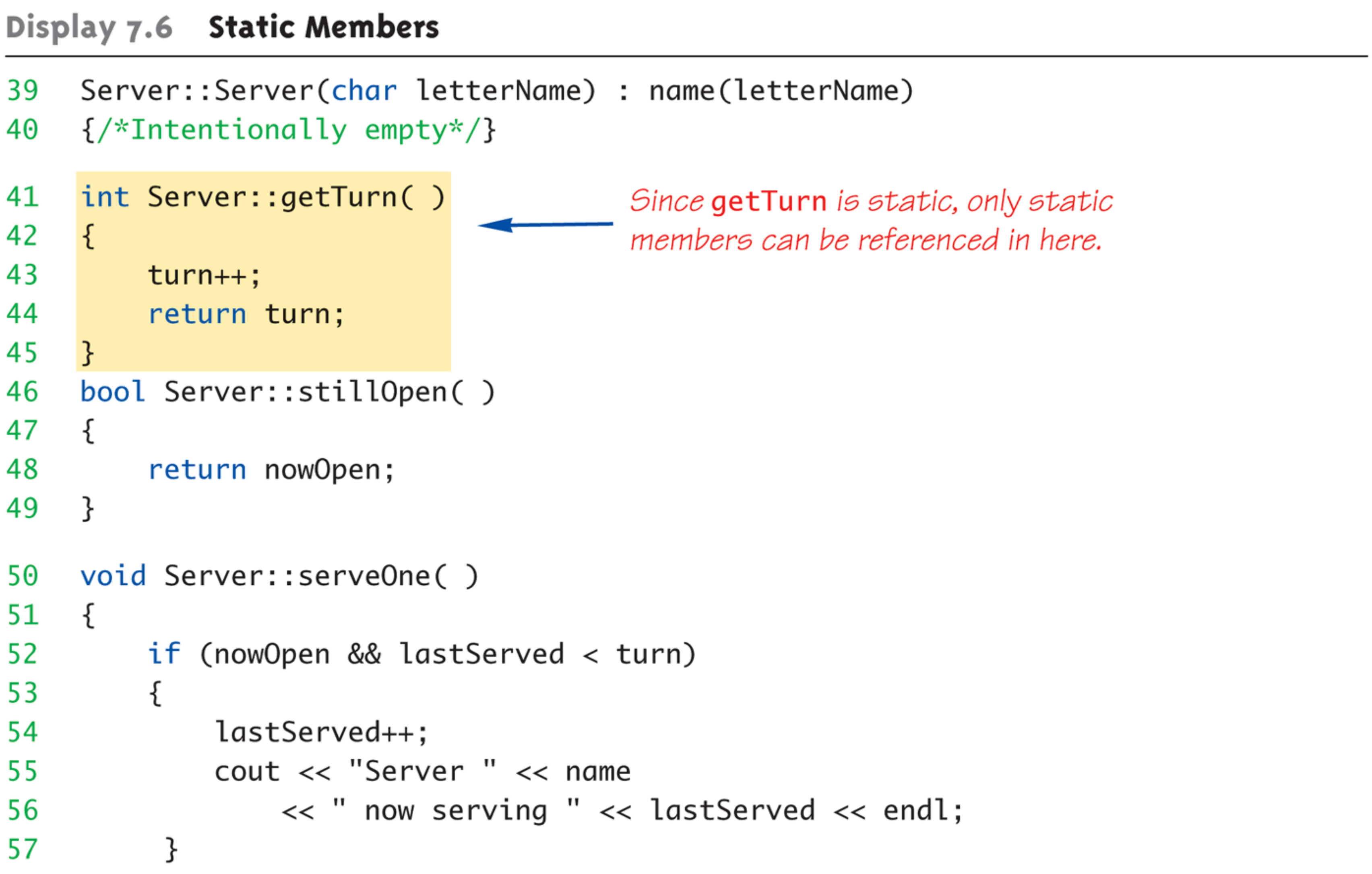
Static Members Example



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;
```

```
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;
}


```



Display 7.6 Static Members

```
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;
     }
```

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

```
    if (lastServed >= turn) //Everyone served
        nowOpen = false;
}
```

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.
```

Summary 7-2-1

- **Constructors: automatic initialization of class data**
  - Called when objects are declared
  - Constructor has same name as class
- **Default constructor has no parameters**
  - Should always be defined
- **Destructor is used to clean-up dynamically allocated resources**
- **Class member variables**
  - Can be objects of other classes
    - Require initialization-section

Summary 7-2-2

- **Constant call-by-reference parameters**
  - More efficient than call-by-value
- **Can *inline* very short function definitions**
  - Can improve efficiency
- **Static member variables**
  - Shared by all objects of a class

Homework 7-2

7-2.1 class Mtrx

Program a header file “Class_Mtrx.h” with a “class Mtrx” with following members:

```
/* Class_Mtrx.h */

class Mtrx {
public:
            Mtrx(int mSize); // constructor
            Mtrx(double dA[], int num_data, int mSize); // constructor
            ~Mtrx(); // destructor
            void print();
            Mtrx add(Mtrx);
            Mtrx subtract(Mtrx);
            Mtrx multiply(Mtrx);
private:
            int mSize;
            double **dM;
            double det;
};
```

7-2.2 Write a C++ "mtrx.cpp" that implements the member functions:

(1) Mtrx(int mSize);

- constructor that dynamically creates two dimensional array for the data member dM[mSize][mSize], and initialize the dM[][] with initial values of 0.0

(2) Mtrx(double dA[], int num_data, int mSize);

- constructor that receives an array dA[num_data], and dynamically creates two dimensional array for the data member dM[mSize][mSize], and initialize the dM[][] with the given data

(3) ~Mtrx();

- Destructor that deletes the dynamically allocated memory for two dimensional array

(4) void print();

- prints out the dM[][]

(5) Mtrx add(Mtrx mA);

- creates a Mtrx mR and calculates the addition of mR.dM[][] = this.dM[][] + mA.dM[][], and returns the mR

(6) Mtrx subtract(Mtrx mA);

- creates a Mtrx mR and calculates the subtraction of mR.dM[][] = this.dM[][] - mA.dM[][], and returns the mR

(7) Mtrx multiply(Mtrx);

- creates a Mtrx mR and calculates the addition of mR.dM[][] = this.dM[][] x mA.dM[][], and returns the mR

7-2.3 Use following main() function, and produce the results.

```
/* main.cpp */
#define SIZE_N 5

void main()
{
    double mA[SIZE_N*SIZE_N] =
             { 1.0, 2.0, 3.0, 4.0, 5.0,
               2.0, 3.0, 4.0, 5.0, 1.0,
               3.0, 2.0, 5.0, 3.0, 2.0,
               4.0, 3.0, 2.0, 7.0, 2.0,
               5.0, 4.0, 3.0, 2.0, 9.0 };

    double mB[SIZE_N*SIZE_N] =
             { 1.0, 0.0, 0.0, 0.0, 0.0,
               0.0, 1.0, 0.0, 0.0, 0.0,
               0.0, 0.0, 1.0, 0.0, 0.0,
               0.0, 0.0, 0.0, 1.0, 0.0,
               0.0, 0.0, 0.0, 0.0, 1.0 };
```

```
        Mtrx mtrxA(mA, SIZE_N*SIZE_N, SIZE_N);
        cout <<"MtrxA:₩n";
        mtrxA.print();

        Mtrx mtrxB(mB, SIZE_N*SIZE_N, SIZE_N);
        cout <<"MtrxB:₩n";
        mtrxB.print();

        Mtrx mtrxC(SIZE_N);
        cout <<"MtrxC:₩n";
        mtrxC.print();

        mtrxC = mtrxA.add(mtrxB);
        cout <<"MtrxC = mtrxA.addMtrx(mtrxB) :₩n";
        mtrxC.print();

        mtrxD = mtrxA.subtract(mtrxB);
        cout <<"MtrxC = mtrxA.subtractMtrx(mtrxB) :₩n";
        mtrxD.print();

        Mtrx mtrxE(SIZE_N);
        mtrxE = mtrxA.multiply(mtrxD);
        cout <<"MtrxE = mtrxA.multiply(mtrxD) :₩n";
        mtrxE.print();

}  // end main()
```